NXP Semiconductors

Document Number: QORIQSDK21703
Rev. 0, 03/2017

QorIQ SDK v2.0-1703 Documentation

For QorIQ Processors

Contents

Chapter 1
Welcome to the Linux SDK for QorIQ Processors Knowledge Center

QorIQ LS Series Communications Processors

The QorIQ communications processor portfolio is unmatched in depth and breadth. With the addition of the next-generation QorIQ LS1 and LS2 product families, the portfolio extends performance from the smallest form factor, power-constrained networking applications to software-defined networking applications requiring an advanced datapath and network peripheral interfaces. The LS1 and LS2 families based on ARM® technology offer optimized performance and power efficiency, plus pin compatibility, enabling customers to simply and smoothly migrate applications between these next-generation QorIQ devices.

QorIQ T Series Communications Processors

The QorIQ T series communications processors introduced 28nm to the portfolio in 2010 with scalable, pin-compatible 64-bit and multi-threaded Power Architecture cores, new and improved acceleration engines for fixed function processing as well as advanced power management technology. The T series products offer up to 4x performance improvements over the previous generation and reintroduces the highly-popular AltiVec vector processing unit for high-bandwidth data processing and algorithmic-intensive computations.

QorIQ P Series Communications Processors

The QorIQ P series communications processors launched in 2008 as an evolution of our leading PowerQUICC line. The devices are designed on 45 nm process technology to reduce power and increase integration – offering some of the industry's best performance to power ratios. The P series processors integrate enhanced Power Architecture cores, ranging from single to eight core devices, and include hallmark features such as CoreNet coherency fabric and acceleration engines for NXP's data path acceleration architecture (DPAA) including security and pattern matching. Today the P series is 25 products strong and powers solutions across every node of the network.

QorIQ Qonverge Series Baseband Processors

The QorIQ Qonverge portfolio of multimode solutions for femtocell, picocell, metrocell and macrocell base station applications support WCDMA, LTE and LTE-Advanced. The baseband SoC heterogeneous platform combines high-performance, market-proven 32-bit & 64-bit multi-threaded Power Architecture® CPU cores and performance leading StarCore DSP cores, featured with application acceleration platforms such as the MAPLE-B for baseband processing and DPAA for data path processing, security acceleration and more. The QorIQ Qonverge portfolio spans across world-class silicon process nodes of 45nm and 28nm already in production and beyond. NXP provides comprehensive solutions for the QorIQ Qonverge platform with ready-to-go software, ecosystem, software and hardware development tools to ease development and speed time to market for wireless base station applications.

See also the Linux SDK for QorIQ Processors summary page.

Chapter 2
SDK Overview

2.1 Introduction

This topic provides an introduction to the NXP Linux Software Development Kit, including intended audience, purpose and scope, and detailed sections on technical considerations.

Intended Audience

NXP SDK documentation is written for software developers and system engineers who have a technical background, and a working knowledge of Linux. The audience for the NXP QorIQ Linux SDK are users of QorIQ Communication, QorIQ Qonverge, and select PowerQUICC Communications Processors.

Overview of the SDK: Purpose and Scope

The NXP SDK is a Linux-oriented development kit. It allows users to

1. Evaluate and explore NXP SoC processors' features; experience how they are supported in Linux by working "out of the box" on NXP development boards.
2. Develop Linux-based solutions; the NXP SDK is thoroughly tested and backed by NXP software and services support from development to production.

One question might be why is this called a Software Development Kit (SDK) rather than a board support package (BSP)? The reason is that the SDK contains more than a single BSP; it supports numerous NXP processors and contains all the tools, accelerators, and unique features supported in the particular release. Many customers never need additional tools to get into production.

Normally a complete Linux environment is called a Linux "distribution" and contains a boot loader, the Linux kernel and user space components, a tool chain, a build system, and a package manager. All of these building blocks are included in the SDK, and can be used across many products.

The SDK is based on a widely-supported and open source community-oriented Linux Foundation effort called the Yocto Project and its' Poky software; for more information see http://www.yoctoproject.org. The SDK is a Linux-distribution created using Yocto Project/Poky that provides integrated SoC software support. The SDK includes support for the NXP Power™ architecture and ARM cores, and drivers for NXP-specific peripheral hardware.

Yocto Project/Poky distributions are embedded-oriented, and Yocto Project/Poky uses the OpenEmbedded-core package set. Embedded distributions use cross-compilation, and are capable of producing focused target images that are efficiently sized. Yocto Project/Poky is flexible, and its build system can also produce larger images with features such as target-resident (native) tool chains; the NXP SDK leverages this flexibility, and supports a wide range of pre-defined configurations.

The NXP SDK strategy is to use Yocto Project/Poky, and enhance it with NXP-specific SoC support; this way the basic SDK framework is familiar to anyone who has used Yocto Project/Poky on another architecture. Yocto Project resources, such as documentation, now become useful and relevant for development with the NXP SDK. The use of Yocto Project/Poky provides a straight-forward way to achieve a minimum footprint and maximum performance for your project.

Boot Loader

The SDK includes a popular and widely-used open source community boot loader called U-Boot. NXP, along with community developers, add support for NXP SoCs and peripherals. See www.denx.de/wiki/U-Boot.

NXP recommends and provides support services when using the CodeWarrior embedded software development studio and a CodeWarrior TAP to program U-Boot into flash memories for the first time.

Linux Kernel

The SDK is based upon the standard Linux kernel from http://www.kernel.org, but with patches added by the Yocto Project, NXP, and other community members to support NXP SoCs. Much of the NXP-supplied Linux software is within the kernel; NXP is one of the top 20 global kernel.org contributors.

One or more times per year NXP will upgrade to the latest long-term supported kernel version; then throughout the year, the team will accumulate changes for that kernel version. Occasionally, NXP will produce an “off cycle” SDK release that adds specific support for a new IC using patches. NXP is experienced with porting code to the latest kernels, across several product lines and architectures, when it is needed by our customer base.

Standard User Space

Many user space components, such as applications, initialization scripts, shells, etc., are architecture-independent so that the same software is built for the needed target architecture. For the most part, the NXP SDK provides these components unchanged from Yocto Project/Poky, and the build system compiles them for cores based on Power Architecture© or ARM technology.

The SDK includes upwards of 2,500 packages that add up to 15+ million lines of code, and NXP offers reference designs in our test benches. For example, we provide a storage area networking reference design to demonstrate how NXP processors and the SDK can work for those applications.

NXP User Space

NXP also supplies its own user space software as part of the SDK. QorIQ Communications Processes have extensive support for direct userspace access to NXP's DPAA and DPAA2 hardware blocks. Opensource packages like OpenDataPlane (ODP) and Data Plane Development Kit (DPDK) coupled with NXP enablement, and NXP's USDPAA package, provide a large set of userspace drivers and sample applications to enable this functionality. ODP, DPDK and USDPAA are packaged in SDK as Yocto Project/Poky packages with standard open-embedded recipes.

Virtualization

Virtualization provides an environment that enables multiple virtual machines (or partitions) on a single system. NXP offers two virtualization solutions as part of the SDK: a Linux-based KVM and the NXP Embedded Hypervisor. NXP tests these configurations to ensure quality. We have customers who use these configurations to keep from rewriting software to amalgamate separate boards together into a single board. The support also includes running OVS in kernel or userspace along with Virtual Machine running kernel or DPDK based user space networking.

Tool Chain

The SDK uses the GNU tool chain, but with additional patches supplied by the Yocto Project, other community members, and NXP. The SDK tool chain supports C and C++, and cross and native compiling; patches that provide the latest optimization for cores based on Power Architecture© or ARM technology are included. The standard gdb debugger, and other tools such as binutils , are also supported. To speed development time, NXP also has a commercial software development system called CodeWarrior Development Studio that compliments the SDK. Find that software here .

Build System

The SDK uses the Poky build system that uses the OpenEmbedded-core. The SDK strategy is to use this component in the same fashion across all architectures. To reduce build times, Yocto Project/Poky supports caching binaries.

Package Management

The SDK supports package management and selection using standard Yocto Project/Poky methods. The distribution is source based, and source code is provided for most packages and all packages are provided under the GNU Public License.

SDK System Images

As mentioned above, the SDK can generate several different user space system images that range in size from large to small. Users who are new to the SDK may wish to start by booting a system using" fsl-image-full." In fact, this image is pre-installed on many NXP development boards.

2.2 What's New

NXP Digital Networking is pleased to announce the release of QorIQ Linux SDK V2.0 supporting our QorIQ family of processors. All processors and boards supported in QorIQ SDK V2.0 utilize a common source base.

What's New in SDK V2.0-1703

Highlights

- Integrates off-train release for LS2088A BSP v0.3
- Integrates off-train release for LS1012A BSP v0.5
- Removal of LS2080A and LS2085A-RDB
- MC 10.1.2 update
- Added DPDK support for DPAA2 platforms (LS2088A)
- Integrates Open Data Plane (ODP) for LS2088A
- Includes several software fixes. See Fixed, Open, and Closed Issues on page 64 section which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-009241, A-009942, A-010554, A-010635, A-010812

See list of changes in below

Processor and Board Support

- LS2088A r1.0 and Rev.D RDB, Rev. F RDB
- Removal of LS2080A and LS2085A-RDB
- LS1012A RDB revC and FRDM revD

Linux Kernel Core, Virtualization

- LS2088A: libvirt, LXC, docker engine

Linux Kernel Drivers

- LS2088A: DUART, DDR4, I2C, PCIe, SATA, USB, SD, MMC, NOR, NAND flash, Networking support, SEC
- LS1012A: 2.5G SGMII, Checksum offload to PFE hardware
- CPU Frequency on LS1012A
- DPAA2 ethernet: socket busy poll, IEEE802.3x flow control, tx congestion management
- eMMC HS200 and SD UHS-I on LS1012A and LS1046A
- PFE driver: Checksum offload to PFE hardware

Data Plane Development Kit (DPDK)

- DPDK v16.07.2 Support with Stable patches from Upstream
- DPAA2 support for DPDK
- Various performance optimization and bug fixes
- Features supported:
  - Fragmentation and Re-assembly
  - RX Queue Tail drop for performance improvement at high traffic rates
  - DPAA2 now supports Flow Control (Pause Frame)

- DPAA1 now supports Multicast address filtering

Virtualization - OVS-DPDK

- DPAA2 support enabled in OVS-DPDK
- DPDK working in Virtual Machine on DPAA2 and DPAA1

Open Data Plane (ODP)

- Supported on LS2088A
- Supports ODP API v1.11
- Following Applications are supported:
  - ODP generator sample application
  - ODP pktio sample application
  - ODP ipsec transport and tunnel sample applications
  - ODP packet classify sample application
  - ODP timer sample application
  - ODP LPM IP Forwarding sample application
  - ODP Traffic Manager sample application
  - ODP OpenFastPath Applications (FPM & FPM_BURSTMODE)

U-Boot Boot Loader

- LS2088A: DUART, DDR4, I2C, PCIe, SATA, USB, SD, MMC, NOR, NAND flash, Networking support, Boot from NOR flash
- NAND secure boot on LS1043A
- SD secure boot on LS1043A and LS1046A

Other Tools and Utilities

- LS2088A: Primary Protected Application (PPA) firmware
- MC 10.1.2 updates
- Restool: DPNI configuration support for Order Restoration, Debug Configration, Loopback Option
- Benchmark: power management benchmark on LS1046A

What's New in SDK V2.0-1701

Highlights

- Kernel 4.1.35 upgrade
- Integrates off-train releases for LS1012A BSP v0.4
- DPDK on LS1043A, LS1046A and LS2080A
- Real-time support recovery on B4860, LS1012A and P4080
- PSCI and sleep 32-bit kernel on LS1012A, LS1043A and LS1046A
- Includes several software fixes. See Fixed, Open, and Closed Issues on page 64 section which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-010284, A-010150, A-008975

See list of changes in below

Processor and Board Support

- LS1012A r1.0 and RDB, Freedom
- LS1046A r1.0: upgrading platform and fman frequency to 700MHz and 800MHz, adding 0.9v part support, adding core/platform/fman 1.2GHz/400MHz/600MHz
- LS1021A default core frequency 1.2GHz upgrade

Linux Kernel Core, Virtualization

- Kernel 4.1.35 upgrade
- QEMU 2.6 upgrade

Linux Kernel Drivers

- LS1012A: Crypto driver supporting SEC 5 (CAAM), DDR, DUART, DSPI, eSDHC, I2C, PCIe RC, PFE Ethernet (Packet Rx/Tx), PHY support: RGMII & SGMII, Power management, QSPI, SAI/I2S, SATA, UART, USB 2/3 mass storage, Watchdog
- LS1046A: Thermal monitor
- PSCI and sleep 32-bit kernel on LS1012A, LS1043A and LS1046A
- Real-time support
- User space IO

Data Plane Development Kit (DPDK)

- DPDK v16.07 API Support
- Following DPDK Applications have been verified
  - l2fwd
  - l3fwd
  - l2fwd_crypto
  - ipsecgateway

Virtualization - OVS-DPDK

- OVS-DPDK working with vhost-virtio interfaces
- DPDK working in Virtual Machine

U-Boot Boot Loader

- Enable CONFIG_PARTITION_UUIDS, CONFIG_EFI_PARTITION, CONFIG_CMD_GPT
- LS1012A: Non-secure boot, Secure Boot, Clock, CPLD, DDR4, DSPI, eSDHC, I2C, Generic Timers, PCIe, Primary Protected Application (PPA) firmware integration, QSPI, SATA, UART, USB
- Loading 32-bit kernel for ARMv8 with PSCI and PPA enabled

Other Tools and Utilities

- LS1012A: Primary Protected Application (PPA) firmware

What's New in SDK V2.0-1611

NOTE

The list of highlights below are incremental changes relative to functionality available in previous releases

Highlights

- U-Boot 2016.09 upgrade
- Integrates off-train releases for LS1046A r1.0 and LS1043A r1.1

- 32-bit kernel on LS1043A and LS1046A (Note: no 32-bit kernel on LS2080, no 32-bit USDPAA)
- Includes several software fixes. See Fixed, Open, and Closed Issues on page 64 section which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-010539, A-007273, A-008975

See list of changes in below

Processor and Board Support

- LS1046A and LS1046A-RDB, adding core frequency 1.8GHz support
- LS1043A r1.1
- LS1021A default core frequency 1.2GHz upgrade

Linux Kernel Core, Virtualization

- ARM A72 (AARCH64), Little Endian (default)
- ARM A72 32-bit effective kernel addressing
- ARM A72 64-bit effective addressing

Linux Kernel Drivers

- LS1046A: CEETM, Crypto via SEC 5, DSPI, DPAA Networking, eDMA, eSDHC, Flextimer, GIC-400, I2C, IEEE1588, IFC NOR & NAND, MDIO, PCIe RC & Endpoint, PCIe MSI, Power Management: CPU Idle, Device Frequency Scaling (DFS), CPU Hotplug (LPM20), Thermal Monitor (TMU), QDMA, SATA, SMMU, USB 2.0 & 3.0, UART, Watchdog timer
- LS1043A: CPU Hotplug (LPM20), PCIe MSI

User Space Datapath Acceleration Architecture (USDPAA) and Reference Applications

- LS1046A: Base and DPAA drivers, USDPAA Applications: Hello Reflector, PPAC Reflector, RC IPFWD, LPM IPFWD, IPSecFwd, Simple Proto, Simple Crypto

U-Boot Boot Loader

- U-Boot 2016.09
- LS1046A: Non-secure boot, Secure Boot, Clock, CPLD, DDR4, DSPI, eSDHC, FMan IM, GIC-400, I2C, IFC NOR & NAND, MDIO, OCRAM, PCIe, Primary Protected Application (PPA) firmware integration, QSPI, SATA, UART, USB

Other Tools and Utilities

- Primary Protected Application (PPA) firmware [LS1046A, LS1043A] - Hotplug (LPM20), System Reset

What's New in SDK V2.0-1609

Highlights

- Updated to Linux kernel 4.1.30 which includes several fixes from the community
- Includes several software fixes. See Fixed, Open, and Closed Issues on page 64 section which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-009942, A-007728, A-010240, A-007728, A-008822

See list of changes in SDK V2.0 below

What's New in SDK V2.0

Highlights

- Linux 4.1 Long-Term Support (LTS) kernel upgrade
- Yocto 2.0 (jethro) upgrade
- U-Boot 2016.01 upgrade

- Integrates off-train releases for LS1043A and LS2080A
- Continued support for QorIQ ARM and Power Architecture processors in a common source base
- Support for QorIQ LS1012A SoC (32-bit and 64-bit kernel)

See full list of features and changes below:

Processor and Board Support

- LS1043A and LS1043A-RDB
- LS2080A and LS2085A-RDB
- P1010, P1021/2/3/4/5, P2020/10, and BSC9131/2 no longer supported. Please see prior SDK releases and support in upstream repositories

Yocto and Toolchain

- Yocto/Poky 2.0 "jethro"
- power: gcc-4.9.2, glibc-2.20, binutils-2.25, gdb-7.10.1
- arm: gcc-linaro-4.9.3-r2015.03, glibc-linaro-2.20--r2014.11, binutils-linaro-2.25-r2015.01, gdb-7.10.1

Linux Kernel Core, Virtualization

- Linux kernel 4.1.8
- ARMv8: AARCH64, 64-bit effective addressing, Little Endian (default), Multicore SMP, Huge Pages (hugetlbfs), Kernel-based Virtual Machine (KVM), Libvirt, Linux Containers (LXC), Docker Engine

Linux Kernel Drivers

- LS1043A: CEETM, Crypto via SEC 5, DSPI, DPAA Networking, eDMA, eSDHC, Flextimer, GIC-400, I2C, IEEE1588, IFC NOR & NAND, MDIO, PCIE, Power Management: CPU Idle, Device Frequency Scaling (DFS), CPU Hotplug (PW15), Thermal Monitor (TMU), QDMA, QE TDM, SATA, SMMU, USB 2.0 & 3.0, UART, Watchdog timer
- LS2080A: Crypto via SEC6, DCE, Datapath I/O services (DPIO), DPAA2: MC Bus, Ethernet, Edge Virtual Bridge (EVB), DSPI, DUART, GIC-500, I2C, IFC NOR & NAND, PCIE, Power Management: Device Frequency Scaling (DFS), Thermal Monitor (TMU), PHYs: RGMII, SGMII, XFI, Quad-SPI, Real-Time Clock, SATA, SMMU, USB, VFIO, Watchdog timer
- IEEE1588 PTPd support on LS1021A

OpenDataPlane

- v16.05 release (available separately)
- Conforms to ODP 1.4
- Open source APIs for networking data plane
- ODP application support including pktio, l2/l3forwarding, ipsec, classifier, timer

User Space Datapath Acceleration Architecture (USDPAA) and Reference Applications

- LS1043A: Base and DPAA drivers, USDPAA Applications: Hello Reflector, PPAC Reflector, RC IPFWD, LPM IPFWD, IPSecFwd, Simple Proto, Simple Crypto
- CAPWAP no longer supported

U-Boot Boot Loader

- NOTE: FLASH MAP CHANGE FOR SDK 1.6 AND LATER
- U-Boot 2016.01
- ARMv8 core
- LS1043A: Non-secure boot, Secure Boot, Clock, CPLD, DDR3L, DDR4, DSPI, eSDHC, FMan IM, GIC-400, I2C, IFC NOR & NAND, MDIO, OCRAM, PCIe, Primary Protected Application (PPA) firmware integration, SATA, UART

- LS2080A: LS2040A personality, Non-secure boot, Secure Boot, DUART, DDR3, DPAA2 networking, DSPI, GIC-500, I2C, IFC NOR & NAND, Boot from NOR, MC boot, MDIO, PCIE, SATA, USB 2 & 3
- Voltage ID on T4240RDB

Other Tools and Utilities

- Primary Protected Application (PPA) firmware [LS1043A] - Hotplug (PW15), System Reset
- Management Complex (MC) Firmware version 9.0.5 – binary only, supporting DPAA2 resource containers and network objects, Resource Manager and Link Manager
- DPAA2 resource container and object management tool (RESTOOL)
- Convenience scripts to create and manage common objects like network interfaces. There scripts are packaged in ls2-scripts tarball.

For a list of Fixed, Open and Closed Issues, see “Fixed, Open, and Closed Issues on page 64”.

2.3 Components

Top-level components in the QorIQ SDK

Overall

- Yocto
- GNU Toolchain
- Linux Kernel and Virtualization
- Linux Kernel Drivers
- Application Specific Fastpath (ASF)
- Data Plane Development Kit (DPDK)
- Virtulization - OVS-DPDK
- Open Data Plan (ODP)
- U-Boot Boot Loader
- User Space Datapath Acceleration Architecture (USDPAA) and Applications
- Embedded Hypervisor (Topaz)
- Other Tools and Utilities

Yocto

- Yocto/Poky 2.0 "jethro"
- Open source collaboration project that provides templates, tools and methods for building custom Linux-based systems for embedded products
- 32-bit or 64-bit user space

GNU Toolchain

- power: gcc-4.9.2, glibc-2.20, binutils-2.25, gdb-7.10.1
- arm: gcc-linaro-4.9.3-r2015.03, glibc-linaro-2.20--r2014.11, binutils-linaro-2.25-r2015.01, gdb-7.10.1
- Based on sources from Free Software Foundation
- Includes additional updates from NXP

Linux Kernel Core and Virtualization

- Linux kernel 4.1.35
- ARM A7 (AARCH32), A53, A57 and A72 (AARCH64), Little Endian (default)
- Power Architecture e500mc, e5500, e6500
- Multicore SMP support and multithread (e6500)
- 32-bit effective kernel addressing [e500mc, e5500, A57, A72]
- 64-bit effective addressing [e6500, A53, A57, A72]
- Huge Pages (hugetlbfs)
- Kernel-based Virtual Machine (KVM)
- Libvirt 1.2.19
- Linux Containers (LXC) 1.1.4 function support
- QEMU 2.6

Linux Kernel Drivers

- Customer Edge Egress Traffic Management (CEETM)
- Crypto driver via SEC 3, 4, 5 & 6 (CAAM)
- Display Control Unit (DCU) and HDMI [LS1021A]
- Display Interface Unit (DIU) [T1042 and P1022]
- DPAA Offloading Driver [T4240, T2080, P4080, P2041, B4860, B4420]
- DUART, eSPI, I2C
- Edge Virtual Bridge (EVB) [DPAA2 processors]
- Ethernet DPAA [DPAA1 processors]
- Ethernet DPAA2 [DPAA2 processors]
- Ethernet eTSEC (gianfar) [LS1012A, C29x]
- Frame Manager (FMan) [DPAA1 processors]
- GIC-400, GIC-500 [ARM processors]
- IEEE1588
- Integrated Flash Controller (IFC) NOR and NAND flash
- Local Bus Controller (eLBC) NOR and NAND flash
- LPUART [LS1021A, LS1043A], LS1046A]
- Management Complex Bus [DPAA2 processors]
- MDIO
- Multiprocessor Interrupt Controller (MPIC)
- PCIe Root Complex and Endpoint, MSI
- PFE Ethernet (Packet Rx/Tx) [LS1012A]
- Platform DMA
- Pattern Matching Engine (PME)
- Peripheral Access Management Unit (PAMU) [Power Architecture]
- PHY support: RGMII, SGMII, XFI and XAUI

- Power Management (PM) – CPU hotplug (PH20), CPU idle (PW15/20), Sleep (LPM20), Deep sleep (LPM35), Auto-Response, Dynamic Frequency Scaling (DFS), Thermal Monitor, Power Monitor (board specific)
- Power Monitor using on-board sensors [T4240QDS, P5020DS, P1022DS]; direct access or through FPGA-OCM
- Queue Manager (QMan) and Buffer Manager (BMan) [DPAA1 processors]
- QUICC Engine UART, Ethernet, TDM
- RAID Engine & support for hardware-assist [P5020]
- Real-Time Linux [B4860, T4240, LS1021A, LS2088A]
- SATA
- Secured Digital Host Controller (eSDHC) and SD/MMC support [all except for TF on T2080]
- Serial RapidIO (SRIO)
- System Memory Management Unit (SMMU) [ARM processors]
- Universal Serial Bus (USB) 2.0 and 3.0 [LS1021A, LS1043A, LS1046A, LS2088A]
- User space IO
- Virtual Function I/O (VFIO) - PCIe pass-through [Power Architecture Processors]
- Virtual Function I/O (VFIO) - mmap PCI sources [LS1043A, LS1046A, LS2088A]
- Watchdog Timers

Application Specific Fastpath (ASF)

- Loadable kernel module based fastpath solution
- IP Forward, IPSec Forward, QoS
- IPv4 and IPv6
- 32-bit and 64-bit effect addressing
- Supported platforms are LS1021A, T1040D4RDB, P4080

Data Plane Development Kit (DPDK) [LS1043A, LS1046A, LS2088A]

- DPDK v16.07 API Support
- Following DPDK Applications have been verified
  - l2fwd
  - l3fwd
  - l2fwd_crypto
  - ipsecgateway

Virtualization - OVS-DPDK

- OVS-DPDK working with vhost-virtio interfaces
- DPDK working in Virtual Machine

Open Data Plane (ODP) [LS2088A]

- ODP API v1.11
- ODP generator sample application
- ODP pktio sample application
- ODP ipsec transport and tunnel sample applications
- ODP packet classify sample application

- ODP timer sample application
- ODP LPM IP Forwarding sample application
- ODP Traffic Manager sample application
- ODP OpenFastPath Applications (FPM & FPM_BURSTMODE)

User Space Data Path Acceleration Architecture (USDPAA)

- Device-tree handling
- QMan and BMan drivers and C API
- DMA Memory Management
- Network Configuration
- CPU Isolation
- Drivers - BMan, QMan, RMan, SRIO, PME, Offloading

USDPAA Reference Applications

- PPAC Reflector
- Hello Reflector
- IP Forward (route cache)
- IP Forward (longest prefix match)
- IPSec Forward
- PME Loopback
- RMan application
- RMU (RapidIO Message Unit) application
- Simple Crypto
- Simple Proto
- SRIO Applicaton
- DPAA Offload: Classifier [LS1043A], IPSec Offload Fragmentation, Reassembly [T4240, T2080, B4860, P4080, P2041]

U-Boot Boot Loader

- **NOTE: FLASH MAP CHANGE FOR SDK 1.6 AND LATER (on Power Architecture processors)**
- U-Boot: 2016.09
- On ARM platforms, the U-Boot image includes the device tree
- Non-secure and Secure Boot (ESBC)
- Primary Protected Application (PPA) firmware integration. See PPA features in “Other Tools …” below
- Clock, CPLD, DUART, DDR3, DDR4, DSPI, eSDHC, GIC-400, GIC-500, I2C, OCRAM, PCIe, USB 2 & 3, SATA, UART
- Networking support using eTSEC, FMAN Independent Mode or DPAA2 networking, PFE based Ethernet support
- DCU, eMMC 4.5, I2C3, LPUART, QSPI [LS1021A, LS1043A, LS1046A]
- eLBC and IFC access to NOR and NAND flash
- Boot from NOR, NAND flash, eSPI, SDHC
- Boot from SRIO [B4860, P5020, P4080]
- Voltage ID (board specific)
- CodeWarrior debug patch for U-Boot

Embedded Hypervisor (Topaz)

- Partitioning of CPUs, memory and I/O devices
- Protection and isolation of resources from other partitions
- Sharing of hardware resources between partitions
- Virtualization of devices
- [all Power Architecture processors]

Other Tools and Utilities

- Primary Protected Application (PPA) firmware [LS1012A, LS1043A, LS1046A, LS2088A]
- Management Complex (MC) Firmware version 10.1.2 – binary only, supporting DPAA2 resource containers and network objects, Resource Manager and Link Manager
- Manager, Link Manager, DPDMUX basic configurations
- DPAA2 resource container and object management tool (RESTOOL)
- Convenience scripts to create and manage common objects like network interfaces. There scripts are packaged in ls2-scripts tarball
- FLIB/RTA - SEC descriptor creation library [all processors with SEC 4 or 5]
- OpenSSL 1.0.2h - includes heartbleed patch
- OpenSSL offload - includes TLS Record Layer and Public Key offload
- DSP Boot [B4860/4420]
- Frame Manager Configuration Tool (FMC) [DPAA1 processors]
- Frame Manager Ucode [DPAA1 processors]
- PME Tools [DPAA1 processors]
- Inter-process Communication (IPC) between Power and DSP cores [B4860/4420]
- L1 Defense [B4860/4420]
- L2 Switch User Space Driver and Demo Application [T1040]

2.4 Supported Targets

Processors, development boards, and cards supported all releases.

NOTE

In the following tables, in the rows corresponding to the processors, the silicon revision is indicated. In the rows corresponding to the development boards, the board is marked with a checkmark if it is supported. "N" means that a processor or development board is not supported.

Table 1. QorIQ Value-Performance Processors Supported

Processor	Board	SDK 1.2	SDK 1.3	SDK 1.3.2	SDK 1.4	SDK 1.5	SDK 1.6	SDK 1.7	SDK 1.8	SDK 1.9	SDK 2.0	SDK 2.0-1611	SDK 2.0-1701	SDK 2.0-1703

Table continues on the next page...

Table 1. QorIQ Value-Performance Processors Supported (continued)

LS1012A		N	N	N	N	N	N	N	N	N	N	N	rev 1.0	rev 1.0
	LS1012ARDB	N	N	N	N	N	N	N	N	N	N	N	✓	✓
	FRDM-LS1012A	N	N	N	N	N	N	N	N	N	N	N	✓	✓
LS1021A/ LS1020A		N	N	N	N	N	N	rev 1.0	rev 1.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0
	TWR-LS1021A	N	N	N	N	N	N	✓	✓	✓	✓	✓	✓	✓
LS1043A		N	N	N	N	N	N	N	N	N	rev 1.0	rev 1.0	rev 1.0	rev 1.0
	LS1043ARDB	N	N	N	N	N	N	N	N	N	✓	✓	✓	✓
LS1046A		N	N	N	N	N	N	N	N	N	N	rev 1.0	rev 1.0	rev 1.0
	LS1046ARDB	N	N	N	N	N	N	N	N	N	N	✓	✓	✓
P1010/P1014		rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 2.0	rev 2.01	rev 2.01	rev 2.01	rev 2.01	N	N	N	N
	P1010RDB	✓	✓	✓	✓	N	N	N	N	N	N	N	N	N
	P1010RDB-PB	N	N	N	N	✓	✓	✓	✓	✓	N	N	N	N
P1020/P1011		rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	N	N	N	N
	P1020RDB-PC	✓	✓	✓	N	N	N	N	N	N	N	N	N	N
	P1020RDB-PD	N	N	N	✓	✓	✓	✓	✓	✓	N	N	N	N
P1021/ P1012		rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	N	N	N	N
	P1021RDB-PC	✓	✓	✓	✓	✓	✓	✓	✓	✓	N	N	N	N
P1022/ P1013		rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	N	N	N	N
	P1022DS-PA	✓	✓	✓	N	N	N	N	N	N	N	N	N	N
	P1022DS-PB	N	N	N	✓	✓	✓	✓	✓	✓	N	N	N	N

Table continues on the next page...

Table 1. QorIQ Value-Performance Processors Supported (continued)

P1023/ P1017		rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	N	N	N	N
	P1023RDB-PA	N	N	N	✓	✓	✓	✓	✓	✓	N	N	N	N
	P1023RDS	✓	✓	✓	N	N	N	N	N	N	N	N	N	N
P1024/ P1015		rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	N	N	N	N
	P1024RDB-PC	✓	✓	✓	N	N	N	N	N	N	N	N	N	N
P1025/ P1016		rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	N	N	N	N
	P1025RDB-PC	✓	✓	✓	N	N	N	N	N	N	N	N	N	N
	TWR-P1025	N	N	N	✓	✓	✓	✓	✓	✓	N	N	N	N
T1023/T1013		N	N	N	N	N	N	N	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0
	T1023RDB-PC	N	N	N	N	N	N	N	✓	✓	✓	✓	✓	✓
T1024/T1014		N	N	N	N	N	N	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0
	T1024RDB-PA	N	N	N	N	N	N	✓	N	N	N	N	N	N
	T1024RDB-PC	N	N	N	N	N	N	N	✓	✓	✓	✓	✓	✓
T1040/ T1020		N	N	N	N	N	rev 1.0	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1
	T1040RDB-PB	N	N	N	N	N	✓	✓	N	N	N	N	N	N
	T1040D4RDB-PA	N	N	N	N	N	N	N	✓	✓	✓	✓	✓	✓
T1042/ T1022		N	N	N	N	N	rev 1.0	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1
	T1042D4RDB-PA	N	N	N	N	N	N	N	✓	✓	✓	✓	✓	✓

Table 2. QorIQ Mid-Performance Processors Supported

Processor	Board	SDK 1.2	SDK 1.3	SDK 1.3.2	SDK 1.4	SDK 1.5	SDK 1.6	SDK 1.7	SDK 1.8	SDK 1.9	SDK 2.0	SDK 2.0-1611	SDK 2.0-1701	SDK 2.0-1703

Table continues on the next page...

Table 2. QorIQ Mid-Performance Processors Supported (continued)

LS2080A		N	N	N	N	N	N	N	N	N	rev 1.0	rev 1.0	rev 1.0	N
	LS2085ARDB	N	N	N	N	N	N	N	N	N	✓	✓	✓	N
LS2088A		N	N	N	N	N	N	N	N	N	N	N	N	rev 1.0
	LS2088ARDB	N	N	N	N	N	N	N	N	N	N	N	N	✓
P2020		rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	N	N	N	N
	P2020RDB-PCA	✓	✓	✓	✓	✓	✓	✓	✓	✓	N	N	N	N
P2040/ P2041		rev 1.0	rev 1.0 rev 1.1	rev 1.1 rev 2.0	rev 1.1 rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0
	P2041RDB-PA	✓	✓	N	N	N	N	N	N	N	N	N	N	N
	P2041RDB-PB	N	✓	✓	✓	N	N	N	N	N	N	N	N	N
	P2041RDB-PC	N	N	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P3041		rev 1.0 rev 1.1	rev 1.1	rev 1.1 rev 2.0	rev 1.1 rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0
	P3041DS-PA	✓	✓	✓	✓	N	N	N	N	N	N	N	N	N
	P3041DS-PC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T2080		N	N	N	N	N	rev 1.0	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1
	T2080QDS	N	N	N	N	N	✓	✓	✓	✓	✓	✓	✓	✓
	T2080RDB-PB	N	N	N	N	N	✓	N	N	N	N	N	N	N
	T2080RDB-PC	N	N	N	N	N	N	✓	✓	✓	✓	✓	✓	✓

Table 3. QorIQ High-Performance Processors Supported

Processor	Board	SDK 1.2	SDK 1.3	SDK 1.3.2	SDK 1.4	SDK 1.5	SDK 1.6	SDK 1.7	SDK 1.8	SDK 1.9	SDK 2.0	SDK 2.0-1611	SDK 2.0-1701	SDK 2.0-1703
P4080/ T4040		rev 2.0	rev 2.0 rev 3.0	rev 2.0 rev 3.0	rev 2.0 rev 3.0	rev 3.0	rev 3.0	rev 3.0	rev 3.0	rev 3.0	rev 3.0	rev 3.0	rev 3.0	rev 3.0
	P4080DS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P5020/ P5010		rev 1.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0
	P5020DS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P5040/ P5021		rev 1.0	rev 1.0	rev 2.0	rev 2.0	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1	rev 2.1
	P5040DS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T4240/ T4080/ T4160		N	N	rev 1.0	rev 1.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0
	T4240QDS	N	N	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	T4240RDB-PB	N	N	N	N	N	✓	✓	✓ [1]	✓	✓	✓	✓	✓

Table 4. QorIQ C29x Family of Crypto Coprocessors Supported

Processor	Board	SDK 1.2	SDK 1.3	SDK 1.3.2	SDK 1.4	SDK 1.5	SDK 1.6	SDK 1.7	SDK 1.8	SDK 1.9	SDK 2.0	SDK 2.0-1611	SDK 2.0-1701	SDK 2.0-1703
C29x		N	N	N	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0
	C29xPCIE	N	N	N	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 5. QorIQ Qonverge Processors Supported

Processor	Board	SDK 1.2	SDK 1.3	SDK 1.3.2	SDK 1.4	SDK 1.5	SDK 1.6	SDK 1.7	SDK 1.8	SDK 1.9	SDK 2.0	SDK 2.0-1611	SDK 2.0-1701	SDK 2.0-1703
BSC9131		N	N	N	rev 1.0	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	N	N	N	N
	BSC9131RDB	N	N	N	✓	✓	✓	✓	✓	✓	N	N	N	N

Table continues on the next page...

[1] Starting with SDK 1.8, T4240RDB utilizes 1.8GHz core frequency and upgraded DDR DIMMs

Table 5. QorIQ Qonverge Processors Supported (continued)

BSC9132		N	N	N	rev 1.0	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	N	N	N	N
	BSC9132QDS	N	N	N	✓	✓	✓	✓	✓	✓	N	N	N	N
B4420		N	N	rev 1.0	rev 1.0	rev 1.0 rev 2.1	rev 2.2	rev 2.2	rev 2.2	rev 2.2	rev 2.2	rev 2.2	rev 2.2	rev 2.2
	B4420QDS	N	N	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
B4860/ B4460		N	N	rev 1.0	rev 1.0	rev 1.0 rev 2.1	rev 2.2	rev 2.2	rev 2.2	rev 2.2	rev 2.2	rev 2.2	rev 2.2	rev 2.2
	B4860QDS	N	N	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 6. QorIQ Riser Cards Supported

Riser Card	SDK 1.2	SDK 1.3	SDK 1.3.2	SDK 1.4	SDK 1.5	SDK 1.6	SDK 1.7	SDK 1.8	SDK 1.9	SDK 2.0	SDK 2.0-1611	SDK 2.0-1701	SDK 2.0-1703
SGMII-RISER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
XAUI-RISER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
XAUI-RISER-B	N	N	N	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
XFI	N	N	N	N	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 7. PowerQUICC III Processors Supported

Processor	Board	SDK 1.2	SDK 1.3	SDK 1.3.2	SDK 1.4	SDK 1.5	SDK 1.6	SDK 1.7	SDK 1.8	SDK 1.9	SDK 2.0	SDK 2.0-1611	SDK 2.0-1701	SDK 2.0-1703
MPC8536		rev 1.2	rev 1.2	rev 1.2	N	N	N	N	N	N	N	N	N	N
	MPC8536DS	✓	✓	✓	N	N	N	N	N	N	N	N	N	N
MPC8544		rev 2.1	rev 2.1	rev 2.1	N	N	N	N	N	N	N	N	N	N
	MPC8544DS	✓	✓	✓	N	N	N	N	N	N	N	N	N	N

Table continues on the next page...

Table 7. PowerQUICC III Processors Supported (continued)

MPC8548		rev 3.1	rev 3.1	rev 3.1	N	N	N	N	N	N	N	N	N	N
	MPC8548DS	✓	✓	✓	N	N	N	N	N	N	N	N	N	N
MPC8572		rev 2.2.1	rev 2.2.1	rev 2.2.1	N	N	N	N	N	N	N	N	N	N
	MPC8572DS	✓	✓	✓	N	N	N	N	N	N	N	N	N	N

2.5 Feature Support

Features supported by each processor.

Table Legend:

"Y" - Feature is supported by software

"—" - Feature is not supported by software

"na" - Hardware feature is not available

Table 8. Key Features

Feature	QorIQ Processing Platforms																	
	LS1012A	**LS1021A**	**LS1043A**	**LS1046A**	**LS2088A**	**T1024**	**T1040**	**T1042**	**T2080**	**T4240**	**C29x**	**B4860**	**B4420**	**P2041**	**P3041**	**P4080**	**P5020**	**P5040**
32-bit Userspace	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
64-bit Userspace	Y	na	Y	Y	Y	Y	Y	Y	Y	Y	na	Y	Y	na	na	na	Y	Y
36b phys mem	—	Y	na	na	na	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
40b phys mem	Y	na	Y	Y	Y	Y	Y	Y	Y	Y	na	na	na	na	na	na	na	na
AIOP[2]	na	na	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na
ASF Applications	—	Y	—	—	—	—	Y	Y	—	—	na	—	—	—	—	Y	—	—
Data Plane Development Kit (DPDK)	—	—	Y	Y	Y	—	—	—	—	—	—	—	—	—	—	—	—	—
Huge Pages (tlbfs)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y

Table continues on the next page...

[2] The current SDK release does not support any SoC's that contain AIOP. Forthcoming SoC's such as LS2088A and LS1088A will support AIOP.

Table 8. Key Features (continued)

OpenDataPlane (ODP)	—	—	—	—	Y	—	—	—	—	—	—	—	—	—	—	—	—	—
Real-Time Linux	—	Y	—	—	—	—	—	—	—	—	—	Y	Y	—	—	Y	—	—
Multithreading - Hardware thread support	na	na	na	na	na	na	na	na	Y	Y	na	Y	Y	na	na	na	na	na
Secure Boot	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y
USDPAA Applications	—	na	Y	Y	na	Y	Y	Y	Y	Y	na	Y	Y	Y	Y	Y	Y	Y

Table 9. Virtualization

Feature	**QorIQ Processing Platforms**																	
	LS1012A	**LS1021A**	**LS1043A**	**LS1046A**	**LS2088A**	**T1024**	**T1040**	**T1042**	**T2080**	**T4240**	**C29x**	**B4860**	**B4420**	**P2041**	**P3041**	**P4080**	**P5020**	**P5040**
KVM	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y
LXC	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y
libvirt	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y
Topaz	na	na	na	na	na	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y

Table 10. Linux Applications

Feature	**QorIQ Processing Platforms**																	
	LS1012A	**LS1021A**	**LS1043A**	**LS1046A**	**LS2088A**	**T1024**	**T1040**	**T1042**	**T2080**	**T4240**	**C29x**	**B4860**	**B4420**	**P2041**	**P3041**	**P4080**	**P5020**	**P5040**
Linux IPFwd	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y
Linux IPSec	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y
Linux Termination	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y
Linux NAS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	na	—	—	Y	Y	Y	Y	Y
Linux RAID	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	na	—	—	Y	Y	Y	Y	Y
Linux SATA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	na	—	—	Y	Y	Y	Y	Y

Table 11. Linux Kernel Drivers

Feature	QorIQ Processing Platforms																	
	LS1012A	LS1021A	LS1043A	LS1046A	LS2088A	T1024	T1040	T1042	T2080	T4240	C29x	B4860	B4420	P2041	P3041	P4080	P5020	P5040
Audio - SAI	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
DCE	na	na	na	na	Y	na	na	na	Y	Y	na	na	na	na	na	na	na	na
DPAA	na	na	Y	Y	na	Y	Y	Y	Y	Y	na	Y	Y	Y	Y	Y	Y	Y
DPAA2	na	na	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na
eSDHC	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	na	na	Y	Y	Y	Y	Y
FlexCAN	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
I2C	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
IEEE1588, PTPd	na	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
IFC	na	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	na	na	na	na	na
L2 Switch	na	na	na	na	na	na	Y	na	na	na	na	na	na	na	na	na	na	na
LPUART	na	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na
PAMU	na	na	na	na	Y	Y	Y	Y	Y	Y	na	Y	Y	Y	Y	Y	Y	Y
PCIe RC	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
PCIe EP	—	—	na	Y	Y	—	—	—	Y	Y	Y	—	—	—	—	Y	—	—
PME	na	na	na	na	Y	na	Y	Y	Y	Y	na	Y	Y	Y	Y	Y	Y	na
Power Management	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y
RAID HW assist	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	Y	Y
RMAN	na	na	na	na	na	na	na	na	Y	Y	na	Y	Y	na	na	na	na	na
SATA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	na	na	na	Y	Y	na	Y	Y
SEC	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
SPI	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
SRIO	na	na	na	na	na	na	na	na	Y	Y	na	Y	Y	Y	Y	Y	Y	na

Table continues on the next page...

Table 11. Linux Kernel Drivers (continued)

TDM (QE)	na	na	Y	na	na	Y	Y	Y	na	na	na	na	na	na	na	na	na	na
USB	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	na	Y	Y	Y	Y	Y	Y	Y
veTSEC/eTSEC	na	Y	na	na	na	na	na	na	na	na	Y	na	na	na	na	na	na	na
Video - DIU	na	na	na	na	na	Y	na	Y	na	na	na	na	na	na	na	na	na	na
Video - DCU	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Watchdog	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	—	Y	Y	Y	Y	Y	Y	Y

2.6 Acronyms and Abbreviations

Term	Definition
AH	Authentication Header (RFC 4302) – a network protocol designed to provide authentication services in IPv4 and IPv6.
AMP	Asynchronous Multi-Processing, running multiple operating system images on different processors of the same machine without virtualization.
API	Application Programming Interface
ARP	Address Resolution Protocol
ASF	Application Specific Fast Path – a software abstraction layer providing different network services (e.g. firewall, IPSec, IP reassembly, forwarding, etc.) to top level applications implemented in various environments (such as Linux kernel, Linux user space, etc.). These services may be hardware accelerated or not depending on the hardware the ASF is running on. The hardware offloading is provided transparently to the top level applications.
BMan	Buffer Manager – a DPAA hardware block performing buffer and buffer pool management.
BPID	(BMan) Buffer Pool ID
CAAM	Cryptographic Acceleration and Assurance Module
CCSR	Configuration and Control Status Register
CEETM	Customer Edge Egress Traffic Management
CPU	Central Processing Unit, also known more generally as "Processor"
DCD	Device Configuration Data
DMA	Direct Memory Access

Table continues on the next page...

Table continued from the previous page...

Term	Definition
DPAA	Data Path Acceleration Architecture
DPAA2	Data Path Acceleration Architecture (Second Generation)
DSK	Device Secret Key
DTB	Device Tree Blob—the binary representation of device trees
DTS	Device Tree Syntax—the textual representation of device trees
ePAPR	Embedded Power Architecture Platform Requirements
ESBC	External Secure Boot Code
ESP	Encapsulating Security Payload (RFC 4303) – a network protocol designed to provide a mix of security services in IPv4 and IPv6.
FDB	Forwarding Data Base
FMan	Frame Manager – a DPAA hardware block performing frame processing management.
FMC	Frame Manager Configuration application
FRA	Freescale RMan Application
FUID	Freescale Unique ID
HAL	Hardware Abstraction Library
IBR	Internal Boot ROM
IPC	Inter-Process Communication, can be interpreted as being communication between distinct application execution flows or between distinct hardware processing units.
inbound (traffic)	Encrypted traffic which is coming from an unprotected interface. This traffic will be terminated on the CPU.
IPFwd	IPv4 Forward
IPSec	IP Security – a communication standard defined and refined by several public RFCs (such as RFC-2401 and RFC-4301) where hosts exchange encrypted IP data packets.
IPSec Tunnel	A communication convention between two network gateways to IPSec process certain network traffic in a particular way. An IPSec tunnel has two endpoints (which are the gateways), a clearly delimited set of encryption and authentication methods, keys, encapsulation headers and security policies, which define the traffic that is sent through the tunnel.
ISBC	Internal Secure Boot Code
ISR	Interrupt Status Register

Table continues on the next page...

Table continued from the previous page...

Term	Definition
ITF	Intent to Fail
ITS	Intent to Secure
KVM	Kernel-based Virtual Machine - A Linux kernel module that allows a user space program access to the hardware virtualization features of NXP processors.
LIODN	Logical I/O Device Number
OEM	Original Equipment Manufacturer
oNIC	Offload Network Interface Controller - a virtual Ethernet driver, similar to Macless.
OS	Operating System
OUID	OEM Unique ID
outbound (traffic)	Clear traffic which is coming from a software application which generates traffic that must be encrypted and forwarded via an unprotected interface.
PAMU	Peripheral Access Management Unit
PBL	Pre-Boot Loader
PCD	Parse, Classify, Distribute – a software architecture concept in NXP DPAA drivers which allows the user to configure the DPAA hardware (FMan) to do frame parsing, classification or distribution on a series of frame queues.
PDCP	Packet Data Convergence Protocol – It is one of the layers of the Radio Traffic Stack in UMTS/LTE and performs IP header compression and decompression, transfer of user data and maintenance of sequence numbers for Radio Bearers which are configured for lossless serving radio network subsystem (SRNS) relocation.
PME	Pattern Matcher Engine
PPAM	Packet Processing Application Module
QEMU	Quick EMUlator - A hosted hypervisor that performs hardware virtualization.
QMan	Queue Manager – a DPAA hardware block performing frame queue management.
RC	Route Cache
RFC	Request for Comments – a public document which describes a software standard.
RCW	Reset Configuration Word - hardware boot-time parameters for the P4080
RMan	RapidIO Message Manager – a DPAA hardware block performing message passing for inter-processor and inter-devices communication.

Table continues on the next page...

Table continued from the previous page...

Term	Definition
SA	Security Association – a data record, defined by RFC 4301, which stores the information related to the IPSec processing needed for a specific network traffic type (such as encryption/decryption keys and algorithms, traffic endpoints description, authentication algorithms, and so on).
SAD	Security Association Database – the database holding all the valid SAs in a system.
SDK	Software Development Kit
SEC	Security Engine Coprocessor – a DPAA hardware block performing cryptographic acceleration and offloading hardware.
SFP	Secure Fuse Processor
SKMM	Secure Key Management Module
SMP	Symmetric Multi-Processing, running an operating system image on multiple CPUs simultaneously.
SNVS	Secure Non-Volatile Storage
SoC	System on a Chip, a chip integrating one or more processors and on-chip peripherals.
SP	Security Policy – a set of rules that network traffic must comply with in order to be eligible for IPSec processing.
SPD	Security Policy Database – the database storing all the SPs in a system.
SRE	Stateful Rule Engine
SRK	Super Root Key
SRKH	Super Root Key Hash
SUI	String Under Inspection
TLB	Translation Lookaside Buffer
TTL	Time To Live
UID	Unique Device ID
UIO	User space I/O
USDPAA	User Space Data Path Acceleration Architecture
VID	Voltage IDentifier

2.7 Fixed, Open, and Closed Issues

Known issues for this and previous releases of the QorIQ SDK

This section contains 3 tables: Fixed, Open and Closed issues. Fixed issues have a software fix that has been integrated into the 'Fixed In' Release. Open issues do not currently have a resolution. Workaround suggestions are provided where possible. Closed issues are issues where the root cause and fix are outside the scope of the QorIQ SDK.

Table 12. SDK V2.0-1703 Fixed Issues

ID	Description	Disposition	Opened In	Fixed in
QUBOOT-2338	The data read from the address of 32M in QSPI flash is incorrect on LS1012AFDRM and LS1012ARDB.	Fixed	SDK v2.0-1701	SDK v2.0-1703
QLINUX-6048	Out of memory (OOM) might be reported at high traffic rates, when running IPSec in tunnel mode with 3DES-CBC-HMAC-SHA1 cipher. The SEC cryptographic engine will be oversubscribed, backlog will increase, buffers will not be released, and in the end the system memory will be depleted. Affected SoCs: P2041, P3041, P4080, P5020.	Fixed	SDK v1.9	SDK v2.0-1703
QLINUX-5996	The behaviour ("Error enqueuing frame" with EBUSY error code) is expected when there is oversubscription. It results in traffic getting dropped due to congestion. It does not affect the result of performance tests like RFC2544.	Fixed	SDK v2.0-1609	SDK v2.0-1703
QSDK-2420	When testing Linux_IEEE1588_ptpd_unicast, the synchronization got more than 300000. This is the bug of PTPd 2.3.1-rc2 version.	Fixed	SDK v1.9	SDK v2.0-1703
QLINUX-6130	Realtime support is not available in SDK 2.0-1611 as 4.1.30 is not the official version supported by Realtime kernel.	Fixed	SDK v2.0-1611	SDK v2.0-1701
QUBOOT-1998	The system cannotenter sleep whenenabling Qman.	Fixed	LS1046A BSP v0.4	SDK v2.0-1611
QLINUX-5988	There is kernel calltrace while doingbridgeperformance testover PCIe and 10Gports in MSI mode	Fixed	LS1046A BSP v0.4	SDK v2.0-1611
QLINUX-5898	PCIe end pointmode can't bedetected with x86PC host.	Fixed	LS1046A BSP v0.2	SDK v2.0-1611
QSDK-3006	C293 - SKMM host driver - Host kernel recompile IOMMU disable dependency removed.	Fixed	SDK v2.0	SDK v2.0-1609

Table continues on the next page...

Table 12. SDK V2.0-1703 Fixed Issues (continued)

QSDK-3005	C29x - fix instability issue observed while loading PKC firmware T4240 based systems	Fixed	SDK v2.0	SDK v2.0-1609
QLINUX-5794	The SDK2.0 is based on the vanilla kernel release v4.1.8. The networking stack in this kernel version has an issue that results in RCU stalls in certain conditions. Two patches addressing the RCU stall issue were backported from kernel version v4.6.	Fixed	SDK v2.0	SDK v2.0-1609
QLINUX-5764	CAAM - JR Driver - fix for public key offload reported post SDK 2.0	Fixed	SDK v2.0	SDK v2.0-1609
QUBOOT-1640	On LS1043A RDB board, when running Secure Boot, U-boot must be executed with a valid PPA binary and CSF Header of PPA loaded on NOR Flash.	Fixed	SDK v2.0	SDK v2.0-1609
QLINUX-5416	DPAA2 Ethernet cannot be compiled as a module.	Fixed	LS2085A EAR6	SDK v2.0
QLINUX-5333	dpaa2-evb: port state is not updated as expected.	Fixed	LS2085A EAR6	SDK v2.0
QUBOOT-1522	The USB controller will work with specific clock settings (multiple of 100 MHz), so 533MHz is believed to render USB non-functional. Usb controller derives its clock from the system bus clock itself (PFA). When SoC operates at 533 Mhz sys clock, disable the USB.	Fixed	LS2085A EAR6	SDK v2.0
QLINUX-4941	The "cat" output of MC console displays a junk line as first line	Fixed	LS2085A EAR6	SDK v2.0
QLINUX-4624	Messages like "Error enqueuing frame: -16" may be observed when running IPsec performance tests. The behaviour ("error enqueuing frame" with EBUSY error code) is expected when there is oversubscription. It results in traffic getting dropped due to congestion. It does not affect the result of performance tests like RFC2544.	Fixed	SDK v2.0	SDK v2.0-1609
QSDK-2235	C293 - PKC host driver - Host kernel recompile IOMMU disable dependency removed.	Fixed	SDK v2.0	SDK_V2-0_1609
QSDK-1793	On LS1021A, some USB3.0 flash drives do not work with USB xHCI controller. These devices suddenly stop responding to commands from xHCI controller (there is no bus/system error).	Fixed	SDK v1.8	SDK_V2-0_1609

Table 13. SDK V2.0-1703 Open Issues

ID	Description	Disposition	Opened In	Workarounds
QUBOOT-20333	For the PCIe card with multiple switch, it fails when needing more stream IDs to support mutilple switch.	Open	LS2088A BSP v0.3	Increase the stream IDs in u-boot code.
QLINUX-69611	In 32-bit kernel, jumboframe floodping causes call trace on LS1012A RDB and LS1012A FRDM board.	Open	LS1012A BSP v0.5	
QLINUX-67119	KVM VFIO test fails and call trace occurs on guest on T1023 and T1024.	Open	SDK v2.0-1701	
QLINUX-65995	Sleep causes the hardware interrupt timeout issue for SD CMD18 on LS1046ARDB.	Open	SDK v2.0-1701	

Table continues on the next page...

Table 13. SDK V2.0-1703 Open Issues (continued)

DPDK-3033	There is "IPSEC: Unsupported packet type" when running dpdk_ipsec_secgw_armce-4c4g test on LS1043ARDB.	Open	SDK v2.0-1701	
QLINUX-6507	In the current release the IPSec performance tests will be affected by the missing congestion group support from MC firmware. The DPAA2_CAAM Driver has been updated to support congestion group management, ut in order to validate the IPSec peformance, a new MC binary with DPSECI version >= 5.1 will have to be deployed on the board. The new versions MC binaries will be available on https://github.com/qoriq-open-source/mc-binary	Open	LS2088A BSP v0.3	
QLINUX-61700	IFC NAND on LS2088A RDB does not work in Linux.	Open	LS2088A BSP v0.3	
QLINUX-60644	PPFE on LS1012A doesn't support real-time feature.	Open	LS1012A BSP v0.4	

Table continues on the next page...

Table 13. SDK V2.0-1703 Open Issues (continued)

QLINUX-58833	The PPFE port does not work after power management being entered and exited repeatedly.	Open	LS1012A BSP v0.4	
QLINUX-58411	Linux IPfwd for small frame size has more degradation compared with LS1012A BSP v0.3 on LS1012ARDB.	Open	LS1012A BSP v0.3	
QLINUX-57768	rmmod of PFE modules crashes the kernel on LS1012A.	Open	LS1012A BSP v0.2	
QLINUX-57738	USB HDD (Oyen Digital) cannot be detected in Linux properly on LS1012AFRDM and LS1012ARDB	Open	LS1012A BSP v0.1	

Table continues on the next page...

Table 13. SDK V2.0-1703 Open Issues (continued)

QLINUX-6089	Traffic stops after running two netperf sessions (TCP_STREAM and TCP_MAERTS) per interface/core on LS1043A	Open	SDK v2.0-1611	
QLINUX-6082	Macless devices are not supported (neither in the Linux kernel nor in usdpaa) on LS1043A and LS1046A.	Open	SDK v2.0	
QLINUX-6049	On T1023/T1024/P5040 platforms KVM USB pass-through feature causes a call trace in guest.	Open	SDK 2.0-1611	
QLINUX-5981	The USB do not work properly after system resumes from deep sleep. Enabling USB with deep sleep leads to unidentified behavior of kernel.	Open	SDK v2.0-1609	Disable USB when using deep sleep

Table continues on the next page...

Table 13. SDK V2.0-1703 Open Issues (continued)

QSDK-28889	The vulnerability is indexed as CVE-2015-7547. Multiple stack-based buffer overflows in the (1) send_dg and (2) send_vc functions in the libresolv library in the GNU C Library (aka glibc or libc6) before 2.23 allow remote attackers to cause a denial of service (crash) or possibly execute arbitrary code via a crafted DNS response that triggers a call to the getaddrinfo function with the AF_UNSPEC or AF_INET6 address family, related to performing "dual A/AAAA DNS queries" and the libnss_dns.so.2 NSS module.	Open	SDK v2.0	The fix is available on https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=d5a4840c6b4025302f485b9271e4c72d315221f5
QLINUX-56016	On LS1043A, KVM support on host machines with 64KB pages is not functional. Limitation exists because the memory range associated with the GIC CPU interface, in the GIC400 memory map, is not aligned to 64KB.	Open	SDK v2.0	Use only host machines with 4KB pages in order to support KVM virtualization.
QLINUX-56022	Please make sure to enable "CONFIG_MTD_CFI_BE_BYTE_SWAP" for LS1043A. It is required because IFC is big-endian in LS1043A. SDK2.0 by-default does not enable CONFIG_MTD_CFI_BE_BYTE_SWAP for ARMV8 defconfig to support LS2080A which has IFC as little endian.	Open	SDK v2.0	

Table 14. SDK V2.0-1701 Closed Issues

ID	Description	Disposition	Found In	Workarounds
DPDK-327	In DPDK test, OVS setup fails on LS1043ARDB.	Not an Issue in the latest release	SDK v2.0-1701	

Table continues on the next page...

Table 14. SDK V2.0-1701 Closed Issues (continued)

QLINUX-6529	For ARMv8 platforms LS1012ARDB, LS1043ARDB and LS1046ARDB compiled in ARMv7 compatibility mode, the ARMv8 Crypto functionalities are not available.	Won't Fix	SDK v2.0-1701	
QUSDPA-878	USDPAA IPSec High Bandwidth is not available on LS1046.	Won't Fix	SDK v2.0-1611	
QUBOOT-2161	The LS1021ATWR board and CPLD could not provide clock frequency information to the software. So the u-boot software can only support static frequency settings (100M DDR clock and 100M system clock). If different clock settings are selected through the switch configurations. User needs to change the settings in software manually(include/ configs/ls1021atwr.h). Change the following settings to the frequency actually selected by the switches. #define CONFIG_SYS_CLK_FREQ 100000000 #define CONFIG_DDR_CLK_FREQ 100000000	Won't Fix	DK v2.0-1701	
QUBOOT-2055	The parameter of fdt_high set to 0xffffffff causes the failure of booting images from nor flash directly.	Won't Fix	SDK v2.0-1611	Set fdt_high in uboot environment variable to 0xa0000000 by using the command in uboot: =>setenv fdt_high 0xa0000000
ODP-417	Per CoS, pool configuration is not supported.	Won't Fix	ODPv16.08	
QUBOOT-1737	If Code Warrior is used to program u-boot or RCW on QSPI Flash, the flash gets into invalid state.	Won't Fix		Use Code-Warrior to program the 64 MB SDK image (LS1012) thereby programming the entire flash. After power recycle the board will start functioning.

Table continues on the next page...

Table 14. SDK V2.0-1701 Closed Issues (continued)

QLINUX-5717	USB 3.0 sticks (Kingston 3.0 16 gb and HP 3.0 8 GB) are not getting detected in Linux on LS2080RDB board. 1)When devices are connected through USB2.0 cable, then those are getting detected in high speed mode. [expected] When CATC is in path (without CATC device is not getting detected) : 2)USB3.0 :: When connected through CATC with Sniffing disabled, devices are getting detected as USB2.0 devices. [wrong behavior :: expected speed is USB3.0] 3)When connected through CATC but with CATC sniffing enabled, devices are getting connected as USB3.0 devices. (experiment is same as in item-2 but capturing traces via CATC)	Hardware Issue	NA	
QLINUX-5661/ QUBOOT-1320	HP 2.0 pen drive not enumerated in Standard A port on LS2080ARDB and LS1043ARDB board. It is concluded as USB connector (Standard A port) issue. However, it is properly enumerated in micro port.	Hardware Issue	NA	
QLINUX-5671	On LS1021A TWR board, when doing deep sleep with all three Ethernet ports on, there may be the error message "PM: Device mdio@2d24000:02 failed to suspend: error -16".	Hardware Issue	NA	Upgrade the on-board CPLD to version 3.2.
QLINUX-5637	On LS1021A TWR board, after resuming from deep sleep, the kerenl can't initialize SD card and occurs call trace because the card never leaves busy state.	Hardware Issue	NA	Upgrade the on-board CPLD to version 3.2.
QLINUX-5504	Performance measurements in different IPv6 forwarding and IPv6 SEC forwarding scenarios revealed that a small performance degradation was introduced by the kernel upgrade from version 3.12 to 4.1. The actual amount of this reduction in performance varies with the use case, some showing a certain depreciation, others no degradation. Tests performed in isolation showed no or insignificant performance impact for the changes made in NXP owned components.	Upstream Issue	SDK v2.0	
QLINUX-5417	Cortina PHY LEDs are permanently off.	Hardware Issue	NA	
QLINUX-5325	AQR PHY LED remains off if link is at 1Gbps.	Hardware Issue	NA	

Table continues on the next page...

Table 14. SDK V2.0-1701 Closed Issues (continued)

QSDK-2478	On LS1021A, boot dtb, kernel and filesystem directly from QSPI flash could not be supported.	Won't Fix	SDK v1.9	1.Program the general dtb, Linux kernel and ramdisk to the QSPI flash by 'sf write' command(under sdboot or qspiboot). 2.Boot the dtb, kernel and ramdisk from QSPI flash. Avoid booting from QSPI flash directly. Read the dtb, kernel and ramdisk from the QSPI flash to RAM by 'sf read' u-boot command and then boot from RAM. By default, the QSPI flash on the TWR-LS1021A includes: rcw, uboot, kernel, dtb, and ramdisk. Note: This workaround only applies to QSPI flash.
QUSDPA-797	On LS1043A RDB board, USDPAA IPFwd performance does not scale up from 2 cores to 4 cores.	Hardware Issue	NA	
QLINUX-3645	When using T4240QDS or RDB, PCL10 and CPU frequency scaling may not always work reliably.	Hardware Issue	NA	Please check with your sales representative for a board replacement.

Table continues on the next page...

Table 14. SDK V2.0-1701 Closed Issues (continued)

QUSDPA-749	On e6500 processors, Linux application performance may be impacted due to the upgrade of eglibc from 2.15 to 2.19 as compared to SDK 1.7. These changes were introduced by the upstream community. Some of these changes address a security concern discovered in eglibc 2.15. In the SDK, this can be observed when executing USDPAA IPFwd LPM and comparing results against SDK 1.7.	Hardware Issue	NA	None. Recommend to use eglibc 2.19 or later to avoid the security concern in previous versions.
QLINUX-3357	With TWR-LS1021A, some resolutions (e.g. 1920x1080) may not work well with some monitors. The software will not downgrade to another resolution automatically.	Hardware Issue	NA	Manually set another resolution such as: 1024x768@60 : fbset -fb /dev/fb0 -g 1024 768 1024 768 24 -t 15384 168 8 29 3 144 6
QSDK-1841	With TWR-LS1021A, copy from NOR flash to NOR flash fails. It is a known limitation with Micron flash.	Hardware Issue	NA	
QSDK-1716	The read performance using PCIe SATA 3132 card on T4240QDS/T4240RDB/T2081QDS is about 93MB/s which is lower than the read performance (~133MB/s) on other platforms. Using LS2 PCIe card (LSI SAS 9211-8i) can get the similar read performance between different platforms.	Won't Fix	SDK v1.6	Use GEN2 PCIe card (LSI SAS 9211-8i) to get better read performance.
QSDK-1677	When telneting to board console from Linux server connected to on LS1021A TWR, the board will power reset due to wrong signal sent.	Hardware Issue	NA	Remove R214 from the board.
QSDK-1616	To bring up UP (uni-processor), only disabling the SMP in kernel configuration will result in the warning information.	Functions as designed	SDK v1.7	Delete the node for another processor and change the reg of the left one to 00.

Table continues on the next page...

Table 14. SDK V2.0-1701 Closed Issues (continued)

QLINUX-2924	Linux IPv6 network stack exhibits exponential degradation for large number of cores, this affects performance for IPv6 forwarding, which degrades from 16 to 24 cores. Degradation is caused by atomic operations being used in IPv6 routing code from upstream kernel: # Overhead Command Shared Object Symbol # with 16 cores 17.26% ksoftirqd/1 [kernel.kallsyms] [k] .atomic_dec_return_noinline 13.45% ksoftirqd/1 [kernel.kallsyms] [k] .atomic_inc_noinline 5.53% ksoftirqd/1 [kernel.kallsyms] [k] .ip6_pol_route # with 24 cores 32.45% ksoftirqd/1 [kernel.kallsyms] [k] .atomic_dec_return_noinline 30.56% ksoftirqd/1 [kernel.kallsyms] [k] .atomic_inc_noinline 4.71% ksoftirqd/1 [kernel.kallsyms] [k] .ip6_pol_route	Upstream Issue	SDK v1.6	One workaround is to use a PCD policy that distributes IPv6 traffic only on 16 cores e.g. <distribution name="ipv6eth14" > <queue count="16" base="0x7c00" / > <key > <fieldref name="ipv6.src"/ > <fieldref name="ipv6.dst"/ > <fieldref name="ipv6.tos"/ > </key > </distribution > This distribution can be added to a standard IPv4 PCD policy that distributes IPv4 traffic on core-affine FQs <policy name="linux_fman_tester_policy_14" > <dist_order > <distributionref name="ipv6eth14"/ > <distributionref name="udpeth14"/ >

Table continues on the next page...

Table 14. SDK V2.0-1701 Closed Issues (continued)

				<distributionref name="tcpeth14"/ > <distributionref name="ipv4eth14"/ > <distributionref name="garbage_dist_14"/ > </dist_order > </policy > If you want to use the pool channel for IPv4 traffic, as if no policy was applied, it will not work. The FMan will drop all traffic that is not matched by the policy. You can read the RX default FQID of the interface from sysfs and create a distribution that sends all IPv4 traffic on the default RX FQ. root@t4240rdb:~# cat /sys/devices/fsl,dpaa.19/ethernet.30/net/fm2-mac9/fqids Rx error: 32782 Rx default: 32783 # which means in hexa 0x800f Rx PCD: 31744 - 31871

Table continues on the next page...

Table 14. SDK V2.0-1701 Closed Issues (continued)

				Edit the policy file: <distribution name="ipv4eth14" > <queue count="1" base="0x800f"/ > </distribution > <policy name="linux_fman_tester_policy_14" > <dist_order > <distributionref name="ipv6eth14"/ > <distributionref name="ipv4eth14"/ > </dist_order > </policy >
QUBOOT-799	When using the T4240QDS, the system may hang after reset of the board. This is due to an erratum for the eMMC device (MTFC4GGQDQ) in which it fails to reset.	Resolved	QUBOOT_2014_SDK_V1-6	Cycle power to the board.
QUBOOT-749	A compatibility issue has been found on some PCIe Gen 1 cards when used with a PCIe Gen2 controller. A common symptom is that U-Boot hangs during PCIe initialization with such a card inserted.	Closed	QUBOOT_T1040_V0-3	Use PCI Gen2 cards. Or set the following RCW: RCW[SRDS_DIV_PEX]=2 for T1024 and T1040 RCW[SRDS_DIV_PEX_S1]=2 and RCW[SRDS_DIV_PEX_S2]=1 for T2080

Table continues on the next page...

Table 14. SDK V2.0-1701 Closed Issues (continued)

QLINUX-2504	On e6500 core, if hardware threads are disabled, such as by the use of CONFIG_PPC_DISABLE_THREADS, tracebacks will appear roughly two minutes after boot, indicating that certain threads have hung. This is due to Linux creating per-cpu software threads for the hardware threads that are disabled. There should be no harmful effect of this beyond the error output.	Won't Fix	SDK v1.5	Enable hardware threads, or ignore the error output.
QLINUX-2240	When running 8C20G Linux IPv4 forward using two XFI ports on B4860QDS, 'non-zero error counters in fman statistics' is found	Hardware Issue	NA	
QLINUX-1816	On B4860QDS and B4420QDS, a hang may be observed when performing the reboot command at the kernel prompt. The system won't be able to reach the U-Boot prompt. This is when RCW is at NOR Flash. This is a issue of NOR flash(S29GL01GS11TFI010) and It is reported to "Spansion".	Hardware Issue	NA	Options: 1) Re-configure the board to read the RCW from I2C EEPROM. 2) Boot from NAND Flash such that the RCW, U-Boot, etc. are in NAND flash.
QSDK-610	On B4860QDS and B4420QDS even though the NOR Flash size is 128M, only 64M can be accessed due to a board issue. Standard CFI commands will return the NOR Flash size to be 128M; however, only 64M can be accessed. The address range for NOR Flash is limited to 0xec000000 to 0xffffffff instead of 0xe8000000 to 0xffffffff. When attempting to read the NOR Flash at addresses 0xe8000000 to 0xebffffff, the data shown corresponds to 0xec000000 to 0xffffffff. As a result only 4 vbanks are supported. Please refer to the SDK documentation for more details of the NOR Flash memory map.	Hardware Issue	NA	None

Chapter 3
Getting Started

Yocto Project is an open-source collaboration project for embedded Linux developers. Yocto Project uses the Poky build system to make Linux images. For complete information about Yocto Project, see https://www.yoctoproject.org/.

3.1 SDK File System Images

This section describes the file system images that can be built using standard Yocto Project recipes included with the NXP SDK.

The file system images contain the programs, scripts, and other files that make up Linux user space. There are five standard images. They are described in the following sections.

In the SDK installation directory, look in `meta-freescale/recipes-fsl/images` for files that define these images.

Where to start: "`fsl-image-full`" contains a rich set of standard Linux features and all special NXP SDK-specific features. It is the best starting point for exploration and evaluation.

3.1.1 fsl-image-minimal: A Barebones Starting Point for Products

Contents:

This is a barebones image. It contains a small file set that allows Linux to boot and little else.

Purpose:

This image is intended as a starting point for product development. Users may add packages to it to form an image that targets their particular project or purpose. Packages may be added by editing conf/local.conf and adding new packages to be built and installed via

```
IMAGE_INSTALL_append = " package1 package2 etc"
```

Then, rebuild the image using bitbake. The result will be an image that is small enough for simple flash devices and is narrowly focused on a specific goal.

Users must add packages to "fsl-image-minimal" to make it useful. Thus it is not intended for "out-of-the-box" evaluation. Instead, use it as the basis for targeted images for your specific product.

3.1.2 fsl-image-mfgtool: A Small Flash Image for Managing Disks and Larger Images

Contents:

This is a small image that NXP preprograms into the flash on development boards. On many boards, the image is stored in a NOR flash and is loaded into a RAM disk when Linux boots. It contains disk management functions as described in the next section.

Purpose:

This image is intended to help users to load much larger images onto disks or disk-like devices such as SDHC cards or USB thumb drives. Thus, this image contains networking support to transfer images and also standard Linux disk and file system manipulation commands.

NXP preloads "fsl-image-full" (see below) onto disks on development boards that have them. For these boards, "fsl-image-mfgtool" can be used to restore the larger disk image if it becomes corrupted.

Users will find that it is often convenient to work with larger images and may wish to install “fsl-image-full” onto a disk-type device and use it with NXP development boards that do not come with a disk drive.

The networking and disk management commands that NXP supplies are standard Linux commands. They are, in fact, identical on all architectures and thus are not unique to NXP. Users with Linux experience will find them familiar. They include:

- ifconfig, ip, route, etc. (configure networking)
- ftp (transfer files via the network)
- scp (another way to transfer files via the network)
- date and rdate (set date and time)
- fdisk (partition disks and disk-type devices)
- mkfs (make file systems)
- tar (extract file system images, and more)
- fsck (check file systems)
- mount/umount (mount and umount file systems)

3.1.3 fsl-image-full: A Full-Featured Image, Useful Out-of-Box

Contents:

This is a large image that contains many standard Linux commands and features including native (target-resident) versions of the GNU tools including gcc and gdb.

If you boot this image, the resulting Linux environment will be much like a command-line on a full desktop-type Linux system rather than an embedded system.

Purpose:

While this type of image may not be appropriate for a final embedded product, it can be very helpful for many development and evaluation tasks. The reason is the full set of standard Linux facilities that are already present in the image. In fact, users may find that they can use this image instead of installing the Yocto Project-based NXP SDK onto a development system, at least initially.

For this reason, “fsl-image-full” is preinstalled on the disk drives of NXP development boards that have disks.

For example, the image is complete enough that the standard Linux open source command sequence “configure; make” stands a decent chance of working for arbitrary open source packages that do not happen to be on the image already.

To be clear, the NXP SDK is a Yocto Project/Poky-derived embedded distribution. However, the Yocto Project standard Linux package set is large enough that if one enables a lot of the available packages, the result begins to have the feel of desk top Linux. NXP added the special NXP-specific packages, and the result is “fsl-image-full.” It is intended for “out-of-the-box” evaluation because it is rather complete.

3.1.4 fsl-image-core: A Small Image with NXP-Specific Packages Present

Contents:

This is a small image somewhat like “fsl-image-minimal” except it contains all of the NXP-specific SDK packages.

Purpose:

This image is useful for evaluating the NXP-specific software packages in the context of a file system image that is much more embedded-oriented than “fsl-image-full”.

Embedded file system images contain fewer helpful tools and utilities by definition. They are intended to support an embedded product's functionality rather than a developer's tasks. Thus, it can be convenient to begin with "fsl-image-core" for evaluation and planning and then later narrowly extend "fsl-image-minimal" to support your embedded product.

3.1.5 fsl-image-virt: An image for KVM deployment

Contents:

This is an image which contains the specific packages needed to enable virtualization.

Purpose:

This image is useful for virtualization scenarios (KVM, libvirt, lxc). It contains:

- the guest root filesystem
- the guest image (uImage format for Power based architectures and zImage for ARM based architectures)
- QEMU
- all necessary libraries and tools for libvirt and lxc support

3.2 Essential Build Instructions

The following sections are essential to the build process and must be performed when using Yocto Project to build the SDK. In order to install the SDK, download the necessary ISO's and follow steps for installing SDK 2.0, install SDK v2.0-1703. When those steps are completed, prepare the host environment, setup Poky, perform builds, and follow the instructions in the subsequent sections. When these steps are completed, the build process will be complete. Linux images that have been built will be found in the following directory: `build_<machine>/tmp/deploy/images/<machine>`

See Additional Instructions for Developers for more information on using Yocto Project.

3.2.1 Install SDK V2.0-1703

SDK-V2.0-1703 is an incremental update to SDK 2.0. Prior to installing SDK-V2.0-1703, install SDK 2.0. Both SDK 2.0 and SDK V2.0-1703 can be downloaded from www.nxp.com/sdk. SDK 2.0 documentation and installation instructions can be found on Install SDK 2.0 on page 81 section.

SDK-V2.0-1703 includes all the changes of previous incremental/NPI releases, such as SDK-V2.0-1701/SDK-V2.0-1611/SDK-V2.0-1609/LS1046A-SDK-V0.4. They can't be installed at the same time. The install script will detect conflict releases and prompt the steps to deal with them.

How to install SDK-V2.0-1703 on a host machine.

1. Extract the SDK-V2.0-1703.tar.bz2, and run the 'install' script to install the incremental updates:

```
$ tar -xjf SDK-V2.0-1703.tar.bz2
$ ./SDK-V2.0-1703/install
```

2. During the install process, the user will be prompted to input the SDK 2.0 ISO installed location, i.e., `<ISO_INSTALL_PATH>/QorIQ-SDK-V2.0-20160527-yocto.`

3.2.2 Install SDK 2.0

How to install Yocto Project on the host machine.

1. Mount the ISO on your machine:

```
$ sudo mount -o loop QorIQ-SDK-<version>-<target>-<yyyymmdd>-yocto.iso /mnt/cdrom
```

2. As a non-root user, install Yocto Project:

```
$ /mnt/cdrom/install
```

3. When you are prompted to input the install path, ensure that the current user has the correct permission for the install path.

There is no uninstall script. To uninstall Yocto Project, you can remove the <yocto_install_path>/QorIQ-SDK-<version>-<yyyymmdd>-yocto directory manually.

NOTE

* The *source* ISO contains the package source tarballs and Yocto Project recipes. It can be installed and used to do non-cache build.

* The *cache* ISO contains the pre-built cache binaries. To avoid a long time build, you can install the source ISO and the cache ISO in the same installation folder.

* The *image* ISO includes all prebuilt images: flash images, standalone toolchain installer, HD rootfs images and small images.

* The source ISO can be used separately. The cache ISO and the source ISO should work together.

3.2.3 Set Up Host Environment

Yocto Project requires some packages to be installed on the host.

Use the following steps to prepare the Yocto Project environment.

In general, Yocto Project can work on most recent Linux distributions with Python-2.7.3 or greater (excluding python3 which is not supported), git-1.7.8 or greater, tar-1.24 or greater and required packages installed. The default Python is not 2.7.x on some Linux distros, e.g. CentOS 6.5 installs python 2.6.6. Follow the instructions below to install the Python 2.7.x in custom path instead of override the system default python, the override may cause system utilities breaking.

```
$ wget https://www.python.org/ftp/python/2.7.6/Python-2.7.6.tar.xz
[NOTE: Python 2.7.3 and python 2.7.5 can be used as well.]
$ tar -xf Python-2.7.6.tar.xz
$ cd Python-2.7.6
$ ./configure --prefix=/opt/python-2.7.6
$ make
$ sudo make install
```

Run the export command below to ensure python 2.7.x is used for Yocto build.

```
$ export PATH=/opt/python-2.7.6/bin:$PATH
```

Yocto Project supports typical Linux distributions: Ubuntu, Fedora, CentOS, Debian, OpenSUSE, etc. More Linux distributions are continually being verified. This SDK has been verified on following Linux distributions: Ubuntu 14.04, CentOS-7.1.1503, Debian 8.2, Fedora 22 and OpenSUSE 13.2

For a list of the Linux distributions tested by the Yocto Project community see `SANITY_TESTED_DISTROS` in `poky/meta-yocto/conf/distro/poky.conf`.

To ensrure Yocto can run on host machine with different Linux distribution, the following packages should be installled on the host.

For CentOS hosts:

```
$ sudo yum install gawk make wget tar bzip2 gzip python unzip perl patch \
     diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath socat SDL-devel xterm
```

For the Fedora hosts:

```
$ sudo yum install gawk make wget tar bzip2 gzip python unzip perl patch \
     diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath \
     ccache perl-Data-Dumper perl-Text-ParseWords perl-Thread-Queue socat \
     findutils which SDL-devel xterm
```

For Ubuntu and Debian hosts:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
     build-essential chrpath socat libsdl1.2-dev xterm
```

Extra packages are needed for Ubuntu-64b:

```
$ sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0 ia32-libs lib32ncurses5-dev
```

For OpenSUSE host:

```
$ sudo zypper install python gcc gcc-c++ libtool subversion git chrpath automake make wget
diffstat makeinfo freeglut-devel libSDL-devel
```

3.2.4 Set Up Poky

Source the following poky script to set up your environment for your particular NXP platform. This script needs to be run once for each terminal, before you begin building source code. Make sure that you are currently in the <install_path> when running the commands below.

```
$ . ./fsl-setup-env -m <machine>
```

For example:

```
$ . ./fsl-setup-env -m b4860qds-64b
```

The following shows the usage text for the `fsl-setup-env` command:

Usage:

```
$ . ./fsl-setup-env -h
Usage: . fsl-setup-env -m <machine>

    Supported machines
    QorIQ: b4420qds-64b b4420qds b4860qds-64b b4860qds c293pcie ls1021atwr ls1043ardb p2041rdb
p3041ds p4080ds p5020ds-64b p5020ds p5040ds-64b p5040ds t1023rdb-64b t1023rdb t1024rdb-64b t1024rdb
t1040d4rdb-64b t1040d4rdb t1042d4rdb-64b t1042d4rdb t2080qds-64b t2080qds t2080rdb-64b t2080rdb
t4160qds-64b t4160qds t4240qds-64b t4240qds t4240rdb-64b t4240rdb ls1012afrdm-32b ls1012afrdm
ls1012ardb-32b ls1012ardb ls1043ardb-32b ls1046ardb-32b ls1046ardb ls2088ardb

  Op  tional parameters:
    * [-m machine]: the target machine to be built.
    * [-b path]:    non-default path of project build folder.
    * [-j jobs]:    number of jobs for make to spawn during the compilation stage.
    * [-t tasks]:   number of BitBake tasks that can be issued in parallel.
    * [-d path]:    non-default path of DL_DIR (downloaded source)
    * [-c path]:    non-default path of SSTATE_DIR (shared state Cache)
    * [-g]:         enable Carrier Grade Linux
    * [-l]:         lite mode. To help conserve disk space, deletes the building
```

```
                    directory once the package is built.
    * [-h]:         help
```

3.2.5 Perform Builds

How to Set Up a Cross Compile Environment and Perform Builds

Follow these steps to do builds using Yocto Project. Be sure to set up the host environment before doing these steps.

1. `$ cd <sdk-install-dir>/build_<machine>`
2. `$ bitbake <image-target>`

   Where `<image-target>` is one of the following:

   - `fsl-image-minimal`: contains basic packages to boot up a board
   - `fsl-image-core`: contains common open source packages and NXP specific packages.
   - `fsl-image-full`: contains all packages in the full package list.
   - `fsl-image-kernelitb`: A FIT image comprising the Linux image, dtb and rootfs image.
   - `fsl-image-mfgtool`: contains all the user space apps needed to deploy the fsl-image-mfgtool image to a USB stick, hard drive, or other large physical media.
   - `fsl-image-virt`: contains toolkit to interact with the virtualization capabilities of Linux
   - `core-image-x11`: NXP image with a very basic X11 image with a terminal
   - `fsl-toolchain`: the cross compiler binary package
   - `package-name(usdpaa)`: build a specific package

Contents of the Built Images Directory:

A Yocto Project build produces images that will be located in the following directory:

```
<sdk-install-dir>/build_<machine>/tmp/deploy/images/<machine>/
```

The following list shows the typical directory/image files (exact contents depend on the setting of the `<IMAGE_FSTYPES>` variable):

- `fsl-image-<machine>.ext2.gz.u-boot` - ramdisk image that can be loaded with U-Boot
- `fsl-image-<machine>.ext2.gz` - gzipped ramdisk image
- `fsl-image-<machine>.tar.gz` - gzipped tar archive of the image
- `uImage-<machine>.bin` - kernel binary of the image
- `u-boot-<machine>.bin` - U-Boot binary image that can be programmed into board Flash
- `uImage-<machine>.dtb` - device tree binary (dtb).
- `fsl_fman_ucode_<machine>_<version>.bin` - fman ucode for <machine> board
- `hv/hv.uImage` - uImage for hypervisor
- `hv-cfg/*/*/hv.dtb` - dtb for hypervisor
- `rcw/*/rcw_*.bin` - rcw

NOTE

For additional Yocto Project usage information, please refer to the https://www.yoctoproject.org/.

3.3 Additional Instructions for Developers

This section describes additional "How To" instructions for getting started with Yocto Project.

Each set of instructions is aimed towards developers that are interested in modifying and configuring the source beyond the default build. Each section will describe instructions on how to use Yocto Project to achieve a specific development task.

3.3.1 Customize U-Boot

How to Configure, Modify, or Rebuild U-Boot

To Modify U-Boot Configuration:

Modify `UBOOT_CONFIG`. Values for `UBOOT_CONFIG` are listed in `<sdk-install-dir>/sources/meta-freescale/conf/machine/<machine>.conf`

e.g. UBOOT_CONFIG ??= "nor"

To Modify U-Boot Source Code:

If source code has already been installed, please skip steps 1 & 2 and proceed modifying source code.

1. `$ bitbake -c cleansstate u-boot`

   NOTE

   Other helpful bitbake cleaning commands:

   `bitbake -c clean <target>`

   - Removes work directory in `build_<machine>/tmp/work`

   `bitbake -c cleansstate <target>`

   - Removes work directory in `build_<machine>/tmp/work`
   - Removes cache files in `<sdk-install-dir>/sstate-cache/` directory.

2. `$ bitbake -c patch u-boot`
3. `$ cd <S>` and modify the source code. Follow instructions to "Rebuild U-Boot Image" when you are finished modifying source code.

   NOTE

   Use `bitbake -e <package-name> | grep ^S=` to get value of <S>, the package source code directory.

To Rebuild U-Boot Image:

1. `$ cd build_<machine>`
2. `$ bitbake -c compile -f u-boot`
3. `$ bitbake u-boot`

   NOTE

   U-Boot image can be found in `build_<machine>/tmp/deploy/images/<machine>/`

3.3.2 Customize Linux Kernel

How to Configure, Modify, or Rebuild the Linux Kernel

To Modify Kernel Source Code:

If source code has already been installed, please skip steps 1 & 2 and proceed modifying source code.

1. `$ bitbake -c cleansstate virtual/kernel`
2. `$ bitbake -c patch virtual/kernel`
3. `$ cd <S>` and change the source code. Follow instructions to "Rebuild Kernel Image" when you are finished modifying source code.

NOTE

Use `bitbake -e <package-name> | grep ^S=` get the value of `<S>` (package source code directory).

To Change the Kernel defconfig:

1. Update `KERNEL_DEFCONFIG` variable in <sdk-install-dir>/sources/meta-freescale/conf/machine/<machine>.conf

To Change dts:

1. Update `KERNEL_DEVICETREE` variable in <sdk-install-dir>/sources/meta-freescale/conf/machine/<machine>.conf

To Do menuconfig:

1. `$ bitbake -c menuconfig virtual/kernel`

NOTE

If you are going to reuse this new kernel configuration for future builds, tell menuconfig to "Save Configuration to Alternate File" and give it an absolute path of `/tmp/my-defconfig`. If you do not do this, the new `defconfig` file will be removed when doing a `clean/cleansstate` for kernel.

NOTE

This runs the normal kernel `menuconfig` within the Yocto Project environment. If the kernel configure UI cannot open, edit `/<yocto_install_path>/build_<machine>/conf/local.conf` and add the following based on your environment (prior to issuing the bitbake command).

For a non-X11 environment:

- `OE_TERMINAL = "screen"`

The following commands can be used for the other environments:

For a GNOME environment (default):

- `OE_TERMINAL = "gnome"`

For a KDE environment:

- `OE_TERMINAL = "konsole"`

For non-GNOME and non-KDE environments:

- `OE_TERMINAL = "auto"`

To Rebuild Kernel Image:

1. `$ cd build_<machine>`
2. `$ bitbake -c compile -f virtual/kernel`
3. `$ bitbake virtual/kernel`
4. `$ bitbake fsl-image-kernelitb`

NOTE

Kernel images can be found in `<sdk-install-dir>/build_<machine>/tmp/deploy/images/<machine>/`

3.3.3 Patch Packages

How to Patch and Rebuild a Package

1. `$ cd <sdk-install-dir>/build_<machine>`
2. `$ bitbake -c cleansstate <package-name>`
3. `$ cd <RECIPE_FOLDER>`

> **NOTE**
> Use bitbake `<package-name> -e | grep ^FILE_DIR` to get the value of `<RECIPE_FOLDER>`

4. `$ mkdir -p <RECIPE_FOLDER>/files`
5. Copy patch into `<RECIPE_FOLDER>/files`
6. Modify `<BB_FILE>` and add follow content in `package-name-<version>.bb` file

```
SRC_URI += "file://<name-of-patch1> \
                  file://<name-of-patch2> \
                                ... \
                  file://<name-of-patchn>"
```

7. Rebuild this package:

```
$ bitbake <package-name>
```

3.3.4 Extract Source Code

How to Extract the Source Code for a Package

To extract the source code of a package, do the following:

1. `$ bitbake -c cleansstate <package-name>`
2. `$ bitbake -c patch <package-name>`
3. `$ cd <S>`

> **NOTE**
> Use `bitbake -e <package-name> | grep ^S=` to get the value of `<S>`.

For example, to do a U-boot of a T2080RDB processor.

1. `$ bitbake -c cleansstate u-boot`
2. `$ bitbake -c patch u-boot`

> **NOTE**
> U-boot source code is installed in the folder: `build_t2080rdb/tmp/work/t2080rdb-fsl-linux/u-boot-qoriq/2016.01+fslgit-r0/git/`

3.3.5 Customize Root Filesystem

How to Customize a Root Filesystem

Packages included in a rootfs can be customized by editing the corresponding recipe:

```
fsl-image-mfgtool:sources/meta-freescale/recipes-fsl/images/fsl-image-mfgtool.bb
```

```
fsl-image-core:sources/meta-freescale/recipes-fsl/images/fsl-image-core.bb
fsl-image-full:sources/meta-freescale/recipes-fsl/images/fsl-image-full.bb
fsl-image-virt: sources/meta-freescale/recipes-fsl/images/fsl-image-virt.bb
fsl-image-minimal:sources/meta-freescale/recipes-fsl/images/fsl-image-minimal.bb
fsl-image-kernelitb:sources/meta-freescale/recipes-fsl/images/fsl-image-kernelitb.bb
fsl-toolchain:sources/meta-freescale/recipes-fsl/images/fsl-tooichain.bb
```

The rootfs type can be customized by setting the IMAGE_FSTYPES variable in the above recipes.

Supported rootfs types include the following:

```
cpio
cpio.gz cpio.xz
cpio.lzma
cramfs
ext2
ext2.gz
ext2.gz.u-boot
ext2.bz2.u-boot
ext3
ext3.gz.u-boot
ext2.lzma
jffs2
live
squashfs
squashfs-lzma
ubi
tar
tar.gz
tar.bz2
tar.xz
```

Path of source tarballs and patches:

- Package source tarballs are in the folder named `sources`, which is in the same folder level as `build_<machine>`.
- Patches are in the corresponding recipe folder

Specify the preferred version of package:

`<PREFERRED_VERSION_pkgname>` is used to configure the required version of a package.

If `<PREFERRED_VERSION>` is not defined, Yocto Project will pick up the recent version. For example, to downgrade Samba from 3.4.0 to 3.1.0: add `PREFERRED_VERSION_samba = "3.1.0"` in `<sdk-install-dir>/sources/meta-freescale/conf/machine/<machine>.conf`

Rebuild rootfs:

```
$ bitbake <image-target>
```

3.3.6 Build Native Packages

How to Build Native Packages for the Host

Native packages such as cst-native are supported in Yocto Project. To build a native package, do the following:

```
$ bitbake cst-native
```

NOTE

The binaries can be found in `build_<machine>/tmp/sysroot/`

3.3.7 Install the Standalone Toolchain

Build and install the standalone toolchain with Yocto Project:

1.
```
$ . ./fsl-setup-env -m <machine>
```

2.
```
$ bitbake fsl-toolchain
```

3.
```
$ cd build_<machine>/tmp/deploy/sdk
```

4.
```
$ ./fsl-qoriq-glibc-<host_arch>-<core>-toolchain-<release>.sh
```

NOTE

The default installation path for standalone toolchain is `/opt/fsl-qoriq/2.0/`. The install folder can be specified during the installation procedure.

To use the installed toolchain, go the the location where the toolchain is installed and source the `environment-setup-<core>` file. This will set up the correct path to the build tools and also export some environment variables relevant for development (eg. `$CC`, `$ARCH`, `$CROSS_COMPILE`, `$LDFLAGS` etc).

To invoke the compiler, use the $CC variable (eg. `$CC <source files>`).

NOTE

This is a sysrooted toolchain. GCC needs option `--sysroot=<path-to-target-sysroot>` to find target fragments and libraries (eg. crt*, libgcc.a) as the default sysroot is poisoned to /not/exist. However, when invoking the compiler through the $CC variable, there is no need to pass the --sysroot parameter as it is already included in the variable (check by running echo $CC).

3.3.8 How to Use Multilib

Build an rootfs image with multilib libraries (e.g. ppc64 rootfs with lib32).

1. Use the following setting in local.conf (this example is for e6500 targets)

```
require conf/multilib.conf
MULTILIBS = "multilib:lib32"
DEFAULTTUNE_virtclass-multilib-lib32 = "ppce6500"
```

2. Add `IMAGE_INSTALL += "lib32-<pkg>"` in `<image_target>.bb`
3. `$ bitbake <rootfs_image>`

3.3.9 Mixture Build Mode for e6500 Targets

Both "64bit kernel + 32bit userspace" and "64bit kernel + 64bit userspace" are supported by e6500 targets.

Below introduces how to built 64bit rootfs for e6500 targets.

1. Install both source ISO and e6500-64b cache ISO in the same folder
2. `$ cd <sdk-install-dir>`
3. `$ . ./fsl-setup-env -m <machine> # e.g. . ./fsl-setup-env -m t2080rdb-64b`
4. `$ bitbake <rootfs-target>`

The standalone toolchain generated by Yocto Project doesn't support multilib, 32bit and 64bit build should be done by different toolchain. This SDK supports to build 32-bit and 64-bit standalone toolchains. Both prebuilt toolchains are provided in the image ISO. The kernel image should be built using `fsl-qoriq-glibc-*-ppc64e6500-.toolchain-<sdk_version>+*.sh`.

- `fsl-qoriq-glibc-i686-ppce6500-toolchain-<sdk_version>.sh`: toolchain for building 32-bit images on i686 machine.
- `fsl-qoriq-glibc-i686-ppc64e6500-toolchain-<sdk_version>.sh`: toolchain for building 64-bit images on i686 machine.
- `fsl-qoriq-glibc-x86_64-ppce6500-toolchain-<sdk_version>.sh`: toolchain for building 32-bit images on x86_64 machine.
- `fsl-qoriq-glibc-x86_64-ppc64e6500-toolchain-<sdk_version>.sh`: toolchain for building 64-bit images on x86_64 machine.

3.3.10 Build rootfs which Supports OpenJDK

How to build the rootfs which supports openjdk.

The SDK supports openjdk-7 on 32b targets and 64b targets, Jtreg has been used to verify the openjdk.

Following describes how to build the rootfs image of java support.

1. `$ . ./fsl-setup-env -m <machine>`
2. `$ bitbake java-test-image`

NOTE

The rootfs image with openjdk support can be found in `build_<machine>/tmp/deploy/images/<machine>/`

3.3.11 Deploy Openstack

How to deploy OpenStack on QorIQ platform

Setup Diagram

Host Machine based on QorIQ platform can be used for Openstack Multiple Node demo in the following way:

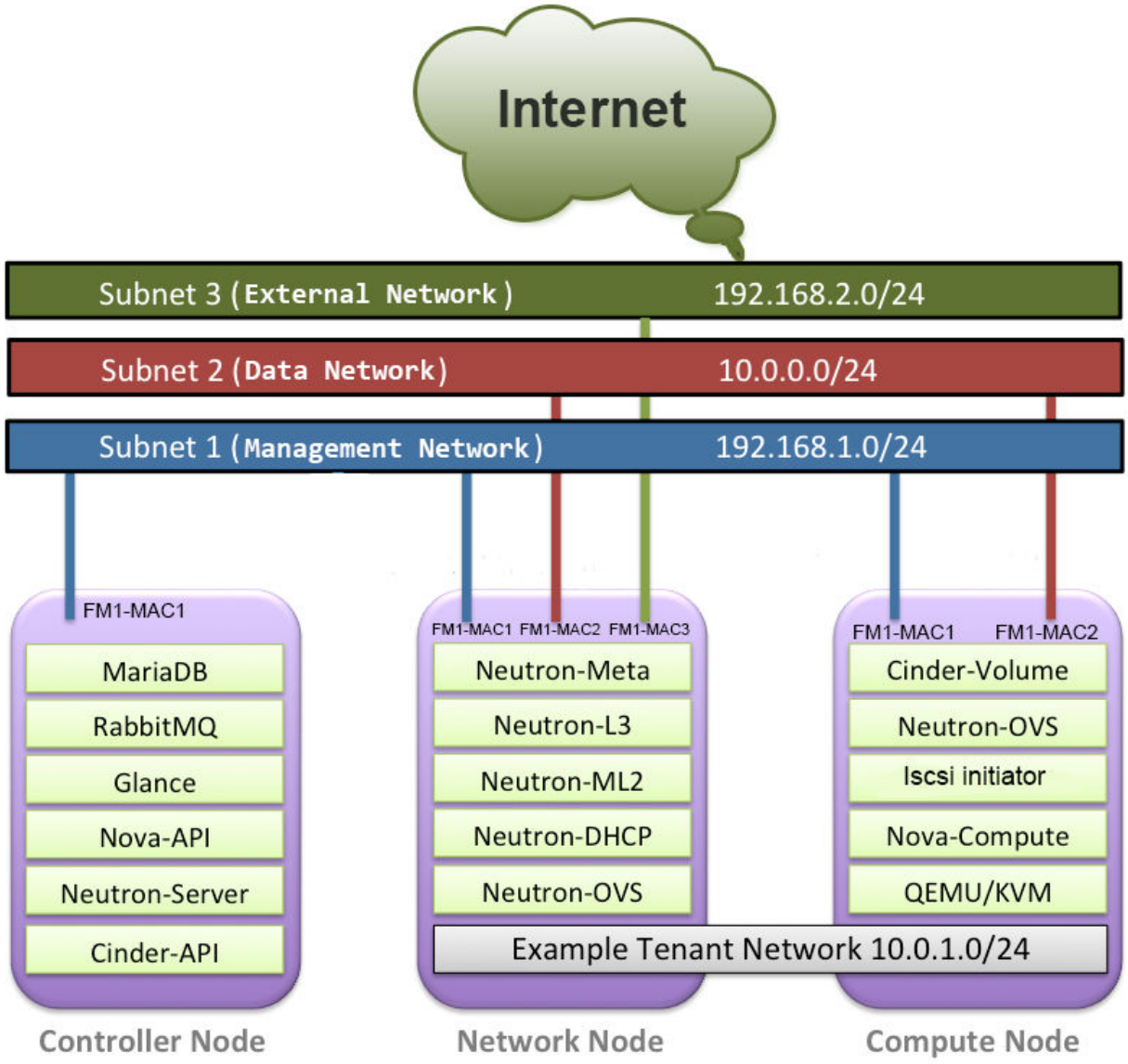


Figure 1. Host Machine based on QorIQ Platform Setup Diagram

Build Openstack images for QorIQ Platform

To build Openstack images for QorIQ Platform, X86 Laptop/Host machine for Yocto Project build should be available. After build environment is ready, refer to following steps to build images:

1. Install QorIQ SDK ISO on X86 host
   a. Install cache ISO(`QorIQ-SDK-<version>-<target>-CACHE-<yyyymmdd>-yocto.iso`)
   b. Then install source(`QorIQ-SDK-<version>-SOURCE-<yyyymmdd>-yocto.iso ) ISO`
2. After those two ISOs install successfully, create project for target platform

```
$ . ./fsl-setup-poky -m <targer_platform>
```

Or

```
$ . ./fsl-setup-poky -m <targer_platform>
```

3. Add following line into conf/local.conf

```
sources/meta-DELTA_KERNEL_DEFCONFIG_append = " <ISO Install folder>/meta-fsl-networking/recipes-kernel/linux/files/openstack_config "
```

4. Build images for Openstack

```
Openstack Controller
$ bitbake openstack-image-controller
Openstack Network Node
$ bitbake openstack-image-network
Openstack Compute Node
$ bitbake openstack-image-compute
```

5. After the image builds successfully, you will see following images under directory

```
cd tmp/deploy/images/<target_platform>
Controller Rootfs:
        openstack-image-controller-<target_platfrom>.tar.gz
Network Node Rootfs:
        openstack-image-network-<target_platform>.tar.gz
Compute Node Rootfs:
       openstack-image-compute-<target_platform>.tar.gz
```

6. Transfer those images to tftpserver for SATA Boot Deployment

Deploy Openstack images(controller/compute/network) on Target platform

1. Bootup target platform by referring target platform deployment guide
2. After booting, login using username `'root'`
3. Unmount the Hard drive if it's already mounted
4. Create the partition in the Hard drive using the following commands:
   a. Run the following command to enter into partitioning system

```
$ fdisk /dev/sda
```

   b. Enter `'n'` and press enter to create new partition

```
Command (m for help): n
```

   c. Enter `'p'` and press enter to create a primary partition

```
Command action
      e   extended
      p   primary partition (1-4)
      p
```

   d. Enter `'1'` and press enter to create partition number 1

```
Partition number (1-4):1
```

   e. Press enter to select first cylinder as `'1'`

```
First cylinder (1-1018, default 1):
```

f. Press enter to select complete disk size

```
Last cylinder or +size or +sizeM or +sizeK (1-1018, default 1018):
```

g. Enter 'w' and press enter to write the changes to Hard drive

```
Command (m for help): w
```

5. Format the partition as ext3 file system using following command

```
$ mkfs.ext2 /dev/sda1
```

6. Mount the formatted partition to local folder. Run the following command to execute it

```
$ mkdir /media/sda1
$ mount /dev/sda1 /media/sda1
```

7. Download the openstack rootfs to SATA HDD Drirve

```
$ scp <username>@<tftp_server>:<Openstack_image_dir>/openstack-image-controller-
<target_platform>.tar.gz /media/sda1
```

8. Uncompress the tarball to SATA HDD Drive.

```
$ cd /media/sda1
$ tar -zxf openstack-image-controller-<target_platform>.tar.gz
```

9. Umount the SATA HDD Drive and reboot system

```
$ umount /media/sda1; reboot
```

10. Boot the board with Hard drive

```
For PPC platform:
=> setenv sataboot 'setenv bootargs root=/dev/sda1 rootfstype=ext2 rootdelay=10 rw console=
$consoledev,$baudrate $othbootargs;tftp $loadaddr $bootfile;tftp $fdtaddr $fdtfile;bootm
$loadaddr - $fdtaddr'
Note: assume the SATA HDD drive is sda1
=> run sataboot
For LS2085ARDB/LS2088AARDB platform:
=> tftp a0000000 <ls2085ardb_kernel_itb>.itb
=> setenv bootargs 'root=/dev/sda1 rw rootdelay=10 console=ttyS1,115200 earlycon=uart8250,mmio,
0x21c0600 default_hugepagesz=2m hugepagesz=2m hugepages=256
=> bootm a0000000:kernel@1 - a0000000:fdt@1
For LS1043ARDB platform:
=> tftp a0000000 <ls1043ardb_kernel_itb>.itb
=> setenv bootargs 'root=/dev/sda1 rw rootdelay=10 console=ttyS0,115200
earlyprintk=uart8250-8bit,0x21c0600,115200 default_hugepagesz=2m hugepagesz=2m hugepages=256
=> bootm a0000000:kernel@1 - a0000000:fdt@1
```

NOTE

Make sure at least two ethernet ports on target platform are available. One port is for management interface, another one is for data interface.

Configuration for Openstack

1. Please refer to Openstak Kilo configuration as following link:

```
http://docs.openstack.org/kilo/install-guide/install/apt/content/index.html
```

References

OpenStack Installation Guide:

```
*http://docs.openstack.org/kilo/config-reference/content/list-of-compute-config-options.html
*hhttp://docs.openstack.org/kilo/install-guide/install/apt/openstack-install-guide-apt-kilo.pdf
```

3.3.12 Shared State (sstate) Cache

BitBake has the capability to accelerate builds based on previously built output. This is done using "shared state" files which can be thought of as cache objects and this option determines where those files are placed. The shared state cache (sstate-cache), as pointed to by SSTATE_DIR.

1. Use the following setting in local.conf:

```
SSTATE_DIR="<absolute_path_of_cache_folder>"
e.g. SSTATE_DIR = "/home/yocto/sdk/sstate-cache/"
```

NOTE

Some packages have no caches because the ISO uses 4GB of space. Thus, cache size is limited by ISO size. Some packages (e.g. u-boot, kernel, rcw, hv-cfg, fmc, depmodwrapper-cross, keymaps, base-files, merge-files, shadow-securetty, etc.) have no caches and building them requires about 20 minutes.

3.3.13 FAQ

Frequently Asked Questions about Yocto Project

Q: How do I install source code, modify source code and rebuild u-boot/kernel?

A: Use the following steps:

1. `$ cd <sdk-install-dir>/build_<machine>`
2. If the source code has been installed, please skip this step.
   - `$ bitbake -c patch <package-name>`
   - `$ cd <S>` and do change

NOTE

Use `bitbake -e <package-name> | grep ^S =` to get the value of `<S>`.

3. Modify configure (dts can also be modified):
   - Modify the `UBOOT_CONFIG` variable in `sources/meta-freescale/conf/machine/<machine>.conf` or update the `KERNEL_DEFCONFIG` variable in `sources/meta-freescale/conf/machine/<machine>.conf`
4. Rebuild images:
   - `$ cd <sdk-install-dir>/build_<machine>`
   - `$ bitbake -c compile -f <package-name>`
   - `$ bitbake <package-name>`

NOTE

u-boot.bin, Image and dtb files can be found in build_<machine>/tmp/deploy/images/<machine>.

Or set <ARCH> and <CROSS_COMPILE> and build the u-boot/kernel manually

Q: How do I build u-boot/kernel with debugger (CodeWarrior support)?

A: *For u-boot:*

1. $ cd <sdk-install-dir/build_<machine>
2. $ bitbake -c cleansstate u-boot
3. Modify the u-boot-qoriq_<version>.bb file and add following content:
   - $ cd sources/meta-freescale/recipes-bsp/u-boot
   - $ add 'EXTRA_OEMAKE += "CONFIG_CW=1"' in u-boot-qoriq_<version>.bb file
4. Rebuild u-boot:
   - $ bitbake u-boot

For kernel:

1. $ cd <sdk-install-dir>/build_<machine>.
2. If kernel source code is not installed, install the kernel source code first.
   - $ bitbake -c cleansstate virtual/kernel
   - $ bitbake -c patch virtual/kernel
3. Configure kernel to enable CW support:
   - $ bitbake -c menuconfig virtual/kernel
   - Enable Kernel hacking -> Include CodeWarrior kernel debugging in kernel configuration UI.
4. Rebuild kernel:
   - $ bitbake virtual/kernel

Q: How do I view the content of rootfs?

A: The expanded rootfs is in <IMAGE_ROOTFS>

NOTE

Use bitbake -e <rootfs-name> | grep ^IMAGE_ROOTFS= to get the value of <IMAGE_ROOTFS>.

Q: How do I add a pre-built binary into the rootfs?

A: Do the following:

1. $ cd <sdk-install-dir>.
2. Add the files:
   - $ cd sources/meta-freescale/recipes-extended/merge-files
   - Put the files into files/merge, e.g. put bash into files/merge/, bash will be included in /home/root/ of the new rootfs.
3. Build new rootfs image:
   - $ bitbake <rootfs-target>

Q: What's an example of using dtc to reverse to a dts from a dtb?

A: Do the following

1. ```
   $ export PATH=<path-of-dtc>:$PATH
   ```
2. ```
   $ dtc -I dtb -O dts -o name-of-dts name-of-dtb
   ```

Q: How do I build fsl-toolchain?

A: Do the following

1. ```
   $bitbake fsl-toolchain
   ```

NOTE

fsl-qoriq-glibc-x86_64-<target>-toolchain-<VERSION>.sh can be found in `build_<machine>/tmp/deploy/sdk/`

NOTE

fsl-qoriq-glibc-x86_64-<target>-toolchain-<VERSION>.sh runs and the toolchain can be installed in special path

Q: I failed to get source tarball of a packages with wget. What should I do?

A: Use the `--no-check-certificates` optionwhen Yocto Project uses `wget` to fetch source tarball from internet, the option is only available when `wget` was configured at build time with SSH support. Ensure that `wget` on your host is built with SSH support.

Q: How do I include toolchain in rootfs?

A: The toolchain runs on target board, which is the same exact version as the cross tools in this SDK, is only included in `fsl-image-full` rootfs, if you want to add toolchain into other rootfs images, do the following:

1. Edit `fsl-image-minimal.bb/fsl-image-core.bb` to add `packagegroup-core-buildessential` in the `IMAGE_INSTALL` variable.
2. `$ bitbake <rootfs-target>.`

Q: How do I use standalone toolchain to build kernel?

A: If you want to build the linux kernel using toolchain, you can do the following:

1. ```
   cd <toolchain-install-dir>
   ```
2. ```
   source environment-setup-<core>-fsl-linux
   ```
3. ```
   export LDFLAGS=""
   ```
4. ```
   cd <linux-src-dir>
   ```
5. ```
   make mrproper
   ```
6. ```
   ./scripts/kconfig/merge_config.sh
   arch/arm64/configs/defconfig
   arch/arm64/configs/freescale.config
   ```
7. ```
   make CC="$CC" LD="$LD"
   ```

Chapter 4
Deploy U-Boot, Linux Kernel, and Root Filesystem to a Reference Design Board (RDB)

4.1 Introduction

This chapter describes how to deploy U-Boot, Linux kernel and root file system to the NXP Reference Design Board (RDB) . The guide starts with generic host and target board pre-requisites. This is followed by board-specifc configuration:

- Switch Settings
- U-Boot environment Variables
- Device Microcodes
- Management Complex (MC), DPC, DPL
- Reset Configuration Word (RCW) and Ethernet Interfaces (if applicable)
- System Memory Map
- Flash Bank Usage

The "Switch Settings" section within each guide shows the default switch settings for the reference design board. If more information is needed beyond the scope of the default configuration, refer to the reference design board's Quick Start Guide and Reference Manual/ User Manual.

For reference design boards with more than one QSPI flash, the "Programming a New U-Boot, RCW, Management Complex" section describes how the user can individually or simultaneously update U-Boot, RCW, and Management Complex, DPC, DPL by flashing them to the board's second QSPI flash prior to deployment.

Once the board is set-up and configured appropriately, select one of the following deployment methods:

- Ramdisk deployment from TFTP
- Ramdisk deployment from Flash
- NFS deployment
- Harddisk deployment (if applicable)
- SD deployment (if applicable)
- Hypervisor deployment (if applicable)

Each of these guides will step you through the deployment method of your choice.

4.2 Basic Host Set-up

Since TFTP will be used to download files onto the target board, a TFTP server must be running on your host system. If you are going to use NFS deployment then an NFS server must also be running on your host system.

Once TFTP and NFS servers are installed, use the following generic instructions to complete the host set-up:

1. Create the tftboot directory.

```
mkdir /tftpboot
```

2. Copy over kernel, bootloader, and flash filesystem images for your deployment to the /tftpboot directory:

```
cp <yocto_work_dir>/build_<platform>_release/tmp/deploy/images/* /tftpboot
```

3. Use Yocto Project to generate a tar.gz type file system, and uncompress it in <nfs_root_path>.
4. Edit /etc/exports and add the following line:

```
<nfs_root_path> <target_board_IP> (rw,no_root_squash, async)
```

5. Edit /etc/xinetd.d/tftp to enable TFTP server:

```
service tftp
{
disable= no
socket_type= dgram
protocol= udp
wait= yes
user= root
server= /usr/sbin/in.tftpd
server_args= /tftpboot
}
```

6. Restart the nfs and tftp servers on your host:

```
/etc/init.d/xinetd restart
/etc/init.d/nfs restart
```

7. Connect the board to the network.
8. Connect the target to the host via a cross cable serial connection.
9. Open a serial console tool on the host system and set it up to talk to the target board:
   - Select appropriate serial device.
   - Configure the serial port with the following settings: Baud rate = 115,200; Data = 8 bit; Parity = none; Stop = 1 bit; Flow control = none.
   - Power on board and see the console prompt.

NOTE

(i) The Linux distribution running on your host will determine the specific instructions to use.

(ii) Steps 3 and 4 are only necessary when using NFS deployment.

4.3 Target Board Set-up

Once the host set-up is complete, follow these steps to set-up and power on the target board:

1. Power off the Target board system if the power is already on.

2. Connect the Target board to the network via an Ethernet port on the board.
3. Connect the Target board to the host machine via the serial port with an RS-232 cable and the joined NXP adaptor cable, if needed.
4. Start the serial console tool on the host system.
5. Verify that all switches and jumpers are setup correctly as described in the board's Reference Manual/User Guide.
6. Power on the board.

Below is an example of a typical U-Boot log:

```
U-Boot 2016.012.0+g2ea81ad (Apr 10 2016 - 14:30:36 +0800)

CPU0:  T1023E, Version: 1.0, (0x85490010)
Core:  e5500, Version: 2.1, (0x80241021)
Single Source Clock Configuration
Clock Configuration:
       CPU0:1200 MHz, CPU1:1200 MHz,
       CCB:400  MHz,
       DDR:800  MHz (1600 MT/s data rate) (Asynchronous), IFC:100  MHz
       FMAN1: 600 MHz
       QMAN:  400 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 0810000c 00000000 00000000 00000000
       00000010: 3b800003 00000012 ec02f000 21000000
       00000020: 00000000 00000000 00000000 00022800
       00000030: 00000130 04020200 00000000 00000006
I2C:   ready
Board: T1023RDB, RevD, boot from NOR vBank4
SERDES Reference Clocks:
SD1_CLK1=100.00MHZ, SD1_CLK2=125.00MHz
SPI:   ready
DRAM:  Detected UDIMM Fixed DDR4 on board
2 GiB (DDR4, 32-bit, CL=11, ECC off)
Flash: 128 MiB
L2:    256 KiB enabled
Corenet Platform Cache: 256 KiB enabled
Using SERDES1 Protocol: 119 (0x77)
SEC: RNG instantiated
NAND:  512 MiB
MMC:   FSL_SDHC: 0
EEPROM: NXID v1
PCIe1: Root Complex, x1 gen1, regs @ 0xfe240000
  01:00.0     - 8086:107d - Network controller
PCIe1: Bus 00 - 01
PCIe2: Root Complex, x1 gen1, regs @ 0xfe250000
  03:00.0     - 8086:107d - Network controller
PCIe2: Bus 02 - 03
PCIe3: disabled
In:    serial
Out:   serial
Err:   serial
Net:   Fman1: Uploading microcode version 108.4.5
PHY reset timed out
e1000: 00:15:17:80:ae:88
       e1000: 00:15:17:8a:c6:82
       FM1@DTSEC1, FM1@DTSEC3, FM1@DTSEC4, e1000#0 [PRIME]
Warning: e1000#0 MAC addresses don't match:
```

```
Address in SROM is         00:15:17:80:ae:88
Address in environment is  00:e0:0c:00:76:03
, e1000#1
Warning: e1000#1 MAC addresses don't match:
Address in SROM is         00:15:17:8a:c6:82
Address in environment is  00:e0:0c:00:76:04

=>
```

NOTE

If the target board does not have a working U-Boot, see System Recovery on page 406.

4.4 Supported Boards

4.4.1 B4420QDS

4.4.1.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.1.2 Switch Settings

The development system has user selectable switches for evaluating different boot options for the B4420 device. The table below lists the default switch settings. For a description of these settings, see B4420 Reference Manual, B4860QDS User's Guide and B4860QDSHGS.

Table 15. Default DIP Switch Settings

	1	2	3	4	5	6	7	8
SW1	OFF [0]	OFF [0]	OFF [0]	OFF [0]	OFF [0]	OFF [0]	OFF [0]	OFF [0]
SW2	ON	OFF	ON	OFF	ON	ON	OFF	OFF
SW3	OFF	OFF	OFF	ON	OFF	OFF	ON	OFF
SW5	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON

Table 16. Frequency Settings in MHz

SysClk	CPU Core (e6500)	CCB	DDR	FMan	DDRCLK	StarCore (SC3900)	CPRI (Maple)	eTVPE (Maple)	ULB (Maple)
66.67 MHz	1600	667	800	667	133.33 MHz	1200	600	1000	800

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 17. Boot Device Scenarios

Alternate Boot Device	Change these DIP switch settings (0=OFF/1=ON)
a) NAND boot with ECC disabled	• SW1 [1] = 1 • SW2 [1] = 0 • SW3 [1:4] = 1000
b) NAND boot with ECC enabled	• SW1 [1] = 1 • SW2 [1] = 1 • SW3 [1:4] = 1000
c) NOR boot	• SW1 [1] = 0 • SW2 [1] = 1 • SW3 [1:4] = 0001
d) Hard Code setting	• SW2 [1] = 0 • SW2 [5] = 0 • SW3 [1:8] = 01000011
e) I2C EEPROM setting	• SW2 [1] = 1 • SW3 [1:8] = 00100100

4.4.1.3 U-Boot Environment Variables

4.4.1.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig=fsl_ddr:ctlr_intlv=null,bank_intlv=cs0_cs1;en_cpc:cpc1,usb1:dr_mode=host,phy_type=ulpi;fs
l_fm1_xaui_phy:xfi
```

NOTE

For USB gadget set "dr_mode=peripheral"

VID Support for B4420

The fuse status register provides the values from on-chip voltage ID efuses programmed at the factory. These values define the voltage requirements for the chip, specifically VDD (core voltage). U-Boot reads FUSESR and translates the values into the appropriate commands to set the voltage output value of an external voltage regulator. B4420QDS has a PowerOne ZM7300 programmable digital Power Manager, which is programmed as per the value read from the fuses.

There is an environment variable also provided to override the fuse specified voltage.

```
"b4qds_vdd_mv"
```

For example:

```
=> setenv b4qds_vdd_mv 950
=> save
=> reset
```

The commands above will override the Voltage with 0.95V.

```
=> setenv b4qds_vdd_mv 1100
=> save
=> reset
```

The commands above will override the Voltage with 1.1V.

NOTE

The time of Power on the board boots at 1.0V default and then the Voltage adjustment happens in the U-Boot flow.

Changing Board Personalities

Use the following steps to change from a B4860QDS to a B4420QDS:

1. Boot from vbank0
2. Flash vbank2 with B4420 RCW binary and U-Boot
3. Type the following commands on the U-Boot prompt:

```
mw.b ffdf0040 0x30;
mw.b ffdf0010 0x00;
mw.b ffdf0062 0x02;
mw.b ffdf0050 0x02;
mw.b ffdf0010 0x30;
```

4. Reset

4.4.1.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
```

```
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.1.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for B4420 version information (e.g., B4420E version 2.2) and also for the version of FMan microcode currently flashed on the B4420QDS (e.g. microcode version 106.4.18). For QorIQ SDK 2.0, the following FMan microcode binary should be used:

- For silicon revision 2.2:

```
fsl_fman_ucode_b4860_r2.2_106_4_18.bin (*)
fsl_fman_ucode_b4860_r2.2_108_4_5.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.1.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK 2.0 contains the following RCW binary for use on the B4420QDS:

- N_PNSS_0x18_0x9E/rcw_2sgmii_1600mhz_rev2.bin (default)

The RCW directories' names for the B4420QDS conform to the following naming convention:

```
a_bcde_f_g
```

Table 18. B4420QDS Naming Convention Legend

Slot	Convention
a	'N' if not available/not supported
b [What is available in Slot 1] c [What is available in Slot 2]	'N' if not available/not used 'P' if PCIe 'R' if SRIO 'S' if SGMII 'A' is AURORA
d	'S' if RGMII is there on phy1 of dual SGMII phy VSC8662 'N' if not available /not used
e	'S' if RGMII is there on phy2 of dual SGMII phy VSC8662 'N' if not available /not used
f	Hex value of serdes1 protocol value
g	Hex value of serdes2 protocol value

For example,

```
N_PNSS_0x9_0xa
```

means:

- RGMII is unsupported
- PCIE on Slot 1
- Nothing on Slot 2
- SGMII on phy1
- SGMII on phy2
- SERDES1 protocol is 0x9
- SERDES2 protocol os 0xa

Below is a table that maps the different ports available on B4420 and their names in different environments.

Table 19. Port Map

Port in U-boot	Port in Linux	FMAN Address
FM1@dTSEC3	fm1-mac3	0xffe4e4000
FM1@dTSEC4	fm1-mac4	Oxffe4e6000

4.4.1.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Discription	Size
0xF_FFDF_1000	0xF_FFFF_FFFF	Free	2 MB
0xF_FFDF_0000	0xF_FFDF_0FFF	IFC - FPGA	4 KB
0xF_FF81_0000	0xF_FFDE_FFFF	Free	5 MB
0xF_FF80_0000	0xF_FF80_FFFF	IFC - NAND Flash	64 KB
0xF_FF00_0000	0xF_FF7F_FFFF	Free	8 MB
0xF_FE00_0000	0xF_FEFF_FFFF	CCSRBAR	16 MB
0xF_F801_0000	0xF_FDFF_FFFF	Free	95 MB
0xF_F800_0000	0xF_F800_FFFF	PCI Express 1 I/O Space	64 KB
0xF_F600_0000	0xF_F7FF_FFFF	Queue manager software portal	32 MB
0xF_F400_0000	0xF_F5FF_FFFF	Buffer manager software portal	32 MB
0xF_F000_0000	0xF_F3FF_FFFF	Free	64 MB
0xF_E800_0000	0xF_EFFF_FFFF	IFC - NOR Flash	128 MB
0xF_E000_0000	0xF_E7FF_FFFF	Promjet	128 MB
0xF_A100_0000	0xF_DFFF_FFFF	Free	1008 MB
0xF_A000_0000	0xF_A0FF_FFFF	MAPLE0, MAPLE1, MAPLE2	16 MB
0xF_0040_0000	0xF_9FFF_FFFF	Free	12 GB
0xF_0000_0000	0xF_003F_FFFF	DCSR	4 MB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Discription	Size
0xC_4000_0000	0xE_FFFF_FFFF	Free	11 GB
0xC_2000_0000	0xC_3FFF_FFFF	Free (Since no sRIO)	512 MB
0xC_0000_0000	0xC_1FFF_FFFF	PCI Express 1 Mem Space	512 MB
0x1_0000_0000	0xB_FFFF_FFFF	Free	44 GB
0x0_8000_0000	0x0_FFFF_FFFF	SC DDR (DDRC1-CS1)	2 GB
0x0_0000_0000	0x0_7FFF_FFFF	PA DDR (DDRC1-CS0)	2 GB

4.4.1.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration B4420QDS preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, please refer to the U-Boot log.

```
CPU0:  B4420E, Version: 2.2, (0x86890222)
Core:  e6500, Version: 2.0, (0x80400120)
Clock Configuration:
       CPU0:1600 MHz, CPU1:1600 MHz,
       CCB:666.667 MHz,
       DDR:800  MHz (1600 MT/s data rate) (Asynchronous), IFC:166.667 MHz
       FMAN1: 666.667 MHz
       QMAN:  333.333 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 14000c18 0f001218 00000000 00000000
       00000010: 2e9d0000 80002400 ec025000 a9000000
       00000020: 00000000 00000000 00000000 0003b000
       00000030: 00000000 14000020 00000000 00000011
Board: B4420QDS, Sys ID: 0x21, Sys Ver: 0x13, vBank: 2
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>qixis_reset
```

- Switch to alternate bank:

```
=>qixis_reset altbank
```

Known Bugs, Limitations, or Technical Issues

Sometime "qixis_reset altbank" hang.

Workaround:- Please use below commands intesad of "qixis_reset" commands for NOR vbank selection.

Moving from vbank0 --> vbank2 :

mw.b ffdf0010 0x00; mw.b 0xffdf0040 0x30; mw.b 0xffdf0050 0x2;mw.b 0xffdf0010 0x30; reset

Moving from vbank2 --> vbank0 :

mw.b ffdf0010 0x00; mw.b 0xffdf0040 0x30; mw.b 0xffdf0050 0x0;mw.b 0xffdf0010 0x30; reset

The table below shows the B4420QDS NOR flash memory map.

Table 20. NOR Flash Memory Map

Range Start	Range End	Definition	Size
0xEFF40000	0xEFFFFFFF	U-Boot (current bank)	768 KB
0xEFF20000	0xEFF3FFFF	U-Boot env (current bank)	128 KB
0xEFF00000	0xEFF1FFFF	FMAN Ucode (current bank)	128 KB
0xEF300000	0xEFEFFFFF	rootfs (alternate bank)	12 MB
0xEE900000	0xEE9FFFFF	HV config device tree (alternate bank)	1 MB
0xEE800000	0xEE8FFFFF	Hardware device tree (alternate bank)	1 MB
0xEE700000	0xEE7FFFFF	HV.uImage (alternate bank)	1 MB
0xEE020000	0xEE6FFFFF	Linux.uImage (alternate bank)	6 MB+896 KB
0xEE000000	0xEE01FFFF	RCW (alternate bank)	128 KB
0xEDF40000	0xEDFFFFFF	U-Boot (alternate bank)	768 KB
0xEDF20000	0xEDF3FFFF	U-Boot env (alternate bank)	128 KB

Table continues on the next page...

Table 20. NOR Flash Memory Map (continued)

0xEDF00000	0xEDF1FFFF	FMAN ucode (alternate bank)	128 KB
0xED300000	0xEDEFFFFF	rootfs (current bank)	12 MB
0xEC900000	0xEC9FFFFF	HV config device tree (current bank)	1 MB
0xEC800000	0xEC8FFFFF	Hardware device tree (current bank)	1 MB
0xEC700000	0xEC7FFFFF	HV.uImage (current bank)	1 MB
0xEC020000	0xEC6FFFFF	Linux.uImage (current bank)	6 MB+896 KB
0xEC000000	0xEC01FFFF	RCW (current bank)	128 KB

4.4.1.8 Programming a New U-Boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *qixis_reset altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *qixis_reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>qixis_reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *qixis_reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location

of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>qixis_reset altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *qixis_reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>. <fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>qixis_reset altbank
```

Writing SPD

DDRC takes its configuration parameters from EEPROM that are on I2C bus. To provide configuration to DDRC, its respective EEPROM needs to be programmed with the desired configuration.

EEPROM for DDRC1 is at address 0x51

B4420QDS has only one DDRC, so only one SPD needs to be programmed for this DDR controller.

By default, EEPROMs are write protected. Use the following commands to make them writable:

```
mw.b 0xffdf0055 0x8
```

In order to write SPD, TFTP SPD binaries and write them on EEPROMs using the following commands:

```
tftp 1000000 spd_1866_dual_rank_4GB_DDRC1.bin
i2c write 0x1000000 0x51 0 0x80
reset
```

After configuring EEPROM, reset board to let DDRC settings take effect.

4.4.1.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.1.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

```
=>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
=>saveenv
```

NOTE

ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

```
=>tftp 1000000 <uImage_name>
=>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>tftp 2000000 <platform_dtb_name>
=>bootm 1000000 5000000 2000000
```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.1.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

```
=>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
=>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
=>saveenv
```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
```

```
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.1.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

   b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
```

```
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
=>bootm 1000000 - c00000
```

   Now the board will boot up with NFS filesystem.

4.4.1.9.4 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

   TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

   ```
   =>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>erase e9300000 +$filesize
   =>cp.b 1000000 e9300000 $filesize
   ```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

   ```
   =>tftp 1000000 hv.uImage
   =>erase e8700000 +$filesize
   =>cp.b 1000000 e8700000 $filesize
   ```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

   ```
   =>tftp 1000000 <platform_name>-usdpaa.dtb
   =>erase e8800000 +$filesize
   =>cp.b 1000000 e8800000 $filesize
   ```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

   ```
   =>boot
   ```

4.4.2 B4860QDS

4.4.2.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.2.2 Switch Settings

The Development System has user selectable switches for evaluating different boot options for the B4860QDS device. The table below lists the default switch settings. For a description of these settings, see B4860QDS Reference Manual.

Table 21. B4860QDS Default DIP Switch Settings

	1	2	3	4	5	6	7	8
SW1	OFF [0]	OFF [0]	OFF [0]	OFF [0]	OFF [0]	OFF [0]	OFF [0]	OFF [0]
SW2	ON	ON	ON	ON	ON	ON	OFF	OFF
SW3	OFF	OFF	OFF	ON	OFF	OFF	ON	OFF
SW5	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 22. Boot Device Scenarios

Alternate Boot Device	Change these DIP switch settings (0=OFF/1=ON)
a) NAND boot with ECC disabled	• SW1 [1] = 1 • SW2 [1] = 0 • SW3 [1:4] = 1000
b) NAND boot with ECC enabled	• SW1 [1] = 1 • SW2 [1] = 1 • SW3 [1:4] = 1000
c) NOR boot	• SW1 [1] = 0 • SW2 [1] = 1 • SW3 [1:4] = 0001
d) Hard Code setting	• SW2 [1] = 0 • SW2 [5] = 0 • SW3 [1:8] = 01000011
e) I2C EEPROM setting	• SW2 [1] = 1 • SW3 [1:8] = 00100100

4.4.2.3 U-Boot Environment Variables

4.4.2.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

If you plan to use optical 10G interfaces, you must add information to "hwconfig". This is important. The optical interface will operate erratically without it. It is easiest to do this using the U-Boot "*editenv*" command. Whatever you type will be appended to "hwconfig". The text you need to append is:

```
;fsl_fm1_xaui_phy:xfi
```

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;fsl_fm1_xaui_phy:xfi
```

To use *hwconfig en_cpc*, append the following:

```
en_cpc:cpc2;
```

or

```
en_cpc=cpc1
```

By default, CPC1 is not enabled since only CPC2 should be used. However, by editing the *hwconfig en_cpc* option, CPC1 can also be enabled.

To use 10G PHY, append the following:

```
fsl_fm1_xaui_phy:xfi
```

To use USB gadget, set the following:

```
dr_mode=peripheral
```

NOTE

To use XFI, these changes are needed:

1. "fsl_b4860_serdes2:sfp_amc=sfp" must be in hwconfig
2. Make sure the B4860QDS board has extra heat sinks on VSC3308 devices. They are labeled U45 and U96 near the AMC connector, located at the two sides of the board
3. Use Source Photonics optical modules SPP-10E-SR-CDFD. Then the XFI ports can be used as normal 10G ports.
4. U-Boot configures matrix for CPRI modes according to SerDes protocol number.

VID Support for B4860

The fuse status register provides the values from on-chip voltage ID efuses programmed at the factory. These values define the voltage requirements for the chip, specifically VDD (core voltage). U-Boot reads FUSESR and translates the values into the appropriate commands to set the voltage output value of an external voltage regulator. B4860QDS has a PowerOne ZM7300 programmable digital Power Manager, which is programmed as per the value read from the fuses.

There is an environment variable also provided to override the fuse specified voltage.

```
"b4qds_vdd_mv"
```

For example:

```
=> setenv b4qds_vdd_mv 950
=> save
=> reset
```

The commands above will override the Voltage with 0.95V.

```
=> setenv b4qds_vdd_mv 1100
=> save
=> reset
```

The commands above will override the Voltage with 1.1V.

NOTE

The time of Power on the board boots at 1.0V default and then the Voltage adjustment happens in the U-Boot flow.

Known Bugs, Limitations, or Technical Issues

Sometime "qixis_reset altbank" hang.

Workaround:- Please use below commands intesad of "qixis_reset" commands for NOR vbank selection.

Moving from vbank0 --> vbank2 :

mw.b ffdf0010 0x00; mw.b 0xffdf0040 0x30; mw.b 0xffdf0050 0x2;mw.b 0xffdf0010 0x30; reset

Moving from vbank2 --> vbank0 :

mw.b ffdf0010 0x00; mw.b 0xffdf0040 0x30; mw.b 0xffdf0050 0x0;mw.b 0xffdf0010 0x30; reset

4.4.2.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.2.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for B4860 version information (e.g., B4860E version 2.2) and also for the version of FMan microcode currently flashed on the B4860QDS (e.g., microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 1

```
fsl_fman_ucode_b4860_r1.0_106_3_15.bin
```

- For silicon revision 2

```
fsl_fman_ucode_b4860_r2.0_106_4_15.bin
```

- For silicon revision 2.2

```
fsl_fman_ucode_b4860_r2.2_106_4_18.bin (*)
fsl_fman_ucode_b4860_r2.2_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a New U-boot, RCW, FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.2.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK 2.0 contains RCW binaries for use on the B4860QDS:

- N_RSSS_0x2A_0x8D/rcw_4sgmii_4srio_2xfi_1600mhz_1866ddr_rev2.bin (default)

The RCW directories' names for the B4860QDS conform to the following naming convention:

```
a_bcde_f_g
```

Table 23. B4860QDS Naming Convention Legend

Slot	Convention
a	'R' if RGMII is supported 'N' if not available/not used
b [What is available in Slot 1] c [What is available in Slot 2]	'N' if not available/not used 'P' if PCIe 'X' if XAUI 'R' if SRIO 'S' if SGMII 'A' is AURORA
d	'S' if SGMII is there on phy1 of dual SGMII phy VSC8662 'N' if not available /not used
e	'S' if SGMII is there on phy2 of dual SGMII phy VSC8662 'N' if not available /not used
f	'hex value of serdes1 protocol value'
g	'hex value of serdes2 protocol value'

For example,

```
N_PNSS_0x9_0xa
```

means:

- RGMII is unsupported
- PCIE on Slot 1
- Nothing on Slot 2
- SGMII on phy1
- SGMII on phy2
- SERDES1 Protocol is 0x9 and the SERDES2 Protocol is 0xa

Below is a table that maps the different ports available on B4860QDS and their names in different environments.

Table 24. B4860QDS Port Map

Port in U-boot	Port in Linux
FM1@DTSEC3	fm1-mac3
FM1@DTSEC4	fm1-mac4
FM1@DTSEC5	fm1-mac5
FM1@DTSEC6	fm1-mac6
FM1@TGEC1	fm1-mac9
FM1@TGEC2	fm1-mac10

4.4.2.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Discription	Size
0xF_FFDF_1000	0xF_FFFF_FFFF	Free	2 MB
0xF_FFDF_0000	0xF_FFDF_0FFF	IFC - FPGA	4 KB
0xF_FF81_0000	0xF_FFDE_FFFF	Free	5 MB
0xF_FF80_0000	0xF_FF80_FFFF	IFC - NAND Flash	64 KB
0xF_FF00_0000	0xF_FF7F_FFFF	Free	8 MB
0xF_FE00_0000	0xF_FEFF_FFFF	CCSRBAR	16 MB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Discription	Size
0xF_F801_0000	0xF_FDFF_FFFF	Free	95 MB
0xF_F800_0000	0xF_F800_FFFF	PCI Express 1 I/O Space	64 KB
0xF_F600_0000	0xF_F7FF_FFFF	Queue manager software portal	32 MB
0xF_F400_0000	0xF_F5FF_FFFF	Buffer manager software portal	32 MB
0xF_F000_0000	0xF_F3FF_FFFF	Free	64 MB
0xF_E800_0000	0xF_EFFF_FFFF	IFC - NOR Flash	128 MB
0xF_E000_0000	0xF_E7FF_FFFF	Promjet	128 MB
0xF_A100_0000	0xF_DFFF_FFFF	Free	1008 MB
0xF_A000_0000	0xF_A0FF_FFFF	MAPLE0, MAPLE1, MAPLE2	16 MB
0xF_0200_0000	0xF_9FFF_FFFF	Free	2528 MB
0xF_0000_0000	0xF_01FF_FFFF	DCSR	32 MB
0xC_4008_0000	0xE_FFFF_FFFF	Free	11263 MB
0xC_4000_0000	0xC_4007_FFFF	SC CPC1 as SRAM	512KB
0xC_3000_0000	0xC_3FFF_FFFF	Serial Rapid I/O 2	256 MB
0xC_2000_0000	0xC_2FFF_FFFF	Serial Rapid I/O 1	256 MB
0xC_0000_0000	0xC_1FFF_FFFF	PCI Express 1 Mem Space	512 MB
0x1_0000_0000	0xB_FFFF_FFFF	Free	44 GB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Discription	Size
0x0_8000_0000	0x0_FFFF_FFFF	SC DDR (DDRC1)	2 GB
0x0_0000_0000	0x0_7FFF_FFFF	PA DDR (DDRC2)	2 GB

4.4.2.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration B4860QDS preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, please refer to the U-Boot log:

```
CPU0:  B4860E, Version: 2.2, (0x86880022)
Core:  e6500, Version: 2.0, (0x80400120)
Clock Configuration:
       CPU0:1600 MHz, CPU1:1600 MHz, CPU2:1600 MHz, CPU3:1600 MHz,
       CCB:666.667 MHz,
       DDR:933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:166.667 MHz
       FMAN1: 666.667 MHz
       QMAN:  333.333 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 14000e18 0f001218 00000000 00000000
       00000010: 52b10000 8000a400 dc025000 a9000000
       00000020: 01000000 00000000 00000000 0001b1f8
       00000030: 00000000 14000020 00000000 00000011
Board: B4860QDS, Sys ID: 0x21, Sys Ver: 0x13, vBank: 0
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>qix
```

- Switch to alternate bank:

```
=>qix altbank
```

The table below shows the B4860QDS NOR flash memory map.

Table 25. NOR Flash Memory Map

Range Start	Range End	Definition	Size

Table continues on the next page...

Table 25. NOR Flash Memory Map (continued)

0xEFF40000	0xEFFFFFFF	U-Boot (current bank)	768 KB
0xEFF20000	0xEFF3FFFF	U-Boot env (current bank)	128 KB
0xEFF00000	0xEFF1FFFF	FMAN Ucode (current bank)	128 KB
0xEF300000	0xEFEFFFFF	Rootfs (alternate bank)	12 MB
0xEE900000	0xEE9FFFFF	HV config device tree (alternate bank)	1 MB
0xEE800000	0xEE8FFFFF	Hardware device tree (alternate bank)	1 MB
0xEE700000	0xEE7FFFFF	HV.uImage (alternate bank)	1 MB
0xEE020000	0xEE6FFFFF	Linux.uImage (alternate bank)	6 MB+896 KB
0xEE000000	0xEE01FFFF	RCW (alternate bank)	128 KB
0xEDF40000	0xEDFFFFFF	U-Boot (alternate bank)	768 KB
0xEDF20000	0xEDF3FFFF	U-Boot env (alternate bank)	128 KB
0xEDF00000	0xEDF1FFFF	FMAN ucode (alternate bank)	128 KB
0xED300000	0xEDEFFFFF	Rootfs (current bank)	12 MB
0xEC900000	0xEC9FFFFF	HV config device tree (current bank)	1 MB
0xEC800000	0xEC8FFFFF	Hardware device tree (current bank)	1 MB
0xEC700000	0xEC7FFFFF	HV.uImage (current bank)	1 MB
0xEC020000	0xEC6FFFFF	Linux.uImage (current bank)	6 MB+896 KB
0xEC000000	0xEC01FFFF	RCW (current bank)	128 KB

4.4.2.8 Programming a New U-Boot, RCW, FMan Microcode, and Writing SPD

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *qix altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters on page 102 to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *qix*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>qix altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *qix*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>qix altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *qix*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>. <fman_ucode_start_addr> is the location of microcode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the microcode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>qix altbank
```

Writing SPD

DDRC's take their configuration parameters from EEPROM that are on I2C bus. To provide configuration to both DDRC, their respective EEPROMs need to be programmed with the desired configuration.

EEPROM for DDRC1 is at address 0x51

EEPROM for DDRC2 is at address 0x53

By default, EEPROMs are write protected. Use the following commands to make them writable:

```
mw.b 0xffdf0055 0x8
```

In order to write SPD, TFTP SPD binaries and write them on EEPROMs using the following commands:

```
tftp 1000000 spd_1866_dual_rank_4GB_DDRC1.bin
i2c write 0x1000000 0x51 0 0x80
reset

tftp 2000000 spd_1866_single_rank.bin
i2c write 0x2000000 0x53 0 0x80
reset
```

After configuring both EEPROMs, reset board to let the DDRC1 and DDRC2 settings take effect.

4.4.2.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.2.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 1000000 <uImage_name>
   =>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 2000000 <platform_dtb_name>
   =>bootm 1000000 5000000 2000000
   ```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.2.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   =>saveenv
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

   ```
   =>tftp 1000000 <uImage name>
   =>protect off <kernel_start_addr> +$filesize
   =>erase <kernel_start_addr> +$filesize
   =>cp.b 1000000 <kernel_start_addr> $filesize
   =>protect on <kernel_start_addr> +$filesize
   ```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

   ```
   =>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>protect off <ramdisk_start_addr> +$filesize
   =>erase <ramdisk_start_addr> +$filesize
   =>cp.b 2000000 <ramdisk_start_addr> $filesize
   =>protect on <ramdisk_start_addr> +$filesize
   ```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

   ```
   =>tftp c00000 <platform dtb name>
   =>protect off <dtb_start_addr> +$filesize
   =>erase <dtb_start_addr> +$filesize
   =>cp.b c00000 <dtb_start_addr> $filesize
   =>protect on <dtb_start_addr> +$filesize
   ```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

   ```
   =>boot
   ```

or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.2.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   <nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/nfs restart
   ```

   NOTE

   nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>:
   <your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
   =>setenv netdev <ethx>
   =>saveenv
   ```

   NOTE

   <ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

   ```
   =>tftp 1000000 <uImage name>
   =>tftp c00000 <platform dtb name>
   =>bootm 1000000 - c00000
   ```

   Now the board will boot up with NFS filesystem.

4.4.2.9.4 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

```
=>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>erase e9300000 +$filesize
=>cp.b 1000000 e9300000 $filesize
```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

```
=>tftp 1000000 hv.uImage
=>erase e8700000 +$filesize
=>cp.b 1000000 e8700000 $filesize
```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

```
=>tftp 1000000 <platform_name>-usdpaa.dtb
=>erase e8800000 +$filesize
=>cp.b 1000000 e8800000 $filesize
```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.3 C29XPCIe

4.4.3.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.3.2 Switch Settings

The C29xPCIe has user selectable switches for evaluating different boot options. The tables below list the default switch settings. For a description of these settings, see C29xPCIe User Guide.

Table 26. NOR Boot Default Switch Settings

Switch	1	2	3	4	5	6	7	8
SW4	ON	OFF	ON	OFF	OFF	ON	ON	ON
SW5	OFF	OFF	OFF	OFF	ON	ON	OFF	OFF
SW6	ON	ON	ON	ON	OFF	OFF	OFF	OFF
SW7	OFF	ON	ON	OFF	ON	OFF	OFF	OFF
SW8	ON	ON	ON	ON	OFF	OFF	OFF	OFF

Table 27. SPI Boot Default Switch Settings

Switch	1	2	3	4	5	6	7	8
SW4	ON	OFF	ON	OFF	OFF	ON	ON	ON
SW5	ON	OFF	OFF	ON	ON	ON	OFF	OFF
SW6	ON	ON	ON	ON	OFF	OFF	OFF	OFF
SW7	OFF	ON	ON	OFF	ON	OFF	OFF	OFF
SW8	ON	ON	ON	ON	OFF	OFF	OFF	OFF

Table 28. NAND Boot Default Switch Settings

Switch	1	2	3	4	5	6	7	8
SW4	ON	OFF	ON	OFF	OFF	ON	ON	ON
SW5	ON	ON	OFF	ON	ON	OFF	OFF	OFF
SW6	ON	ON	ON	ON	OFF	OFF	OFF	OFF
SW7	OFF	ON	ON	OFF	ON	OFF	OFF	OFF
SW8	ON	ON	ON	ON	OFF	OFF	OFF	OFF

NOTE

1 means 'OFF', 0 means 'ON' for all SW settings in this document.

C29XPCIE DIP-switch Settings for Boot Mode:

```
SW5[1:4]= 1111 and SW5[6]= 0 for NOR boot
SW5[1:4]= 0110 and SW5[6]= 0 for SPI boot
SW5[1:4]= 0010 and SW5[6]= 1 for NAND boot
```

C29XPCIE DIP-switch Settings for DDR Speed:

```
SW6[1:5]= 00001 for 400MHz(800MT/s data rate)
SW5[1:5]= 00110 for 500MHz(1000MT/s data rate)
SW5[1:5]= 01011 for 600MHz(1200MT/s data rate)
```

PCIe EP Scenario

Configure PCIe RC or EP mode with the switch settings:

Table 29. PCIe EP Scenario Switch Settings

PCIe Port	Host/Agent	Switch Settings
PCIE 1	Host (RC mode)	SW7[5] = OFF(1)
PCIE 1	Agent (EP mode)	SW7[5] = ON(0)

4.4.3.3 U-Boot Environment Variables

4.4.3.3.1 Boot Settings

Secure Boot Settings

- Make sure the hardware pins PO1VDD_EN(1), PO2VDD_EN(2) to be at 1.5V;
- Check the hardware pin VDD_LP whether it was provide the 1.0v power when board is powered on. This requires J15 to be linked ON and J16 pin 1 and pin 2 to be linked ON in the standalone board;
- SW7[6] is used to set secure boot to Disable; when it is ON(0), secure boot is set to Enable;
- Switch settings

  NOR secure boot mode:

```
SW5[1:4]= 1111, SW5[6]= 0, SW7[1]= 0, SW7[6]= 0
```

  SPI secure boot mode:

```
SW5[1:4]= 0110, SW5[6]= 0, SW7[1]= 0, SW7[6]= 0
```

2. Building an Image for Different Boot Modes on C29xPCIe

NOTE

0 stands for ON, 1 stands for OFF

After setting cross-compiler for PowerPC

```
1) For NOR boot
   If SW5[7:8] = 00, bank description as:
      - bank 1 on the flash 0x0000000~0x0ffffff
      - bank 2 on the flash 0x1000000~0x1ffffff
      - bank 3 on the flash 0x2000000~0x2ffffff
      - bank 4 on the flash 0x3000000~0x3ffffff

   make distclean
   make C29XPCIE

   To make sure the SW5[6] set to ON in order to route the CS0 chip select to NOR flash before
power up.
```

```
   => tftp 1000000 u-boot.bin
   For bank4
     => pro off all;era eff40000 effffffff;cp.b 1000000 eff40000 $filesize
   set SW5[1:4]= 1111, SW5[6]=0 then power on the board
   For bank3
     => pro off all;era eef40000 eeffffff;cp.b 1000000 eef40000 $filesize
   set SW5[1:4]= 1111, SW5[6]=0 then power on the board
   For bank2
     => pro off all;era edf40000 edffffff;cp.b 1000000 edf40000 $filesize
   set SW5[1:4]= 1111, SW5[6]=0 then power on the board
   For bank1
     => pro off all;era ecf40000 ecffffff;cp.b 1000000 ecf40000 $filesize
   set SW5[1:4]= 1111, SW5[6]=0 then power on the board

2) For SPI boot
   make distclean
   make C29XPCIE_SPIFLASH

   Need the boot_format tool to generate u-boot-spi.bin from the u-boot.bin.

   =>  tftp 1000000 u-boot-spi.bin
   =>  sf probe 0; sf erase 0 100000; sf write 1000000 0 100000
   set SW5[1:4]= 0110 and SW5[6]= 0(make sure the SW5[6] set to be ON in order to route the CS0
chip select to NOR flash), then power on the board

3) For NAND boot
__ make distclean
   make C29XPCIE_NAND

   => tftp 1000000 u-boot-with-spl.bin
   => nand erase.chip
   => nand write 1000000 0 $filesize
   set SW5[1:4]= 0010, SW5[6]=1, then power on the board.
```

4.4.3.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.3.4 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Address Start	Address End	Memory Type	Size
0x0_0000_0000	0x0_1fff_ffff	DDR	512 MB
0xc_0000_0000	0xc_8fff_ffff	PCI Express MEM	256 MB
0xf_ec00_0000	0xf_efff_ffff	NOR Flash	64 MB
0xf_ffb0_0000	0xf_ffb7_ffff	SRAM	512 KB
0xf_ffc0_0000	0xf_ffc0_ffff	PCI Express IO	64 KB
0xf_ffdf_0000	0xf_ffdf_0fff	CPLD	4 KB
0xf_ffe0_0000	0xf_ffef_ffff	CCSR	1 MB

4.4.3.5 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>cpld reset
```

- Switch to alternate bank:

```
=>cpld reset altbank
```

The tables below show the C29XPCIe memory maps.

Flash Memory Map

Table 30. NOR Flash Memory Map Address

Start Offset	End Offset	Description	Size
0xec00_0000	0xec3f_ffff	Linux kernel image	4 MB
0xec40_0000	0xec4f_ffff	dtb file	1 MB
0xecf2_0000	0xecf3_ffff	U-Boot ENV (alter bank)	128 KB
0xecf4_0000	0xecff_ffff	U-Boot (alter bank)	768 KB
0xed00_0000	0xeeff_ffff	rootfs	32 MB
0xeff2_0000	0xeff3_ffff	U-Boot ENV	128 KB
0xeff4_0000	0xefff_ffff	U-Boot	768 KB

Table 31. SPI Flash Memory Map Address

Start Offset	End Offset	Description	Size
0x0000_0000	0x000f_ffff	U-Boot	1 MB
0x0010_0000	0x0017_ffff	DTB file	512 KB
0x0018_0000	0x0057_ffff	Linux kernel image	4 MB
0x0058_0000	0x0098_ffff	Compressed RFS image	4 MB
0x0098_0000	0x00ff_ffff	JFFS2 RFS image	6.5 MB

Table 32. NAND Flash Memory Map Address

Start Offset	End Offset	Description	Size
0x0000_0000	0x000f_ffff	U-Boot	1 MB
0x0010_0000	0x001f_ffff	NAND DTB file	1 MB
0x0020_0000	0x005f_ffff	Linux kernel image	4 MB
0x0060_0000	0x009f_ffff	Compressed RFS image	4 MB
0x00a0_0000	0x018f_ffff	NAND JFFS2 RFS	15 MB
0x0190_0000	0x01ff_ffff	User area	7 MB

4.4.3.6 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.3.6.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

```
=>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
=>saveenv
```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

```
=>tftp 1000000 <uImage_name>
=>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>tftp 2000000 <platform_dtb_name>
=>bootm 1000000 5000000 2000000
```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.3.6.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   =>saveenv
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

   ```
   =>tftp 1000000 <uImage name>
   =>protect off <kernel_start_addr> +$filesize
   =>erase <kernel_start_addr> +$filesize
   =>cp.b 1000000 <kernel_start_addr> $filesize
   =>protect on <kernel_start_addr> +$filesize
   ```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

   ```
   =>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>protect off <ramdisk_start_addr> +$filesize
   =>erase <ramdisk_start_addr> +$filesize
   =>cp.b 2000000 <ramdisk_start_addr> $filesize
   =>protect on <ramdisk_start_addr> +$filesize
   ```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

   ```
   =>tftp c00000 <platform dtb name>
   =>protect off <dtb_start_addr> +$filesize
   =>erase <dtb_start_addr> +$filesize
   =>cp.b c00000 <dtb_start_addr> $filesize
   =>protect on <dtb_start_addr> +$filesize
   ```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

   ```
   =>boot
   ```

or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.3.6.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   <nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/nfs restart
   ```

   NOTE

   nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>:
   <your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
   =>setenv netdev <ethx>
   =>saveenv
   ```

   NOTE

   <ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

   ```
   =>tftp 1000000 <uImage name>
   =>tftp c00000 <platform dtb name>
   =>bootm 1000000 - c00000
   ```

   Now the board will boot up with NFS filesystem.

4.4.4 LS1021ATWR

4.4.4.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.4.2 Switch Settings

The TWR has user selectable switches for evaluating different boot options for the LS1021A device. Table below lists the default switch settings and the description of these settings.

Table 33. Default Clock Frequency

ARM CPU Core	Platform	DDR rate
1200 MHZ	300 MHZ	1600 MT/S

Table 34. Default LS1021A TWR-PB Switch Setting

Switch	1	2	3	4	5	6	7	8
SW2	ON [1]	OFF [0]	OFF [0]	OFF [0]	ON [1]	ON [1]	ON [1]	ON [1]
SW3	OFF	ON	ON	OFF	OFF	ON	OFF	ON

4.4.4.3 U-Boot Environment Variables

4.4.4.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set in the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified in the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which should disable interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=null,bank_intlv=null
```

4.4.4.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
```

```
=>setenv eth3addr <mac addr3>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

* <ethx> is the Ethernet port on the board connected to the Linux boot server. "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to approriate MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9f:ef:00:00
=>setenv eth1addr 00:04:9f:ef:01:01
=>setenv eth2addr 00:04:9f:ef:02:02
=>setenv eth3addr 00:04:9f:ef:03:03
=>saveenv
```

NOTE

1. For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).
2. In the overwhelming majority of cases, eth<*>addr can be autoset.

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.4.4 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK 2.0 contains the following RCW binary for use on the LS1021ATWR:

- RSR_PPS_70/rcw_1200.bin (default for bank0)
- SSR_PNS_30/rcw_1200.bin
- SSR_PNS_30/rcw_1200_usb2.bin
- SSR_PNS_30/rcw_1200_lpuart.bin (default for bank1)

The RCW directories' names for the LS1021ATWR conform to the following naming convention:

```
abc_def_g
```

Table 35. LS1021ATWR Directories Naming Convention Legend

Slot	Convention
a[What is available for eTSEC1]	'R' indicates RGMII@eTSEC1 is supported 'S' indicates SGMII@eTSEC1 is supported 'N' if not available/not used
b[What is available for eTSEC2]	'R' indicates RGMII@eTSEC2 is supported 'S' indicates SGMII@eTSEC3 is supported 'N' if not available/not used

Table continues on the next page...

Table 35. LS1021ATWR Directories Naming Convention Legend (continued)

c[What is available for eTSEC3]	'R' indicates RGMII@eTSEC3 is supported 'S' indicates SGMII@eTSEC3 is supported 'N' if not available/not used
d	'P' indicates PCIe@slot1 is supported 'N' if not available/not used
e	'P' indicates PCIe@slot2 is supported 'N' if not available/not used
f	'S' indicates sata is supported 'N' if not available/not used
g	Hex value of serdes protocol value

For example,

```
SSR_PNS_30
```

means:

- SGMII@eTSEC1
- SGMII@eTSEC2
- RGMII@eTSEC3
- PCIE [Slot 1]
- SATA [Support]
- SERDES Protocol is 0x30

The RCW file names for the LS1021ATWR conform to the following naming convention:

```
rcw_<frequency>_<specialsetting>.rcw
```

Table 36. LS1021TWR Files Naming Convention Legend

Code		Convention
frequency		Core frequency(MHZ)
specialsetting	bootmode	sdboot (default is nor boot)
	special support	lpuart:used for lpuart sben:Secure boot support usb2:USB 2.0 support

For example,

```
rcw_1200_sd.rcw means rcw for core frequency of 1200MHz with sd boot.
rcw_1200_lpuart.rcw means rcw for core frequency of 1200MHz with nor boot special for enable lpuart.
```

Default rcw:

- SSR_PNS_30/rcw_1200_lpuart.bin[1 SGMII, 1 RGMII, 2D-ACE, lpuart1, 2PCIE, SATA, CAN, SAI]
- RSR_PPS_70/rcw_1200.bin[2 SGMII, 1 RGMII, 1SATA, 1PCIE, CAN]

4.4.4.5 System Memory Map

In 32-bit u-boot, there is a 1:1 mapping of physical address and effective address. After system startup, the boot loader maps physical address and effective address as shown in the following table:

Start Physical Address	End Physical Address	Memory Type	Size
0x0100_0000	0x0FFF_FFFF	CCSR	240MB
0x1000_0000	0x1000_FFFF	OCRAM0	64KB
0x1001_0000	0x1001_FFFF	OCRAM1	64 KB
0x2000_0000	0x20FF_FFFF	DCSR	16MB
0x4000_0000	0x5FFF_FFFF	QSPI	512MB
0x6000_0000	0x67FF_FFFF	NOR Flash	128MB
0x7FB0_0000	0x7FB0_0FFF	Board CPLD	4KB
0x8000_0000	0xFFFF_FFFF	DDR	2GB

4.4.4.6 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration (for LS1021ATWR, SW3[5]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
CPU:   Freescale LayerScape LS1021E, Version: 2.0, (0x87081120)
Clock Configuration:
       CPU0(ARMV7):1200 MHz,
       Bus:300  MHz, DDR:800  MHz (1600 MT/s data rate),
```

```
Reset Configuration Word (RCW):
       00000000: 0608000c 00000000 00000000 00000000
       00000010: 70000000 00007900 e0025a00 21046000
       00000020: 00000000 00000000 00000000 20000000
       00000030: 00080000 881b7340 00000000 00000000
Board: LS1021ATWR
CPLD:  V2.0
PCBA:  V1.0
VBank: 0
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>reset
```

- Switch to alternate bank:

```
=>boot_bank 1
```

The table below shows a memory map for LS1021A TWR:

Table 37. NOR Flash Memory Map

Start	End Offset	Description	Size
0x60000000	0x6001ffff	bank0 rcw image	128 K
0x600c0000	0x600dffff	bank0 qe	128 K
0x600e0000	0x600fffff	bank0 u-boot env	128 K
0x60100000	0x601fffff	bank0 u-boot Image	1 M
0x60200000	0x602fffff	bank0 DTB	1 M
0x60300000	0x609fffff	bank0 Linux Kernel Image	7 M
0x60a00000	0x63ffffff	bank0 Ramdisk Root File System	54 M
0x64000000	0x6401ffff	bank1 rcw image	128 K
0x640c0000	0x640dffff	bank1 qe	128 K
0x640e0000	0x640fffff	bank1 u-boot env	128 K
0x64100000	0x641fffff	bank1 u-boot Image	1 M
0x64200000	0x642fffff	bank1 DTB	1 M
0x64300000	0x649fffff	bank1 Linux Kernel Image	7 M
0x64a00000	0x67ffffff	bank1 Ramdisk Root File System	54 M

4.4.4.7 Programming a New U-Boot and RCW

The following three sections will discuss how to individually update U-Boot, RCW. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash both two at once, there is no need to switch into the

alternate bank after each configuration, i.e. type the command "boot_bank 1" after U-Boot and RCW have both been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 82000000 <u-boot_file_name>.bin
=>protect off 64100000 +$filesize
=>erase 64100000 +$filesize
=>cp.b 82000000 64100000 $filesize
=>protect on 64100000 +$filesize
=>boot_bank 1
```

The commands above will only program a new U-Boot. Programming a new RCW will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address 0x64000000. 0x64000000 is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 82000000 <rcw_file_name>.bin
=>protect off 64000000 +$filesize
=>erase 64000000 +$filesize
=>cp.b 82000000 64000000 $filesize
=>protect on 64000000 +$filesize
=>boot_bank 1
```

Programming U-boot to SD card

To program U-boot, first boot the board to u-boot. Next, load the new u-boot SD boot image(u-boot-with-spl-pbl.bin) to RAM by downloading it via TFTP and then copying it to SD card with blk offset 0x8. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 81000000 u-boot-with-spl-pbl.bin
=>mmc erase 8 0x800
=>mmc write 0x81000000 8 0x800
```

Program the image to SD card in Linux.

```
dd if=u-boot-with-spl-pbl-sd.bin of=/dev/sdb bs=512 seek=8
```

Programming U-boot and RCW to QSPI

To program a new RCW and u-boot, first switch to QSPI boot mode by performing a hard reset or by typing *reset*. Next, load the new RCW and U-Boot image for QSPI boot to RAM by downloading it via TFTP and then copying it to flash. Execute the following commands at the U-Boot prompt to program the RCW and U-Boot to flash.

```
=> sf probe 0:0
=> tftp 81000000 rcw_qspiboot_1000_swap.bin
=> sf erase 0 0x90000
=> sf write 81000000 0 0x100
=> tftp 82000000 u-boot-qspiboot-swap.bin
=> sf write 82000000 0x10000 0x80000
```

If the original QSPI boot image on board is broken, you can recover it by SD boot image in the BSP release ISO, which supports QSPI.

1. Bring up the board by SD boot mode, you should use the SD boot image in the BSP release ISO;
2. Execute the above commands at the U-Boot prompt to program the RCW and U-Boot to QSPI flash;
3. Switch to QSPI boot mode, then bring up the board.

4.4.4.8 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the NOR Flash Memory Map within Flash Bank Usage as reference for the specific addresses.

4.4.4.8.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters on page 137 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw ramdisk_size=40000000 console=ttyS0,115200'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 83000000 < uImage_name>
   =>tftp 88000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 8f000000 <platform_dtb_name>
   =>bootm 83000000 88000000 8f000000
   ```

   Now the board will boot into Linux using the images generated by Yocto.

4.4.4.8.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   =>saveenv
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_start_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

   ```
   =>tftp 82000000 <uImage name>
   =>protect off <kernel_start_addr> +$filesize
   =>erase <kernel_start_addr> +$filesize
   =>cp.b 82000000 <kernel_start_addr> $filesize
   =>protect on <kernel_start_addr> +$filesize
   ```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

   ```
   =>tftp 88000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>protect off <ramdisk_start_addr> +$filesize
   =>erase <ramdisk_start_addr> +$filesize
   =>cp.b 88000000 <ramdisk_start_addr> $filesize
   =>protect on <ramdisk_start_addr> +$filesize
   ```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

   ```
   =>tftp 8f000000 <platform dtb name>
   =>protect off <dtb_start_addr> +$filesize
   =>erase <dtb_start_addr> +$filesize
   =>cp.b 8f000000 <dtb_start_addr> $filesize
   =>protect on <dtb_start_addr> +$filesize
   ```

5. **Booting Up the System**

   The U-Boot environment variables `fdt_high` and `initrd_high` need to be set to `0xa0000000`.

   ```
   fdt_high=0xa0000000
   initrd_high=0xa0000000
   ```

The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.4.8.3 NFS Deployment

1. **Generating File System with Yocto**

   Use Yocto to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   nfs_root_path board_ipaddress(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/nfs restart
   ```

   NOTE

   nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 137 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>:
   <your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
   =>setenv netdev <ethx>
   =>saveenv
   ```

   NOTE

   <ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

   ```
   =>tftp 83000000 <uImage name>
   =>tftp 8f000000 <platform dtb name>
   =>bootm 83000000 - 8f000000
   ```

   Now the board will boot up with NFS filesystem.

4.4.4.8.4 SD Deployment

The ext2 file system generated by Yocto could be stored in a SD card.

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

   ```
   #fdisk /dev/sdb
   ```

3. Use the mkfs.ext2 command to create the filesystems.

   ```
   # mkfs.vfat /dev/sdb1
   # mkfs.ext2 /dev/sdb2
   ```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

   ```
   # mkdir /temp
   # mount /dev/sdb2 /temp
   # cd /temp
   ```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

   ```
   # mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
   ```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

   ```
   # cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   ```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

   ```
   # cp uImage and uImage.dtb to /temp/boot folder
   # umount /temp
   ```

8. Plug in the SD card to the target board and power on.

4.4.4.8.5 QSPI Deployment

Prepare SD boot to program QSPI flash

1. Build U-Boot image for SD boot(enables QSPI):

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/gcc-linaro-arm-linux-
gnueabihf-4.8-2014.04_linux/bin/arm-linux-gnueabihf-
$make ARCH=arm ${board_name}_sdcard_qspi_defconfig
$make CROSS_COMPILE=${toolchain_path}/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin/arm-
linux-gnueabihf- -j4
```

2. Write the U-Boot image to SD card:

```
=>tftp 0x81000000 u-boot-with-spl-pbl.bin
=>mmc write 0x81000000 8 0x800
```

3. Switch to SD boot(enables QSPI):set switches referring to board configuration document and power on the board from SD boot.

Build U-Boot image for QSPI boot

1. Compile QSPI boot image(enable QSPI):

```
$make distclean ARCH=arm CROSS_COMPILE=${toolchain_path}/gcc-linaro-arm-linux-
gnueabihf-4.8-2014.04_linux/bin/arm-linux-gnueabihf-
$make ARCH=arm  ${board_name}_qspi_defconfig
$make CROSS_COMPILE=${toolchain_path}/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin/arm-
linux-gnueabihf- -j4
```

2. Swap the bytes for RCW:

```
$tclsh byte_swap.tcl rcw_1200_qspiboot.bin rcw_1200_qspiboot_swap.bin 8
```

   The byte_swap.tcl script is a shareable tool and can be found under `rcw/tool/` directory.

3. Write RCW and U-Boot images to QSPI flash under SD boot (enables QSPI) mode:

```
=>sf probe 0:0
```

   SF: Detected N25Q128 with page size 256 Bytes, erase size 64 KiB, total 16 MiB

```
=>tftp 81000000 rcw_1200_qspiboot_swap.bin;sf erase 0 +$filesize;sf write 81000000 0 $filesize
=>tftp 82000000 u-boot-dtb.bin;sf erase 10000 +$filesize;sf write 82000000 10000 $filesize
```

4. Switch to QSPI boot: Set switches referring to board configuration document and power on the board from QSPI boot.

4.4.4.9 Check 'Link Up' for Serial Ethernet Interfaces

If you are experiencing problems with your Ethernet interfaces, this section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
0 - AR8031/AR8033 <--> eTSEC2
1 - AR8031/AR8033 <--> eTSEC3
2 - AR8031/AR8033 <--> eTSEC1
```

The results from the above *mdio list* command show that U-Boot was able to see PHYs on each of the 3 eTSEC interfaces and on the 10GEC interface. If you see “Generic” reported, it is an indication that something is there but the LS1021ATWR can’t communicate with the device/port.

Check Link Status for External PHY

In order to check the status of a SGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read eTSEC1 1
Reading from bus FSL_MDIO0
PHY at address 2:
1 - 0x796d
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7969.

4.4.4.10 Basic Networking Ping Test

In Linux, in order to check that your network driver is set up, follow the commands below:

```
#ip addr show
#ip addr add <ip address of board>/24 brd + dev <port in Linux>
#ip link set <port in Linux> up
#ping <serverip>
```

4.4.4.11 Hardware Setting for Special Purposes

The TWR has user selectable switches for evaluating different pin mux and boot options for the LS1021A device. Information below lists the hardware setting for those special purposes.

General UART port setting for TWR board:

1. J19 and J20 short 2-3.
2. Power on the board first and then connect J5 with USB serial port to host.
3. Wait for auto-install the USB serial port driver on the host.
4. Decompress "Serial_driver.tar".
5. Install "2_mbedWinSerial_16466.exe"(for windows host).

LPUART console setting.

1. J19 and J20 short 1-2.
2. Use rcw: rcw_1200_lpuart.bin.
3. Follow the general UART port setting steps above to install the drivers.

NOR boot setting

1. X2 board: Set SW2[1:8]+SW3[1]=0b'0001_0010_1,X3 board: SW2[1:8]=0b’1000_1111;
2. Bank0: SW3[5]=0b'0; Bank1: SW3[5]=0b'1

SD boot setting

1. image:u-boot-with-spl-pbl.bin
2. Program the image to SD card in u-boot.

```
=>tftp 82000000 u-boot-with-spl-pbl.bin
=>mmc write 82000000 8 800
```

3. Program the image to SD card in Linux.

```
dd if=u-boot-with-spl-pbl-sd.bin of=/dev/sdb bs=512 seek=8
```

4. Insert SD card in adaptor;
5. X2 board: Set SW2[1:8]+SW3[1]=0b'0010_0000_0,X3 board: Set SW2[1:8]=0b’0010’1111.

QSPI boot setting

1. X2 board don't support QSPI boot.
2. X3 board: Set SW2[1:8]=0b’0001’0111.

GPIO test setting

1. GPIO pins 4 to 9 of GPIO controller 2 are tested on board : GPIO2[4:9]
2. Use rcw: rcw_1200_misc.bin
3. eTSEC1 will be disabled under GPIO testing.

2D-ACE test setting

1. Select LPUART0 as the console;
2. Use rcw: rcw_1200_lpuart.bin;

USB 2.0 test setting

1. Select LS1021ATWR X3 board and TWR-SER2;
2. Plug USB device on USB OTG poart;
3. Use rcw: rcw_1000_usb2.bin;
4. Set SW2[6]=0b’0;
5. Set in u-boot: hwconfig=usb1:dr_mode=host,phy_type=ulpi;usb2;

SATA test setting

1. Plug SATA disk on J4 SATA port
2. Use rcw: RSR_PPS_70/rcw_1200.bin;

Secure boot execution setting

1. Short J11.

NOTE

For more information about specific settings for the console (i.e. baud rate, ports used etc.), please refer to the LS1021ATWR Quick Start Guide.

4.4.5 LS1012ARDB

4.4.5.1 Overview

The LS1012A reference design board (RDB) is the high-performance computing, evaluation, and development platform that supports the QorIQ LS1012A processor. This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.5.2 Switch Settings

The RDB has user selectable switches for evaluating different boot options i.e. QSPI Flash 1 for the LS1012A device. Table below lists the default switch settings and the description of these settings.

Table 38. Switch configuration

	1	2	3	4	5	6	7	8
SW1	1	0	1	0	0	1	1	0
SW2	0	0	0	0	0	0	0	0

Below are additional switch settings for alternate boot option i.e QSPI Flash2.

Table 39. Switch configuration

	1	2	3	4	5	6	7	8
SW1	1	0	1	0	0	1	1	0
SW2	0	0	0	0	0	0	1	0

4.4.5.3 U-Boot Environment Variables

The following sections will guide the users on how to set the U-Boot environment and configure the U-Boot network parameters.

4.4.5.3.1 U-Boot Environment Variable "hwconfig"

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set in the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified in the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.The default setting for for "hwconfig" on LS1012ARDB is as follows:

```
hwconfig = fsl_ddr:bank_intlv=auto
```

4.4.5.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD

=>saveenv
```

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

Possible <ethx> value

- pfe_eth0
- pfe_eth1

4.4.5.4 RCW (Reset Configuration Word) and Ethernet Interfaces

The following RCW binary is used on the LS1012ARDB board

`RR_SPNH_3508/PBL_0x35_0x08_800_250_1000_default.bin`

This RCW enables

- Boot from QSPI
- 800MHz Core, 250MHz Platfrom, 1000MT/s DDR
- SDHC1, SDHC2, I2C1,
- SerDes Protocol 0x3508
- PCIe, SATA,
- RGMII, SGMII
- USB 3.0

4.4.5.5 System Memory Map

In 64-bit u-boot, there is a 1:1 mapping of physical address and effective address. After system startup, the boot loader maps physical address and effective address as shown in the following table:

Start Physical Address	End Physical Address	Memory Type	Size
0x00_0000_0000	0x00_00FF_FFFF	Secure Boot ROM	1MB
0x00_0100_0000	0x00_0FFF_FFFF	CCSR	240MB
0x00_1000_0000	0x00_1000_FFFF	OCRAM1	64KB
0x00_1001_0000	0x00_1001_FFFF	OCRAM2	64 KB
0x00_4000_0000	0x00_5FFF_FFFF	QSPI	512MB
0x00_8000_0000	0x00_FFFF_FFFF	DRAM	2GB
0x08_8000_0000	0x0F_FFFF_FFFF	DRAM2	30G
0x40_0000_0000	0x47_FFFF_FFFF	PCI Express1	32G

4.4.5.6 Flash Bank Usage

LS1012ARDB has 2 QSPI flash connected over QSPI contoller.

Only one QSPI flash is available at a time depending upon the board switch settings. These switch settings can also be overriden by I2C commands.

To protect the default U-Boot in flash1 (aka bank1), it is a convention employed by NXP to deploy work images into the flash2 (aka bank2), and then switch to the flash2 (aka bank2) for testing. Switching to the flash2 (aka bank2) can be done in software using I2C commands and effectively swaps the flash1 (aka bank1) with the flash2 (aka bank2). This protects flash1 and keeps the board bootable under all circumstances.

```
U-Boot 2016.01-g5ab5bba (Jun 14 2016 - 12:56:17 +0530)

SoC:  LS1012AE (0x87040010)
Clock Configuration:
       CPU0(A53):800  MHz
       Bus:      250  MHz  DDR:      1000 MT/s
Reset Configuration Word (RCW):
       00000000: 08000008 00000000 00000000 00000000
       00000010: 35080000 c000000c 40000000 00001800
       00000020: 00000000 00000000 00000000 00014571
       00000030: 00000000 18c2a120 00000096 00000000
I2C:   ready
DRAM:  1022 MiB
MMU warning: gd->secure_ram is not maintained, disabled.
SEC: RNG instantiated
Using SERDES1 Protocol: 13576 (0x3508)
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
SF: Detected S25FS512S_256K with page size 512 Bytes, erase size 128 KiB, total 64 MiB
PCIe1: Root Complex no link, regs @ 0x3400000
In:    serial
Out:   serial
Err:   serial
Model: LS1012A RDB Board
Board: LS1012ARDB Version: RevB, boot from QSPI: bank1
SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI:  Net:   cbus_baseaddr: 0000000004000000, ddr_baseaddr: 0000000083800000, ddr_phys_baseaddr:
```

```
03800000
class init complete
tmu init complete
bmu1 init: done
bmu2 init: done
GPI1 init complete
GPI2 init complete
HGPI init complete
hif_tx_desc_init: Tx desc_base: 0000000083e40400, base_pa: 03e40400, desc_count: 64
hif_rx_desc_init: Rx desc base: 0000000083e40000, base_pa: 03e40000, desc_count: 64
HIF tx desc: base_va: 0000000083e40400, base_pa: 03e40400
HIF init complete
bmu1 enabled
bmu2 enabled
pfe_hw_init: done
pfe_firmware_init
pfe_load_elf: no of sections: 13
pfe_firmware_init: class firmware loaded
pfe_load_elf: no of sections: 10
pfe_firmware_init: tmu firmware loaded
ls1012a_configure_serdes 0
pfe_eth0, pfe_eth1
Hit any key to stop autoboot:  0
=>
```

How to boot from flash 2 (aka bank2)

1. To check which bank booted, refer to "**Board: LS1012ARDB Version: RevB, boot from QSPI: bank1"** in the U-Boot logs.
2. I2C commands to flash2 bank2 switch **" i2c mw 0x24 0x7 0xfc; i2c mw 0x24 0x3 0xf5 "**
3. Program QSPI flash as per flash layout
4. To boot from bank2 give "**reset"** command.
5. To move back to bank 1 from bank 2, power on/off the board or use **"i2c mw 0x24 0x3 0xf4 "** and then give "**reset**" command.

QSPI flash Layout

Image	Size	Start Address
RCW + PBI	1MB	0x4000_0000
U-boot boot loader + PFE binary	1MB	0x4010_0000
U-boot Env	1MB	0x4020_0000
PPA FIT image	2MB	0x4050_0000
Kernel ITB	59MB	0x40A0_0000

4.4.5.7 Programming a New U-Boot, RCW, and PPA Firmware

The following sections will discuss how to individually update U-Boot and RCW. For specific addresses, please refer to the QSPI Flash Memory Map as a reference.

Please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot. By default, an existing U-Boot is run in flash1 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate flash/bank, first switch to flash 1 (bank 0) by performing a hard reset or by typing `reset`. Then use the following commands to change flash2/bank1 and program a new U-Boot and then switch to that flash2 or alternate bank where the new U-Boot is flashed:

```
=>i2c mw 0x24 0x7 0xfc; i2c mw 0x24 0x3 0xf5
=>tftp 0x80000000 <u-boot_file_name>.bin
=>sf probe 0:0
=>sf erase 0x100000 +$filesize
=>sf write 0x80000000 0x100000 $filesize
=> reset
```

Programming a New RCW: To program a new RCW, first switch to flash1 (bank 0) by performing a hard reset or by typing `reset`. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash2 at offset 0x000000. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>i2c mw 0x24 0x7 0xfc; i2c mw 0x24 0x3 0xf5
=>tftp 0x80000000 <rcw_file_name>.bin
=>sf probe 0:0
=>sf erase 0x0 +$filesize
=>sf write 0x80000000 0x0 $filesize
=> reset
```

NOTE

RCW and U-Boot binary must be byte swapped using command (tclsh ./byte_swap.tcl u-boot-dtb.bin u-boot-dtb_swap.bin 8)

Programming a New Primary Protected Application (PPA) Firmware: To program a new PPA firmware, first switch to bank 0 by performing a hard reset or by typing *reset*. Next, load the new PPA firmware to RAM by downloading it via TFTP and then copying it to flash at address `0x500000`. `0x500000` is the location of PPA firmware in the alternate bank. Then execute the following commands at the U-Boot prompt to program the PPA firmware to flash and reset to alternate bank.

```
=> tftp 0x80000000 <ppa_file_name>.itb
=> sf probe 0:0
=> sf erase 0x500000 +$filesize
=> sf write 0x80000000 0x500000 $filesize
=> reset
```

NOTE

All the files which are to be flashed need to be byte-swapped.

4.4.5.8 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.5.8.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

```
=>setenv bootargs 'ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500'
=>saveenv
```

NOTE

ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

```
=>tftp 0xa0000000 <kernel_itb_name>
=>bootm 0xa0000000
```

   Now the board will boot into Linux using the images generated by Yocto.

4.4.5.8.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

```
=>setenv bootcmd 'run ramargs; pfe stop; sf probe 0:0; sf read 0x96000000 0xa00000 0x2800000;
bootm 0x96000000
```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to QSPI Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 0x96000000 <kernel ITB>
=>sf probe 0:0;
=>sf erase <kernel_itb_addr> +$filesize
=>sf write 0x96000000 <kernel_itb_addr> $filesize
```

3. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> run ramargs; pfe stop; sf probe 0:0; sf read 0x96000000 0xa00000 0x2800000; bootm 0x96000000
```

4.4.5.8.3 NFS Deployment

1. **Generating File System with Yocto**

   Use Yocto to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   <nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/nfs restart
   ```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 164 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>:
   <your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
   =>setenv netdev <ethx>
   =>saveenv
   ```

NOTE

<ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

   ```
   =>tftp 96000000  kernel.itb;
   =>bootm 96000000:kernel@1 - 96000000:fdt@1
   ```

   Now the board will boot up with NFS filesystem.

4.4.5.8.4 SD Deployment

Partition SD Card

1. Insert SD card into the Linux Host PC.
2. Use the "fdisk" command to repartition the SD card.

   ```
   # fdisk /dev/sdb
   ```

3. Use the mkfs.ext2 command to creat the filesystem.

   ```
   #mkfs.ext2 /dev/sdb1
   ```

NOTE

When connecting the SD card on Linux host, look for the log messages on the Linux console and accordingly, only format that device (which represents SD card).

1. Insert SD card into the Linux Host PC.
2. Create temp director in host PC and mount the ext2 partition to the temp

```
#mkdir temp
#mount /dev/sdb1 temp
```

3. Copy the FIT Kernel Image to the SD card partition.

```
#cp  kernel.itb temp/
```

4. Copy the Root File System to the SD card partition.

```
#cp fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz  temp/
#tar xvfz fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz
#rm  fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz  temp/
```

5. Umount the temp director

```
#umount temp
```

Setting U-Boot Environment

- Execute the following commands at the U-Boot prompt

```
=> setenv bootcmd "ext2load mmc 0 a0000000 kernel.itb && bootm a0000000"
```

- Using the Ramdisk as the Root File Systemp

```
=> setenv bootargs  "root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500
console=ttyS0,115200"
```

- Using the Ext2 Partition of SD card as the Root File Systemp

```
=> setenv bootargs  "root=/dev/mmcblk0p1 rw earlycon=uart8250,0x21c0500
console=ttyS0,115200"
```

- Saving the environment

```
=>saveenv
```

Note: The `kernel.itb` is the name of your FIT Image, you can use the `ext2ls` command to list it at the U-Boot prompt

4.4.5.9 Check 'Link Up' for Serial Ethernet Interfaces

This section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors when experiencing problems with Ethernet interfaces.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command `mdio list`. The U-Boot command mdio list will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
PFE_MDIO:
1 - RealTek RTL8211F <--> pfe_eth1
2 - RealTek RTL8211F <--> pfe_eth0
```

The results from the `mdio list` command above show that U-Boot was able to see PHYs on each of the RGMII/SGMII interfaces.

Check Link Status for External PHY

In order to check the status of a RGMII/SGMII link, use the mdio read command. Since this is a Clause 22 device, we pass two arguments to the `mdio read` command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read pfe_eth0 1
Reading from bus PFE_MDIO
PHY at address 2:
1 - 0x79ad
=> mdio read pfe_eth1 1
Reading from bus PFE_MDIO
PHY at address 1:
1 - 0x79ad
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the example above, the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7989.

4.4.5.10 Basic Networking Ping Test

U-BOOT

The LS1012ARDB has one SGMII and one RGMII. The log below shows how to ping from those 2 interfaces.

```
U-Boot 2016.01-gc423721 (Jun 27 2016 - 13:06:22 +0530)

SoC:  LS1012AE (0x87040010)
Clock Configuration:
       CPU0(A53):800  MHz
       Bus:      250  MHz  DDR:      1000 MT/s
Reset Configuration Word (RCW):
       00000000: 08000008 00000000 00000000 00000000
       00000010: 35080000 c000000c 40000000 00001800
       00000020: 00000000 00000000 00000000 00014571
       00000030: 00000000 18c2a120 00000096 00000000
I2C:   ready
DRAM:  1022 MiB
MMU warning: gd->secure_ram is not maintained, disabled.
SEC: RNG instantiated
Using SERDES1 Protocol: 13576 (0x3508)
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
SF: Detected S25FS512S_256K with page size 512 Bytes, erase size 128 KiB, total 64 MiB
PCIe1: Root Complex no link, regs @ 0x3400000
In:    serial
Out:   serial
Err:   serial
Model: LS1012A RDB Board
Board: LS1012ARDB Version: RevB, boot from QSPI: bank1
SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI:  Net:   cbus_baseaddr: 0000000004000000, ddr_baseaddr: 0000000083800000, ddr_phys_baseaddr:
03800000
class init complete
```

```
tmu init complete
bmu1 init: done
bmu2 init: done
GPI1 init complete
GPI2 init complete
HGPI init complete
hif_tx_desc_init: Tx desc_base: 0000000083e40400, base_pa: 03e40400, desc_count: 64
hif_rx_desc_init: Rx desc base: 0000000083e40000, base_pa: 03e40000, desc_count: 64
HIF tx desc: base_va: 0000000083e40400, base_pa: 03e40400
HIF init complete
bmu1 enabled
bmu2 enabled
pfe_hw_init: done
pfe_firmware_init
pfe_load_elf: no of sections: 13
pfe_firmware_init: class firmware loaded
pfe_load_elf: no of sections: 10
pfe_firmware_init: tmu firmware loaded
ls1012a_configure_serdes 0
pfe_eth0, pfe_eth1
Hit any key to stop autoboot:  0
=>  setenv ethaddr 11:22:33:44:55:55
=> setenv eth1addr 11:22:33:44:55:66
=> setenv serverip 192.168.1.1;setenv ipaddr 192.168.1.136
=> ping $serverip
Speed detected 3e8
Using pfe_eth0 device
host 192.168.1.1 is alive
=> edit ethact
edit: pfe_eth1
=> ping $serverip
Speed detected 3e8
Using pfe_eth1 device
host 192.168.1.1 is alive
```

LINUX

To enable PFE in Linux, first stop PFE in U-Boot. In order to do this, first bring the kernel-ls1012a-rdb.itb via PFE interface then type `pfe stop` command on the U-Boot prompt:

```
=> tftp 0xa0000000  kernel-ls1012a-rdb.itb
Speed detected 3e8
Using pfe_eth1 device
TFTP from server 192.168.1.1; our IP address is 192.168.1.136
Filename 'kernel-ls1012a-rdb.itb'.
Load address: 0xa0000000
Loading: ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
```

```
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ############
         5.6 MiB/s
done
Bytes transferred = 38342491 (2490f5b hex)
=> pfe stop
Stopping PFE...
=> bootm 0xa0000000
## Loading kernel from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'kernel@1' kernel subimage
     Description:  ARM64 Linux kernel
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0xa00000dc
     Data Size:    12482048 Bytes = 11.9 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x80080000
     Entry Point:  0x80080000
## Loading ramdisk from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'ramdisk@1' ramdisk subimage
     Description:  LS2 Ramdisk
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0xa0be9ba4
     Data Size:    25849963 Bytes = 24.7 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: unavailable
     Entry Point:  unavailable
## Loading fdt from FIT Image at a0000000 ...
   Using 'config@1' configuration
```

```
   Trying 'fdt@1' fdt subimage
     Description:  Flattened Device Tree blob
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0xa0be7790
     Data Size:    9101 Bytes = 8.9 KiB
     Architecture: AArch64
   Loading fdt from 0xa0be7790 to 0x90000000
   Booting using the fdt blob at 0x90000000
   Loading Kernel Image ... OK
   Using Device Tree in place at 0000000090000000, end 000000009000538c
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 4.1.8-rt8+g2511ec0 (jenkins@neptune) (gcc version 4.9.4 20150629
(prerelease) (Linaro GCC 4.9-2015.06) ) #1 SMP Sat Aug 27 04:44:19 CST 2016
[    0.000000] CPU: AArch64 Processor [410fd034] revision 4
[    0.000000] Detected VIPT I-cache on CPU0
[    0.000000] alternatives: enabling workaround for ARM erratum 845719
[    0.000000] earlycon: Early serial console at MMIO 0x21c0500 (options '')
[    0.000000] bootconsole [uart0] enabled
[    0.046613] No BMan portals available!
[    0.052448] No QMan portals available!
[    0.186759] Freescale FM module, FMD API version 21.1.0
[    0.192162] Freescale FM Ports module
[    0.200605] vfio_fsl_mc_driver_init: Driver registration fails as no fsl_mc_bus found
[    0.658570] fsl-mc bus not found, restool driver registration failed
[    0.927488] usb usb1-port1: over-current condition
[    0.932320] usb usb2-port1: over-current condition
INIT: version 2.88 booting
Starting udev
[    3.038179] pe_load_ddr_section: load address(3fb0000) and elf file address(ffff0000003fb000)
rcvd
Populating dev cache
hwclock: can't open '/dev/misc/rtc': No such file or directory
Fri Aug 26 20:58:57 UTC 2016
hwclock: can't open '/dev/misc/rtc': No such file or directory
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
Running postinst /etc/rpm-postinsts/101-inetutils-inetd...
Running postinst /etc/rpm-postinsts/102-inetutils-ftpd...
update-rc.d: /etc/init.d/run-postinsts exists during rc.d purge (continuing)
Removing any system startup links for run-postinsts ...
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting system log daemon...0
Starting kernel log daemon...0
Starting internet superserver: xinetd.

QorIQ SDK (FSL Reference Distro) 2.0 ls1012ardb /dev/ttyS0

ls1012ardb login: root
root@ls1012ardb:~# find / -name pfe.ko
/lib/modules/4.1.8+g4b2f599/kernel/drivers/staging/fsl_ppfe/pfe.ko
root@ls1012ardb:~# insmod /lib/modules/4.1.8+g4b2f599/kernel/drivers/staging/fsl_ppfe/pfe.ko
[   39.882345] pfe: module is from the staging directory, the quality is unknown, you have been
warned.
[   39.895444] cbus_baseaddr: ffff000000880000, ddr_baseaddr: ffff800003400000, ddr_phys_baseaddr:
83400000, ddr_size: c00000
[   39.907048] pfe_hw_init
```

```
[   39.909498] CLASS version: 20
[   39.912466] TMU version: 1011231
[   39.916024] BMU1 version: 21
[   39.918907] BMU2 version: 21
[   39.921787] EGPI1 version: 50
[   39.924754] EGPI2 version: 50
[   39.928157] HGPI version: 50
[   39.931041] GPT version: 0
[   39.933747] HIF version: 10
[   39.936851] HIF NOPCY version: 10
[   39.940171] bmu_init(1) done
[   39.943053] bmu_init(2) done
[   39.948670] class_init() done
[   39.961678] tmu_init() done
[   39.964475] gpi_init(1) done
[   39.967585] gpi_init(2) done
[   39.970469] gpi_init(hif) done
[   39.973523] bmu_enable(1) done
[   39.976899] bmu_enable(2) done
[   39.979957] pfe_hif_lib_init
[   39.983049] pfe_hif_init
[   39.985582] pfe_hif_alloc_descr
[   39.989399] pfe_hif_init_buffers
[   39.992789] pfe_firmware_init
[   39.996492] pfe_load_elf
[   39.999043] pe_load_ddr_section: load address(3fb0000) and elf file address(ffff00000050b000)
rcvd
[   40.032785] PFE binary version: pfe_ls1012a_00_1-dirty
[   40.037965] pfe_firmware_init: class firmware loaded 0xa60 0xc3010000
[   40.044415] pfe_load_elf
[   40.048328] pfe_firmware_init: tmu firmware loaded 0x200
[   40.053668] pfe_ctrl_init
[   40.056631] pfe_ctrl_init finished
[   40.060039] pfe_eth_init
[   40.062614] pfe_eth_mdio_init
[   40.066072] pfe_ctrl_timer
[   40.076155] libphy: Comcerto MDIO Bus: probed
[   40.082466] pfe_phy_init interface 3
[   40.175983] eth0: pfe_eth_init_one: created interface, baseaddr: ffff000000a80000
[   40.192945] pfe_phy_init interface 7
[   40.276066] eth1: pfe_eth_init_one: created interface, baseaddr: ffff000000aa0000
[   40.283643] pfe_debugfs_init
root@ls1012ardb:~# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:1a:2b:3c:4d:5e
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 00:aa:bb:cc:dd:ee
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
```

```
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

sit0      Link encap:UNSPEC  HWaddr 00-00-00-00-3A-30-30-30-00-00-00-00-00-00-00-00is insmod a
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@ls1012ardb:~# ifconfig eth1 192.168.1.34 up
[   62.478999] eth1: pfe_eth_open
[   62.482403] hif_process_client_req: register client_id 1
[   62.487772] pfe_hif_client_register
[   62.491755] eth1: pfe_gemac_init
root@ls1012ardb:~# [   67.275999] eth1: Link is Up - 1Gbps/Full - flow control rx/tx

root@ls1012ardb:~#
root@ls1012ardb:~#
root@ls1012ardb:~#
root@ls1012ardb:~# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=128 time=1.81 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=128 time=0.883 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.883/1.347/1.811/0.464 ms
root@ls1012ardb:~#
```

NOTE

Yocto users do not need to install the pfe.ko module. However, developers who are building the kernel need to install the pfe.ko module. Once the pfe.ko module is installed, the pfe interfaces will function like normal ethernet interfaces e.g. eth0, eth1.

4.4.6 FRDM-LS1012A

4.4.6.1 Switch Settings

No On-Board Switch setting available

4.4.6.2 U-Boot Environment Variables

The following sections will guide the users on how to set the U-Boot environment and configure the U-Boot network parameters.

4.4.6.2.1 U-Boot Environment Variable "hwconfig"

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set in the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified in the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.The default setting for for "hwconfig" on LS1012ARDB is as follows:

```
hwconfig = fsl_ddr:bank_intlv=auto
```

4.4.6.2.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD

=>saveenv
```

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

Possible <ethx> value

- pfe_eth0
- pfe_eth1

4.4.6.3 RCW (Reset Configuration Word) and Ethernet Interfaces

The following RCW binary is used on the FRDM-LS1012A board:

```
S_SPNN_3305/PBL_0x33_0x05_800_250_1000_default.bin
```

This RCW enables:

- Boot from QSPI
- 800MHz Core, 250MHz Platfrom, 1000MT/s DDR
- I2C1,
- SerDes Protocol 0x3505

- SGMII, SGMII
- USB 3.0

4.4.6.4 System Memory Map

In 64-bit u-boot, there is a 1:1 mapping of physical address and effective address. After system startup, the boot loader maps physical address and effective address as shown in the following table:

Start Physical Address	End Physical Address	Memory Type	Size
0x00_0000_0000	0x00_00FF_FFFF	Secure Boot ROM	1MB
0x00_0100_0000	0x00_0FFF_FFFF	CCSR	240MB
0x00_1000_0000	0x00_1000_FFFF	OCRAM1	64KB
0x00_1001_0000	0x00_1001_FFFF	OCRAM2	64 KB
0x00_4000_0000	0x00_5FFF_FFFF	QSPI	512MB
0x00_8000_0000	0x00_FFFF_FFFF	DRAM	2GB
0x08_8000_0000	0x0F_FFFF_FFFF	DRAM2	30G
0x40_0000_0000	0x47_FFFF_FFFF	PCI Express1	32G

4.4.6.5 Flash Bank Usage

The FRDM-LS1012A only has one QSPI flash connected over QSPI contoller.

```
U-Boot 2016.01-g1c18d6a (Jun 17 2016 - 15:15:39 +0530)

SoC:  LS1012AE (0x87040010)
Clock Configuration:
       CPU0(A53):800  MHz
       Bus:      250  MHz  DDR:      1000 MT/s
Reset Configuration Word (RCW):
       00000000: 08000008 00000000 00000000 00000000
       00000010: 33050000 c000000c 40000000 00001800
       00000020: 00000000 00000000 00000000 000c4571
       00000030: 00000000 00c28120 00000096 00000000
I2C:   ready
DRAM:  510 MiB
MMU warning: gd->secure_ram is not maintained, disabled.
Using SERDES1 Protocol: 13061 (0x3305)
SF: Detected S25FS512S_256K with page size 512 Bytes, erase size 128 KiB, total 64 MiB
In:    serial
Out:   serial
Err:   serial
Model: LS1012A FREEDOM Board
Board: LS1012AFRDM Net:   cbus_baseaddr: 0000000004000000, ddr_baseaddr: 0000000083800000,
ddr_phys_baseaddr: 03800000
class init complete
tmu init complete
bmu1 init: done
bmu2 init: done
```

```
GPI1 init complete
GPI2 init complete
HGPI init complete
hif_tx_desc_init: Tx desc_base: 0000000083e40400, base_pa: 03e40400, desc_count: 64
hif_rx_desc_init: Rx desc base: 0000000083e40000, base_pa: 03e40000, desc_count: 64
HIF tx desc: base_va: 0000000083e40400, base_pa: 03e40400
HIF init complete
bmu1 enabled
bmu2 enabled
pfe_hw_init: done
pfe_firmware_init
pfe_load_elf: no of sections: 13
pfe_firmware_init: class firmware loaded
pfe_load_elf: no of sections: 10
pfe_firmware_init: tmu firmware loaded
ls1012a_configure_serdes 0
ls1012a_configure_serdes 1
pfe_eth0, pfe_eth1
Hit any key to stop autoboot:  0
```

QSPI flash Layout

Image	Size	Start Address
RCW + PBI	1MB	0x4000_0000
U-boot boot loader + PPFE binary	1MB	0x4010_0000
U-boot Env	1MB	0x4020_0000
PPA FIT image	2MB	0x4050_0000
Kernel ITB	59MB	0x40A0_0000

4.4.6.6 Programming a New U-Boot and RCW

The following sections will discuss how to individually update U-Boot and RCW. For specific addresses, please refer to the QSPI Flash Memory Map as a reference.

Please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot. Use the commands below to update images:

```
=>tftp 0x80000000 <u-boot_file_name>.bin
=>sf probe 0:0
=>sf erase 0x100000 0xa0000
=>sf write 0x80000000 0x100000 $filesize
=> reset
```

The commands above will only program a new U-Boot. Programming a new RCW will be discussed in the next section.

Programming a New RCW: Use the commands below to update images:

```
=>tftp 0x80000000 <rcw_file_name>.bin
=>sf probe 0:0
=>sf erase 0x0 40000
=>sf write 0x80000000 0x0 $filesize
=> reset
```

Note: RCW and U-Boot binaries must be byte swapped using command (tclsh ./byte_swap.tcl u-boot-dtb.bin u-boot-dtb_swap.bin 8). The ./byte_swap.tcl script is included as a part od the RCW source code.

4.4.6.7 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.6.7.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 0x96000000 <kernel_itb_name>
   =>bootm 0x96000000
   ```

   NOTE

   The DDR load address for LS1012ARDB is different from FRDM-LS1012A. Using the wrong address could crash Linux.

   Now the board will boot into Linux using the images generated by Yocto.

4.4.6.7.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv bootcmd 'run ramargs; pfe stop; sf probe 0:0; sf read 0x96000000 0xa00000 0x2800000;
   bootm 0x96000000
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to QSPI Flash**

The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 0x96000000 <kernel ITB>
=>sf probe 0:0;
=>sf erase <kernel_itb_addr> +$filesize
=>sf write 0x96000000 <kernel_itb_addr> $filesize
```

3. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

   ```
   =>boot
   ```

   or

   ```
   => run ramargs; pfe stop; sf probe 0:0; sf read 0x96000000 0xa00000 0x2800000; bootm 0x96000000
   ```

4.4.6.7.3 NFS Deployment

1. **Generating File System with Yocto**

   Use Yocto to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   <nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/nfs restart
   ```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 164 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>:
   <your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
   =>setenv netdev <ethx>
   =>saveenv
   ```

NOTE

<ethx> is the port connected on the Linux boot network.

Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

```
=>tftp 96000000  kernel.itb;
=>bootm 96000000:kernel@1 - 96000000:fdt@1
```

   Now the board will boot up with NFS filesystem.

4.4.6.8 Check 'Link Up' for Serial Ethernet Interfaces

This section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors when experiencing problems with Ethernet interfaces.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command `mdio list`. The U-Boot command `mdio list` will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
PFE_MDIO:
1 - RealTek RTL8211F <--> pfe_eth1
2 - RealTek RTL8211F <--> pfe_eth0
```

The results from the `mdio list` command above show that U-Boot was able to see PHYs on each of the SGMII interfaces.

Check Link Status for External PHY

In order to check the status of a SGMII link, use the `mdio read` command. Since this is a Clause 22 device, we pass two arguments to the `mdio read` command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read pfe_eth0 1
Reading from bus PFE_MDIO
PHY at address 2:
1 - 0x79ad
=> mdio read pfe_eth1 1
Reading from bus PFE_MDIO
PHY at address 1:
1 - 0x79ad
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7989.

4.4.6.9 Basic Networking Ping Test

U-BOOT

The LS1012FRDM has 2 SGMII interfaces. The log below shows how to ping from those 2 interfaces.

```
U-Boot 2016.01-g1c18d6a (Jun 17 2016 - 15:15:39 +0530)
SoC: LS1012AE (0x87040010)
Clock Configuration:
CPU0(A53):800 MHz
Bus: 250 MHz DDR: 1000 MT/s
Reset Configuration Word (RCW):
00000000: 08000008 00000000 00000000 00000000
00000010: 33050000 c000000c 40000000 00001800
00000020: 00000000 00000000 00000000 000c4571
00000030: 00000000 00c28120 00000096 00000000
I2C: ready
DRAM: 510 MiB
MMU warning: gd->secure_ram is not maintained, disabled.
Using SERDES1 Protocol: 13061 (0x3305)
SF: Detected S25FS512S_256K with page size 512 Bytes, erase size 128 KiB, total 64 MiB
In: serial
Out: serial
Err: serial
Model: LS1012A FREEDOM Board
Board: LS1012AFRDM Net: cbus_baseaddr: 0000000004000000, ddr_baseaddr:0000000083800000,
ddr_phys_baseaddr: 03800000
class init complete
tmu init complete
bmu1 init: done
bmu2 init: done
GPI1 init complete
GPI2 init complete
HGPI init complete
hif_tx_desc_init: Tx desc_base: 0000000083e40400, base_pa: 03e40400, desc_count: 64
hif_rx_desc_init: Rx desc base: 0000000083e40000, base_pa: 03e40000, desc_count: 64
HIF tx desc: base_va: 0000000083e40400, base_pa: 03e40400
HIF init complete
bmu1 enabled
bmu2 enabled
pfe_hw_init: done
pfe_firmware_init
pfe_load_elf: no of sections: 13
pfe_firmware_init: class firmware loaded
pfe_load_elf: no of sections: 10
pfe_firmware_init: tmu firmware loaded
ls1012a_configure_serdes 0
ls1012a_configure_serdes 1
pfe_eth0, pfe_eth1
Hit any key to stop autoboot: 0
=> setenv ethaddr 11:22:33:44:55:55
=> setenv eth1addr 11:22:33:44:55:66
=> setenv serverip 192.168.5.124;setenv ipaddr 192.168.5.136
=> edit ethact
edit: pfe_eth1
=> ping $serverip
Speed detected 3e8
Using pfe_eth1 device
host 192.168.5.124 is alive
```

```
=> edit ethact
edit: pfe_eth0
=> ping $serverip
Speed detected 3e8
Using pfe_eth0 device
host 192.168.5.124 is alive
=>
```

LINUX

To enable PFE in Linux, First we need to stop pfe in u-boot but first bring

the kernel-ls1012a-frdm.itb via pfe interface, then perform pfe stop

On U-Boot prompt:

```
=> tftp 0x96000000 kernel-ls1012a-frdm.itb
Speed detected 3e8
Using pfe_eth0 device
TFTP from server 192.168.5.124; our IP address is 192.168.5.125
Filename 'kernel-ls1012a-frdm.itb'.
Load address: 0x96000000
Loading:
#################################################################
#################################################################
#################################################################
#################################################################
#################################################################
#################################################################
#################################################################
#################################################################
#################################################################
#################################################################
#################################################################
#################################################################
2.5 MiB/s
done
Bytes transferred = 40631911 (26bfe67 hex)
=> pfe stop
Stopping PFE...
=> bootm 0x96000000
## Loading kernel from FIT Image at 96000000 ...
   Using 'config@1' configuration
   Trying 'kernel@1' kernel subimage
     Description:  ARM64 Linux kernel
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0x9600013c
     Data Size:    12502016 Bytes = 11.9 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x80080000
     Entry Point:  0x80080000
## Loading ramdisk from FIT Image at 96000000 ...
   Using 'config@1' configuration
   Trying 'ramdisk@1' ramdisk subimage
     Description:  Ramdisk
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0x96bef354
```

```
     Data Size:    24491924 Bytes = 23.4 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: unavailable
     Entry Point:  unavailable
## Loading fdt from FIT Image at 96000000 ...
   Using 'config@1' configuration
   Trying 'fdt@1' fdt subimage
     Description:  Flattened Device Tree blob
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x96bec5f0
     Data Size:    11490 Bytes = 11.2 KiB
     Architecture: AArch64
   Loading fdt from 0x96bec5f0 to 0x90000000
   Booting using the fdt blob at 0x90000000
   Loading Kernel Image ... OK
   Using Device Tree in place at 0000000090000000, end 0000000090005ce1

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 4.1.8-rt8+g2511ec0 (jenkins@neptune) (gcc version 4.9.4 20150629
(prerelease) (Linaro GCC 4.9-2015.06) ) #1 SMP Sat Aug 27 04:44:19 CST 2016
[    0.000000] CPU: AArch64 Processor [410fd034] revision 4
[    0.000000] Detected VIPT I-cache on CPU0
[    0.000000] alternatives: enabling workaround for ARM erratum 845719
[    0.000000] earlycon: Early serial console at MMIO 0x21c0500 (options '')
[    0.000000] bootconsole [uart0] enabled
[    0.046613] No BMan portals available!
[    0.052448] No QMan portals available!
[    0.186759] Freescale FM module, FMD API version 21.1.0
[    0.192162] Freescale FM Ports module
[    0.200605] vfio_fsl_mc_driver_init: Driver registration fails as no fsl_mc_bus found
[    0.658570] fsl-mc bus not found, restool driver registration failed
[    0.927488] usb usb1-port1: over-current condition
[    0.932320] usb usb2-port1: over-current condition
INIT: version 2.88 booting
Starting udev
[    3.038179] pe_load_ddr_section: load address(3fb0000) and elf file address(ffff0000003fb000)
rcvd
Populating dev cache
hwclock: can't open '/dev/misc/rtc': No such file or directory
Fri Aug 26 20:58:57 UTC 2016
hwclock: can't open '/dev/misc/rtc': No such file or directory
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
Running postinst /etc/rpm-postinsts/101-inetutils-inetd...
Running postinst /etc/rpm-postinsts/102-inetutils-ftpd...
update-rc.d: /etc/init.d/run-postinsts exists during rc.d purge (continuing)
Removing any system startup links for run-postinsts ...
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting system log daemon...0
Starting kernel log daemon...0
Starting internet superserver: xinetd.

QorIQ SDK (FSL Reference Distro) 2.0 ls1012afrdm /dev/ttyS0

ls1012afrdm login:
```

```
root@ls1012afrdm:~#
root@ls1012afrdm:~# [ 48.955650] eth0: Link is Up - 1Gbps/Full - flow control rx/tx
root@ls1012afrdm:~# ping 192.168.5.124
PING 192.168.5.124 (192.168.5.124) 56(84) bytes of data.
64 bytes from 192.168.5.124: icmp_seq=1 ttl=128 time=1.92 ms
64 bytes from 192.168.5.124: icmp_seq=2 ttl=128 time=1.51 ms
64 bytes from 192.168.5.124: icmp_seq=3 ttl=128 time=1.44 ms
^C
--- 192.168.5.124 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 1.440/1.625/1.925/0.214 ms
root@ls1012afrdm:~#
root@ls1012afrdm:~# ifconfig -a
eth0 Link encap:Ethernet HWaddr 00:1a:2b:3c:4d:5e
inet addr:192.168.5.125 Bcast:192.168.5.255 Mask:255.255.255.0
inet6 addr: fe80::21a:2bff:fe3c:4d5e/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:39 errors:0 dropped:0 overruns:0 frame:0
TX packets:11 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:9165 (8.9 KiB) TX bytes:894 (894.0 B)
eth1 Link encap:Ethernet HWaddr 00:aa:bb:cc:dd:ee
BROADCAST MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
sit0 Link encap:UNSPEC HWaddr 00-00-00-00-3A-30-30-30-00-00-00-00-00-00-00-00
NOARP MTU:1480 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
root@ls1012afrdm:~#
```

NOTE

Yocto users do not need to install the pfe.ko module. However, developers who are building the kernel need to install the pfe.ko module. Once the pfe.ko module is installed, the pfe interfaces will function like normal ethernet interfaces e.g. eth0, eth1.

4.4.7 LS1043ARDB

4.4.7.1 Overview

The LS1043A reference design board (RDB) is the high-performance computing, evaluation, and development platform that supports the QorIQ LS1043A processor.

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.7.2 Switch Settings

The RDB has user selectable switches for evaluating different boot options and different configurations for the LS1043A device. Table 40. Default Switch Settings on page 174 lists the default switch settings. See the QorIQ LS1043A Reference Design Board Reference Manual for a description of the switch settings.

Table 40. Default Switch Settings

	1	2	3	4	5	6	7	8
SW3	ON [1]	OFF [0]	ON [1]	ON [1]	OFF [0]	OFF [0]	ON [1]	ON [1]
SW4	OFF	OFF	OFF	ON	OFF	OFF	ON	OFF
SW5	ON	OFF	ON	OFF	OFF	OFF	OFF	OFF

The default switch settings select NOR Flash as the boot device. Table 41. LS1043ARDB Switch Settings on page 174 lists additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 41. LS1043ARDB Switch Settings

Boot Source	Switch
NOR (bank0)	SW4[1-8] +SW5[1] = 0b'00010010_1 SW5[4-6]= 0b'000
NOR (bank4)	SW4[1-8] +SW5[1] = 0b'00010010_1 SW5[4-6]= 0b'001
NAND	SW4[1-8] +SW5[1] = 0b'10000011_0
SD	SW4[1-8] +SW5[1] = 0b'00100000_0

4.4.7.3 U-Boot Environment Variables

4.4.7.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set in the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified in the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for for "hwconfig" on LS1043ARDB is as follows:

```
hwconfig = fsl_ddr:bank_intlv=auto
```

4.4.7.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

* <ethx> is the Ethernet port on the board connected to the Linux boot server. "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to approriate MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9f:ef:00:00
=>setenv eth1addr 00:04:9f:ef:01:01
=>setenv eth2addr 00:04:9f:ef:02:02
=>setenv eth3addr 00:04:9f:ef:03:03
=>saveenv
```

NOTE

1. For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).
2. In the overwhelming majority of cases, eth<*>addr can be autoset.

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.7.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for LS1043A version information and also for the version of FMan microcode currently flashed on the LS1043A (e.g., microcode version 106.4.18). For QorIQ SDK 2.0, the following FMan microcode binaries should be used:

```
fsl_fman_ucode_ls1043_r1.1_106_4_18.bin (*)
fsl_fman_ucode_ls1043_r1.1_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a New U-boot, RCW, FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.7.5 RCW (Reset Configuration Word)

The RCW directories' names conform to the following naming convention:

```
ab_cdef_g
```

Table 42. RCW directories Naming Convention Legend

Slot	Convention
a	'R' indicates RGMII1@DTSEC3 is supported 'N' if not available/not used
b	'R' indicates RGMII2@DTSEC4 is supported 'N' if not available/not used
c	What is available in lane A
d	What is available in lane B
e	What is available in lane C
f	What is available in lane D
g	Hex value of serdes1 protocol value

Table 43. For Lanes (C through F)

Flag	Convention
'N'	NULL, not available/not used
'P'	PCIe
'X'	XAUI
'S'	SGMII
'Q'	QSGMII
'F'	XFI

Table continues on the next page...

Table 43. For Lanes (C through F) (continued)

'H'	SATA
'A'	AURORA

For example,

```
RR_FQPP_1455
```

means:

- RGMII1@DTSEC3 on board
- RGMII2@DTSEC4 on board
- XFI
- QSGMII
- PCIe2 on Mini-PCIe slot
- PCIe3 on PCIe Slot
- SERDES1 Protocol is 0x1455

The RCW file names for the ls1043ardb conform to the following naming convention:

```
rcw_<frequency>_<specialsetting>.rcw
```

Table 44. RCW Files Naming Convention Legend

Code		**Convention**
frequency		Core frequency(MHZ)
specialsetting	bootmode	SD/NAND/NOR and so on
	special support	nand: Nand boot sben: Secure boot lpuart: lpuart1 qspiboot: qspi boot

For example,

```
rcw_1500_sd.rcw means rcw for core frequency of 1500MHz with sd boot.
```

```
ls1043ardb/RR_FQPP_1455/rcw_1500.rcw means rcw for core frequency 0f 1500MHz with NOR boot.
```

The following RCW binaries are used on the ls1043ardb

```
RR_FQPP_1455/rcw_1500.bin
```

```
RR_FQPP_1455/rcw_1500_qetdm.bin
```

```
RR_FQPP_1455/rcw_1500_sben.bin
```

```
RR_FQPP_1455/rcw_1600.bin
```

```
RR_FQPP_1455/rcw_1600_qetdm.bin
```

```
RR_FQPP_1455/rcw_1600_sben.bin
```

4.4.7.6 System Memory Map

In 64-bit u-boot, there is a 1:1 mapping of physical address and effective address. After system startup, the boot loader maps physical address and effective address as shown in the following table:

Start Physical Address	End Physical Address	Memory Type	Size
0x00_0000_0000	0x00_00FF_FFFF	Secure Boot ROM	1MB
0x00_0100_0000	0x00_0FFF_FFFF	CCSR	240MB
0x00_1000_0000	0x00_1000_FFFF	OCRAM0	64KB
0x00_1001_0000	0x00_1001_FFFF	OCRAM1	64 KB
0x00_2000_0000	0x00_23FF_FFFF	DCSR	64MB
0x00_4000_0000	0x00_5FFF_FFFF	QSPI	512MB
0x00_6000_0000	0x00_7FFF_FFFF	IFC region	512MB
0x00_8000_0000	0x00_FFFF_FFFF	DRAM	2GB
0x05_0000_0000	0x05_07FF_FFFF	QMAN S/W Portal	128M
0x05_0800_0000	0x05_0FFF_FFFF	BMAN S/W Portal	128M
0x40_0000_0000	0x47_FFFF_FFFF	PCI Express1	32G

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Memory Type	Size
0x48_0000_0000	0x4F_FFFF_FFFF	PCI Express2	32G
0x50_0000_0000	0x57_FFFF_FFFF	PCI Express1	32G

4.4.7.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
U-Boot 2016.012.0+g2ea81ad

Clock Configuration:
       CPU0(A53):1600 MHz  CPU1(A53):1600 MHz  CPU2(A53):1600 MHz
       CPU3(A53):1600 MHz
       Bus:      400  MHz  DDR:      1600 MT/s  FMAN:     500  MHz
Reset Configuration Word (RCW):
       00000000: 08100010 0a000000 00000000 00000000
       00000010: 14550002 80004012 e0025000 c1002000
       00000020: 00000000 00000000 00000000 00038800
       00000030: 00000000 00001101 00000096 00000001
I2C:   ready
Model: LS1043A RDB Board
Board: LS1043ARDB, boot from vBank 4
CPLD:  V1.4
PCBA:  V3.0
SERDES Reference Clocks:
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
DRAM:  Initializing DDR....
Detected UDIMM Fixed DDR on board
2 GiB (DDR4, 32-bit, CL=11, ECC off)
SEC: RNG instantiated
Firmware 'Microcode version 0.0.1 for LS1021a r1.0' for 1021 V1.0
QE: uploading microcode 'Microcode for LS1021a r1.0' version 0.0.1
Waking secondary cores to start from ffd0e000
All (4) cores are up.
Using SERDES1 Protocol: 5205 (0x1455)
Flash: 128 MiB
NAND:  512 MiB
MMC:   FSL_SDHC: 0
EEPROM: NXID v1
PCIe1: disabled
PCIe2: Root Complex no link, regs @ 0x3500000
PCIe3: Root Complex x1 gen1, regs @ 0x3600000
PCI:
     01:00.0    - 8086:10d3 - Network controller
PCIe3: Bus 00 - 01
```

```
In:    serial
Out:   serial
Err:   serial
Net:   Fman1: Uploading microcode version 106.4.18
e1000: 00:1b:21:46:61:df
       FM1@DTSEC1, FM1@DTSEC2, FM1@DTSEC3, FM1@DTSEC4, FM1@DTSEC5, FM1@DTSEC6, FM1@TGEC1, e1000#0
[PRIME]
Warning: e1000#0 MAC addresses don't match:
Address in SROM is         00:1b:21:46:61:df
Address in environment is  00:e0:0c:00:85:07

=>
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0 for RDB:

```
=>cpld reset
```

- Switch to alternate bank for RDB:

```
=>cpld reset altbank
```

The table below shows the memory map for LS1043A:

Table 45. NOR Flash Memory Map for LS1043ARDB

Start	End Offset	Description	Size
0x60000000	0x600FFFFF	bank0 rcw + pbi	1M
0x60100000	0x601FFFFF	bank0 U-Boot image	1 M
0x60200000	0x602FFFFF	bank0 U-Boot Env	1 M
0x60300000	0x603FFFFF	bank0 Fman ucode	1 M
0x60400000	0x604FFFFF	bank0 UEFI	1 M
0x60500000	0x605FFFFF	bank0 Primary Protected Application (PPA)	1 M
0x60600000	0x606FFFFF	bank0 QE firmware	1 M
0x60700000	0x60EFFFFF	bank0 reserved	8 M
0x60F00000	0x60FFFFFF	bank0 PHY firmware	1 M
0x61000000	0x610FFFFF	bank0 CORTINA PHY firmware	1 M
0x61100000	0x638FFFFF	bank0 FIT Image	40M
0x63900000	0x63FFFFFF	bank0 Unused	7 M
0x64000000	0x640FFFFF	bank4 rcw + pbi	1M
0x64100000	0x641FFFFF	bank4 U-Boot image	1 M
0x64200000	0x642FFFFF	bank4 U-Boot Env	1 M
0x64300000	0x643FFFFF	bank4 Fman ucode	1 M

Table continues on the next page...

Table 45. NOR Flash Memory Map for LS1043ARDB (continued)

0x64400000	0x644FFFFF	bank4 UEFI	1 M
0x64500000	0x645FFFFF	bank4 PPA	1 M
0x64600000	0x646FFFFF	bank4 QE firmware	1 M
0x64700000	0x64EFFFFF	bank4 reserved	8 M
0x64F00000	0x64FFFFFF	bank4 PHY firmware	1 M
0x65000000	0x650FFFFF	bank4 CORTINA PHY firmware	1 M
0x65100000	0x678FFFFF	bank4 FIT Image	40M
0x67900000	0x67FFFFFF	bank4 Unused	7 M

4.4.7.8 Programming a New U-Boot, RCW, FMan Microcode, PPA firmware

The following three sections will discuss how to individually update U-Boot, RCW, FMan Microcode, and PPA firmware. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to update mutliple images, there is no need to switch into the alternate bank after each configuration, i.e. there is no need to type the command "cpld reset altbank" after flashing each individual image.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 82000000 <u-boot_file_name>.bin
=>protect off 64100000 +$filesize
=>erase 64100000 +$filesize
=>cp.b 82000000 64100000 $filesize
=>protect on 64100000 +$filesize
=>cpld reset altbank
```

The commands above will only program a new U-Boot.

Programming U-Boot to SD card

For instructions on how to program U-Boot to SD card refer to SD Deployment on page 184.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address 0x64000000. 0x64000000 is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 82000000 <rcw_file_name>.bin
=>protect off 64000000 +$filesize
=>erase 64000000 +$filesize
```

```
=>cp.b 82000000 64000000 $filesize
=>protect on 64000000 +$filesize
=>cpld reset altbank
```

Programming a New FMan Microcode

Program a new microcode to NOR flash:

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address 0x64300000. 0x64300000 is the location of microcode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the microcode to flash and reset to alternate bank.

```
=>tftp 83000000 <ucode_file_name>.bin
=>protect off 64300000 +$filesize
=>erase 64300000 +$filesize
=>cp.b 83000000 64300000 $filesize
=>protect on 64300000 +$filesize
=>cpld reset altbank
```

Program a new microcode to NAND flash:

```
=>tftp 83000000 <ucode_file_name>.bin
=>nand erase 160000 +$filesize;nand write 83000000 160000 $filesize
=>cpld reset nand
```

Program a new microcode to SD Card:

```
=>tftp 83000000 <ucode_file_name>.bin
=>mmc write 83000000 820 50
=>cpld reset sd
```

Programming a New Primary Protected Application (PPA) firmware

Program a new PPA firmware to NOR flash:

To program a new PPA firmware, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new PPA firmware to RAM by downloading it via TFTP and then copying it to flash at address 0x64500000. 0x64500000 is the location of PPA firmware in the alternate bank. Then execute the following commands at the U-Boot prompt to program the PPA firmware to flash and reset to alternate bank.

```
=>tftp 83000000 <ppa_file_name>.itb
=>protect off 64500000 +$filesize
=>erase 64500000 +$filesize
=>cp.b 83000000 64500000 $filesize
=>protect on 64500000 +$filesize
=>cpld reset altbank
```

4.4.7.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the NOR Flash Memory Map within Flash Bank Usage as reference for the specific addresses.

4.4.7.9.1 FIT Image Deployment from TFTP

1. **Setting U-Boot Environment**

   Before performing FIT image deployment, the U-Boot environment variables need to be configured.

Refer to Configuring U-Boot Network Parameters on page 175 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for FIT image deployment from TFTP:

```
=>setenv bootargs 'root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200'
=>saveenv
```

2. **Booting Up the System**

   Execute the following commands to TFTP the image to the board, then boot into Linux.

```
=>tftp a0000000 < FIT_image_name>
=>bootm a0000000
```

   Now the board will boot into Linux .

4.4.7.9.2 FIT Image Deployment from Flash

1. **Setting U-Boot Environment**

   Before performing FIT image from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for deployment from flash:

```
=>setenv  bootargs root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200
=>setenv bootcmd   bootm <FIT_image_addr>
=>saveenv
```

   Now U-Boot is ready for flash deployment.

2. **Programming FIT image to NOR Flash**

   The image should be downloaded to the RAM using TFTP then copied to the flash address <FIT_image_addr>. At the U-Boot prompt, use the following commands to program the image to flash:

```
=>tftp 82000000 <FIT Image name>
=>protect off <FIT_image_addr> +$filesize
=>erase <FIT_image_addr> +$filesize
=>cp.b 82000000 <FIT_image_addr> $filesize
=>protect on <FIT_image_addr> +$filesize
```

3. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <FIT_image_addr>
```

4. **FIT Image address on Flash Memory Map**

```
> Bank0: FIT_image_addr = 0x0_6110_0000
```

```
> Bank4: FIT_image_addr = 0x0_6510_0000
```

4.4.7.9.3 NFS Deployment

1. **Generating File System**

   Use Yocto to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   nfs_root_path board_ipaddress(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/portmap restart
   ```

   ```
   /etc/init.d/nfs-kernel-server  restart
   ```

   NOTE

   nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 175 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>: <your_gatewayip>:<your_netmask>:<board_name>:<ethx>:off
   console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500
   =>saveenv
   ```

   NOTE

   <ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel FIT image to the board, then boot it up.

   ```
   =>tftp a0000000  kernel.itb;
   =>bootm a0000000:kernel@1 - a0000000:fdt@1
   ```

   Now the board will boot up with NFS filesystem.

4.4.7.9.4 SD Deployment

This section shows steps to load the U-Boot image, kernel image and rootfs into a SD card and boot from it. The first step is to partition a SD card. No less than 2M space should be reserved for RCW, PBI, and U-Boot images.

The second step is to put U-Boot image into SDcard:

There are two ways to deploy U-Boot image into SDcard: * Use "dd" command under a linux host. * Use "mmc write" command under U-Boot prompt. The last step is to put the kernel image and rootfs into SDcard: There are two types of rootfs to select. * Use SDcard partition as the rootfs. * Use Ramdisk as the rootfs.

Create a new partition on SD Card

1. Insert SD card into the Linux Host PC. After inserting it, use the `dmesg` command to determine the name of sd card.

```
$ dmesg | tail -20
...
[84626.337992] sd 5:0:0:0: [sdc] 499712 512-byte logical blocks: (255 MB/244 MiB)
[84626.339229] sd 5:0:0:0: [sdc] Write Protect is off
[84626.339234] sd 5:0:0:0: [sdc] Mode Sense: 0b 00 00 08
[84626.340482] sd 5:0:0:0: [sdc] No Caching mode page found
[84626.340488] sd 5:0:0:0: [sdc] Assuming drive cache: write through
```

   From the above log, we could know the name of sd card. In this instance, it is "`sdc`". If you couldn't see the log as above, please use `dmesg | tail - 30(or greater)` to show more.

2. Use the `fdisk` command to repartition the SD card.

   a. Use the "df" to check whether partitions of sd card has being mounted or not by host automatically .

```
$ df
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/sda2       38317716  29660848   6687360  82% /
udev              927520         4    927516   1% /dev
tmpfs             201372      1064    200308   1% /run
/dev/sdc1         236876      2062    218277   1% /media/songwb/
5e36404d-9c18-44ba-9bec-6525862dfdb7
```

   From the log above, if any partition of SD card has been mounted, umount it. For this instance, `sdc1` has been mounted, so umount the `/dev/sdc1` using the `unmount` commmand.

```
$ umount /media/songwb/5e36404d-9c18-44ba-9bec-6525862dfdb7
```

   b. Use the `fdisk` command to create a new partition.

```
#fdisk /dev/sdc
Command (m for help): p

Disk /dev/sdc: 255 MB, 255852544 bytes
4 heads, 16 sectors/track, 7808 cylinders, total 499712 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x354419bd

Device Boot      Start         End      Blocks   Id  System
/dev/sdc1            2048      499711      248832    6  FAT16
```

NOTE

`sdc` is your SD device name from step1.

NOTE

note: "p" means printing all partitions on SD card.

```
Command (m for help): d
Selected partition 1
```

NOTE

if there has been a partition, delete it using `d`.

```
Command (m for help): n
Partition type:
            p   primary (0 primary, 0 extended, 4 free)
            e   extended
```

NOTE

`n` means creating a new partition.

```
Select (default p):
Using default response p
```

NOTE

select a default.

```
Partition number (1-4, default 1): note: select a default.
Using default value 1
First sector (2048-499711, default 2048): 4096
```

NOTE

Skip the first `4096` blocks reserved for deploying U-Boot image.

```
Last sector, +sectors or +size{K,M,G} (4096-499711, default 499711):
Using default value 499711
```

NOTE

select a default.

```
Command (m for help): w
The partition table has been altered!
```

NOTE

`w` means writing the new partition table to sd card.

```
Calling ioctl() to re-read partition table.
Syncing disks.
```

3. Use the `mkfs.ext2` command to format the filesystem.

```
# mkfs.ext2 /dev/sdc1

mke2fs 1.42.9 (4-Feb-2014)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
62000 inodes, 247808 blocks
12390 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=67371008
31 block groups
```

```
8192 blocks per group, 8192 fragments per group
2000 inodes per group
Superblock backups stored on blocks:
       8193, 24577, 40961, 57345, 73729, 204801, 221185

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

NOTE

The first 4096 sectors of SD card must be reserved for deploying U-Boot image.

Deploy the kernel image and rootfs

1. Mount this new partition to a temporary directory

```
#mkdir temp
#mount /dev/sdc1 temp
```

2. Deploy the linux fit iamge.

```
#cp  kernel.itb temp/
```

3. Deploy the rootfs. If you want to use this new partition as rootfs, follow the below steps, if not, skip this step.

```
#cp fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz  temp/
#tar xvfz fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz
#rm  fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz  temp/.
```

4. Umount the temporary directory

```
#umount temp
```

Deploy the U-Boot image

There are two ways to deploy the U-Boot image to SD card. And note that PreBoot loader (PBL) will load the U-Boot image at the offset of `0x1000`, So we should skip 8 blocks(if the block size of sd card is 512B).

- Under the linux host, use `dd` command to deploy it
  1. Insert SD card into Host.
  2. Use the `dd` command

```
# dd if=u-boot-with-spl-pbl.bin of=/dev/sdb  seek=8 bs=512
```

- Under U-Boot prompt, use `mmc write` to deploy it
  1. Insert SD card into target board and power on.
  2. Setup the network connection if necessary
  3. Program U-Boot image to SD Card

```
=> tftpboot 82000000 u-boot-with-spl-pbl.bin
=> mmc write 82000000 8 800
=> cpld reset sd
```

If the SD card is already in the slot before powering on the board, the card will be initialized when U-Boot boots. If the SD card is inserted after U-Boot has booted, use `mmc rescan` to scan and initialize the sd card, and use `mmc info` to make sure the sd card is ok:

```
=> mmc rescan
=> mmcinfo
```

Bootup kernel

If you want to use SD card partition as rootfs, make sure the rootfs image has been delopyed in it, and use the following steps.

```
=> setenv bootargs  "root=/dev/mmcblk0p1 rw earlycon=uart8250,0x21c0500 console=ttyS0,115200"
=> ext2load mmc 0 a0000000 kernel.itb && bootm a0000000
```

If you want to use RAMDISK as rootfs, use the following steps.

```
=> setenv bootargs  "root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200"
=> ext2load mmc 0 a0000000 kernel.itb && bootm a0000000
```

NOTE

The `kernel.itb` is the name of your FIT Image, you can use the `ext2ls` command to list it at the U-Boot prompt

4.4.7.9.5 QSPI Deployment

Insert an X-nor card

1. X-nor card hardware configuration:

```
SW1[1:4]=1000
SW4[1:4]=1111
SW3[1:4]=0001
```

2. Insert the X-nor card into the IFC Card slot to enable QSPI flash on it.

Prepare SD boot to program QSPI flash

1. Build U-Boot image for SD boot (enables QSPI):

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/gcc-linaro-aarch64-linux-
gnu-4.9-2014.07_linux/bin/aarch64-linux-gnu-
$make ARCH=aarch64 ${board_name}_sdcard_qspi_defconfig
$make CROSS_COMPILE=${toolchain_path}/gcc-linaro-aarch64-linux-gnu-4.9-2014.07_linux/bin/
aarch64-linux-gnu- -j4
```

2. Write the U-Boot image to SD card:

```
=>tftp 0x81000000 u-boot-with-spl-pbl.bin
=>mmc write 0x81000000 8 0x800
```

3. Switch to SD boot(enables QSPI):

```
=>qixis_reset sd_qspi
```

Or set switches referring to board configuration document and power on the board from SD boot.

Build U-Boot image for QSPI boot

1. Compile QSPI boot image(enable QSPI):

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/gcc-linaro-aarch64-linux-
gnu-4.9-2014.07_linux/bin/aarch64-linux-gnu-
$make ARCH=aarch64  ${board_name}_qspi_defconfig
$make CROSS_COMPILE=${toolchain_path}/gcc-linaro-aarch64-linux-gnu-4.9-2014.07_linux/bin/
aarch64-linux-gnu- -j4
```

2. Swap the bytes for RCW:

```
$tclsh byte_swap.tcl rcw_1600_qspiboot.bin rcw_1600_qspiboot_swap.bin 8
```

   The byte_swap.tcl script is a shareable tool and can be found under `rcw/tool/` directory.

3. Write RCW and U-Boot images to QSPI flash under SD boot (enables QSPI) mode:

```
=>sf probe 0:0
```

   SF: Detected S25FL128S_64K with page size 256 Bytes, erase size 64 KiB, total 16 MiB

```
=>tftp 81000000 rcw_1600_qspiboot_swap.bin;sf erase 0 +$filesize;sf write 81000000 0 $filesize
=>tftp 82000000 u-boot-dtb.bin;sf erase 10000 +$filesize;sf write 82000000 10000 $filesize
```

4. Switch to QSPI boot:

```
=>qixis_reset qspi
```

   Or set switches referring to board configuration document and power on the board from QSPI boot.

Write FMan ucode to QSPI flash to use FMan
Do it under SD boot(enbles QSPI) or QSPI boot:

```
=>sf probe 0:0
```

SF: Detected S25FL128S_64K with page size 256 Bytes, erase size 64 KiB, total 16 MiB

```
=>tftp 83000000 <ucode_file_name>.bin
=>sf probe 0:0;sf erase d0000 +$filesize;sf write 83000000 d0000 $filesize
```

4.4.7.10 Check 'Link Up' for Serial Ethernet Interfaces

If you are experiencing problems with your Ethernet interfaces, this section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
FSL_MDIO0:
1 - RealTek RTL8211F <--> FM1@DTSEC3
2 - RealTek RTL8211F <--> FM1@DTSEC4
4 - Vitesse VSC8514 <--> FM1@DTSEC1
```

```
5 - Vitesse VSC8514  <--> FM1@DTSEC2
6 - Vitesse VSC8514  <-->  FM1@DTSEC5
7 - Vitesse VSC8514 <--> FM1@DTSEC6
FM_TGEC_MDIO:
1 - Aquantia AQR105  <-->  FM1@TGEC1
```

The results from the above *mdio list* command show that U-Boot was able to see PHYs on each of the 6 DTSEC interfaces and on the 10GEC interface. If you see “Generic” reported, it is an indication that something is there but the ls1043ardb can’t communicate with the device/port.

Check Link Status for External PHY

In order to check the status of a SGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read FM1@DTSEC1 1
Reading from bus FSL_MDIO0
PHY at address 2:
1 - 0x796d
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7969.

4.4.7.11 Hardware Setting for Special Purposes

The RDB has user selectable switches for evaluating different pin mux and boot options for the LS1043A device. Information below lists the hardware setting for those special purposes.

For RDB:

General UART port :

1. UART1(J4 bottom)
2. UART2(J4 top).

NOR boot setting

1. Set SW4[1:8]+SW5[1]=0b'0001_0010_1;
2. Bank0: SW5[4:6]=0b'000;
3. Bank1: SW5[4:6]=0b'001

SD boot setting

1. image:u-boot-with-spl-pbl.bin
2. Program the image to SD card in u-boot.

```
=>tftp 82000000 u-boot-with-spl-pbl.bin
=>=>mmc erase 8 0x800; mmc write 82000000 8 0x800
```

3. Program the image to SD card in Linux.

```
dd if=u-boot-with-spl-pbl-sd.bin of=/dev/sdb bs=512 seek=8
```

4. Insert SD card in adaptor;
5. Set SW4[1:8]+SW5[1]=0b'0010_0000_0

NAND boot setting

1. image:u-boot-with-spl-pbl.bin
2. Program the image to SD card in u-boot.

```
=>tftp 82000000 u-boot-with-spl-pbl.bin
=>=>nand erase.chip; nand write 82000000 0 100000
```

3. Set SW4[1:8]+SW5[1]=0b'1000_0011_0

NOTE

For more information about specific settings for the console (i.e. baud rate, ports used etc.), please refer to the ls1043ardb Quick Start Guide.

For customized board

General UART port:

Refer to board configuration document

Lpuart

1. RCW - support for lpuart
   a. Program the RCW image

```
=>tftp 82000000 <rcw_lpuart_image.bin>;
```

```
=>protect off <rcw_image_addr> +$filesize;
```

```
=>erase <rcw_image_addr> +$filesize
```

```
=>cp.b 0x82000000 <rcw_image_addr> $filesize
```

```
=>protect on <rcw_image_addr> +$filesize
```

2. U-boot - with lpuat console
   a. Program the uboot image

```
=>tftp 82000000 u-boot-dtb.bin
```

```
=>protect off <uboot_image_addr> +$filesize
```

```
=>erase <uboot_image_addr> +$filesize
```

```
=>cp.b 0x82000000 <uboot_iamge_addr> $filesize
```

```
=>protect on <uboot_iamge_addr> +$filesize
```

3. Boot using lpuart console
   a. Set uboot environment variable.

```
=>setenv bootargs "console=ttyLP0,115200 root=/dev/ram0 earlycon=lpuart32,0x2950000"
```

   b. Bootup kernel

```
=>tftp a0000000 kernel.itb
```

```
=>bootm a0000000
```

SATA

1. RCW - support for SATA
   a. Program the RCW image

```
tftp 82000000 <rcw_sata_image.bin>
```

```
protect off <rcw_image_addr> +$filesize
```

```
erase <rcw_image_addr> +$filesize
```

```
cp.b 0x82000000 <rcw_image_addr> $filesize
```

```
protect on <rcw_image_addr> +$filesize
```

2. U-boot
   a. Program the uboot image

```
=>tftp 82000000 u-boot-dtb.bin
```

```
=>protect off <uboot_image_addr> +$filesize
```

```
=>erase <uboot_image_addr> +$filesize
```

```
=>cp.b 0x82000000 <uboot_image_addr> $filesize
```

```
=>protect on <uboot_image_addr> +$filesize
```

3. Boot & Test
   a. Program the RCW and U-Boot image.
   b. Enable SATA support

```
Refer to board configuration document
```

   c. Boot using the new image
   d. Test SATA on uboot

```
=>scsi info
```

e. Bootup kernel

```
=>tftp a0000000 kernel.itb && bootm a0000000
```

f. Test SATA on kernel

```
#hdparm -i /dev/sda
```

4.4.8 LS1046ARDB

4.4.8.1 Overview

The LS1046A reference design board (RDB) is the high-performance computing, evaluation, and development platform that supports the QorIQ LS1046A processor. This guide provides board-specific configurations and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.8.2 Switch Settings

The RDB has user selectable switches for evaluating different boot options for the LS1046A device. Table below lists the default switch settings and the description of these settings.('0' is OFF, '1' is ON.)

Table 46. Default Switch Settings

	1	2	3	4	5	6	7	8
SW3	0	1	0	0	0	1	1	0
SW4	0	0	1	1	1	0	1	1
SW5	0	0	1	0	0	0	1	0

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 47. LS1046ARDB Switch Settings

Boot Source	Switch
QSPI flash 0 (bank0)	SW5[1-8] +SW4[1] = 0b'00100010_0 SW3[3-5]= 0b'000
QSPI flash 1 (bank4)	SW5[1-8] +SW4[1] = 0b'00100010_0 SW3[3-5]= 0b'001
SD	SW5[1-8] +SW4[1] = 0b'00100000_0

4.4.8.3 U-Boot Environment Variables

The following sections will guide the users on how to set the U-Boot environment and configure the U-Boot network parameters.

4.4.8.3.1 U-Boot Environment Variable "hwconfig"

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set in the U-Boot prompt using the "*setenv*" command.

- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified in the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for for "hwconfig" on LS1046ARDB is as follows:

```
hwconfig = fsl_ddr:bank_intlv=auto
```

4.4.8.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

* <ethx> is the Ethernet port on the board connected to the Linux boot server. "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to approriate MAC addresses for your board.

```
=>setenv ethaddr 00:e0:0c:00:89:00
=>setenv eth1addr 00:e0:0c:00:89:01
=>setenv eth2addr 00:e0:0c:00:89:02
=>setenv eth3addr 00:e0:0c:00:89:03
=>saveenv
```

NOTE

1. For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).
2. In the overwhelming majority of cases, eth<*>addr can be autoset.

The flashed version of U-Boot is now ready for TFTP based deployments.

4.4.8.4 Frame Manager Microcode (FMan Ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See

the U-Boot log for LS1046A version information and also for the version of FMan microcode currently flashed on the LS1046A (e.g., microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

```
fsl_fman_ucode_ls1046_r1.0_106_4_18.bin(*)
fsl_fman_ucode_ls1046_r1.0_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(i) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(ii) For instructions on how to flash a new FMan microcode image, see Programming a New U-boot, RCW, FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.8.5 RCW (Reset Configuration Word) and Ethernet Interfaces

The RCW directories' names conform to the following naming convention:

```
ab_cdef_ghij
```

Table 48. RCW directories Naming Convention Legend

Slot	Convention
a	'R' indicates RGMII1@DTSEC3 is supported
b	'R' indicates RGMII2@DTSEC4 is supported 'N' indicates not available/not used
c	What is available in SerDes1 Lane 0
d	What is available in SerDes1 Lane 1
e	What is available in SerDes1 Lane 2
f	What is available in SerDes1 Lane 3
g	What is available in SerDes2 Lane 0
h	What is available in SerDes2 Lane 1
i	What is available in SerDes2 Lane 2
j	What is available in SerDes2 Lane 3

Table 49. For Lanes (c through j)

Flag	Convention
'N'	NULL, not available/not used

Table continues on the next page...

Table 49. For Lanes (c through j) (continued)

'P'	PCIe
'X'	XAUI
'S'	SGMII
'Q'	QSGMII
'F'	XFI
'H'	SATA
'A'	AURORA

For example,

```
RR_FFPPPN_1133_5559
```

means:

- RGMII1@DTSEC3 on board
- RGMII2@DTSEC4 on board
- XFI9 on SFP cage
- XFI10 on SFP cage
- SGMII5 on Slot 2
- SGMII6 on Slot 1
- PCIe1 on Slot 3
- PCIe2 on Slot 4
- PCIe3 on Slot 5
- SATA
- SERDES1 Protocol is 0x1133
- SERDES2 Protocol is 0x5559

The RCW file names for the ls1046ardb conform to the following naming convention:

```
rcw_<frequency>_<specialsetting>.rcw
```

Table 50. RCW Files Naming Convention Legend

Code		Convention
frequency		Core frequency(MHZ)
specialsetting	bootmode	QSPI/SD/EMMC

Table continues on the next page...

Table 50. RCW Files Naming Convention Legend (continued)

	special support	emmc: eMMC boot sdboot: SD boot sben: Secure boot qspiboot: QSPI boot

For example,

```
rcw_1600_sdboot.rcw means rcw for core frequency of 1600MHz with sd boot.
```

```
ls1046ardb/RR_FFPPPN_1133_5559/rcw_1600_qspiboot.rcw means rcw for core frequency 0f 1600MHz with
QSPI boot.
```

The following RCW binaries are used on the ls1046ardb:

```
RR_FFPPPN_1133_5559/rcw_1600_qspiboot.bin
```

```
RR_FFPPPN_1133_5559/rcw_1600_qspiboot_sben.bin
```

4.4.8.6 System Memory Map

In 64-bit u-boot, there is a 1:1 mapping of physical address and effective address. After system startup, the boot loader maps physical address and effective address as shown in the following table:

Start Physical Address	End Physical Address	Memory Type	Size
0x00_0000_0000	0x00_00FF_FFFF	Secure Boot ROM	1MB
0x00_0100_0000	0x00_0FFF_FFFF	CCSR	240MB
0x00_1000_0000	0x00_1000_FFFF	OCRAM0	64KB
0x00_1001_0000	0x00_1001_FFFF	OCRAM1	64KB
0x00_2000_0000	0x00_23FF_FFFF	DCSR	64MB
0x00_4000_0000	0x00_5FFF_FFFF	QSPI	512MB
0x00_6000_0000	0x00_7FFF_FFFF	IFC region	512MB
0x00_8000_0000	0x00_FFFF_FFFF	DRAM	2GB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Memory Type	Size
0x05_0000_0000	0x05_07FF_FFFF	QMAN S/W Portal	128M
0x05_0800_0000	0x05_0FFF_FFFF	BMAN S/W Portal	128M
0x08_8000_0000	0x0F_FFFF_FFFF	DRAM	30GB
0x40_0000_0000	0x47_FFFF_FFFF	PCI Express1	32G
0x48_0000_0000	0x4F_FFFF_FFFF	PCI Express2	32G
0x50_0000_0000	0x57_FFFF_FFFF	PCI Express3	32G

4.4.8.7 Flash Bank Usage

LS1046ARDB has 2 QSPI flash connected over QSPI contoller.

Only one QSPI flash is available at a time depending upon the board switch settings. These switch settings can also be overriden by CPLD commands.

To protect the default U-Boot in flash0 (aka bank0), it is a convention employed by NXP to deploy work images into the flash1 (aka bank4), and then switch to the flash1 (aka bank4) for testing. Switching to the flash1 (aka bank4) can be done in software using CPLD commands and effectively swaps the flash0 (aka bank0) with the flash1 (aka bank4). This protects flash1 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
U-Boot 2016.01 (Aug 19 2016 - 16:59:27 +0800)

SoC:  LS1046E (0x87070010)
Clock Configuration:
       CPU0(A72):1600 MHz  CPU1(A72):1600 MHz  CPU2(A72):1600 MHz
       CPU3(A72):1600 MHz
       Bus:      600  MHz  DDR:      2100 MT/s  FMAN:     700  MHz
Reset Configuration Word (RCW):
       00000000: 0c150010 0e000000 00000000 00000000
       00000010: 11335559 40005012 40025000 c1000000
       00000020: 00000000 00000000 00000000 00238800
       00000030: 20124000 00003101 00000096 00000001
I2C:   ready
Model: LS1046A RDB Board
Board: LS1046ARDB, boot from QSPI vBank 0
CPLD:  V2.2
PCBA:  V2.0
SERDES Reference Clocks:
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
DRAM:  Initializing DDR....using SPD
Detected UDIMM 18ASF1G72AZ-2G3B1
8 GiB (DDR4, 64-bit, CL=15, ECC on)
       DDR Chip-Select Interleaving Mode: CS0+CS1
```

```
SEC0: RNG instantiated
PPA Firmware: Version 0.2
Using SERDES1 Protocol: 4403 (0x1133)
Using SERDES2 Protocol: 21849 (0x5559)
NAND:  512 MiB
MMC:   FSL_SDHC: 0
SF: Detected S25FL512S_256K with page size 256 Bytes, erase size 256 KiB, total 64 MiB
EEPROM: Invalid ID (5a 5a 5a 5a)
PCIe1: Root Complex no link, regs @ 0x3400000
PCIe2: Root Complex no link, regs @ 0x3500000
PCIe3: Root Complex no link, regs @ 0x3600000
In:    serial
Out:   serial
Err:   serial
SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI:  Net:   SF: Detected S25FL512S_256K with page size 256 Bytes, erase size 256 KiB, total 64 MiB
Fman1: Uploading microcode version 106.4.15
FM1@DTSEC3 [PRIME], FM1@DTSEC4, FM1@DTSEC5, FM1@DTSEC6, FM1@TGEC1, FM1@TGEC2
Hit any key to stop autoboot:  0
=>
```

Bank switching can be done in U-Boot using the following statements:

- Switch to QSPI bank 0(default) for RDB:

```
=>cpld reset
```

- Switch to QSPI bank 4 for RDB:

```
=>cpld reset altbank
```

The table below shows the QSPI flash memory map for LS1046ARDB:

Table 51. QSPI Flash Memory Map for LS1046ARDB

Start	End Offset	Description	Size
0x40000000	0x400FFFFF	bank0 rcw + pbi	1 M
0x40100000	0x401FFFFF	bank0 U-Boot image	1 M
0x40200000	0x402FFFFF	bank0 U-Boot Env	1 M
0x40300000	0x403FFFFF	bank0 Fman ucode	1 M
0x40400000	0x404FFFFF	bank0 UEFI	1 M
0x40500000	0x406FFFFF	bank0 Primary Protected Application (PPA)	2 M
0x40700000	0x408FFFFF	bank0 Secure boot header + bootscript	2 M
0x40900000	0x40FFFFFF	bank0 reserved	7 M
0x41000000	0x43FFFFFF	bank0 FIT Image	48 M
0x44000000	0x440FFFFF	bank4 rcw + pbi	1 M
0x44100000	0x441FFFFF	bank4 U-Boot image	1 M

Table continues on the next page...

Table 51. QSPI Flash Memory Map for LS1046ARDB (continued)

0x44200000	0x442FFFFF	bank4 U-Boot Env	1 M
0x44300000	0x443FFFFF	bank4 Fman ucode	1 M
0x44400000	0x444FFFFF	bank4 UEFI	1 M
0x44500000	0x446FFFFF	bank4 PPA	2 M
0x44700000	0x448FFFFF	bank4 Secure boot header + bootscript	2 M
0x44900000	0x44FFFFFF	bank4 reserved	7 M
0x45000000	0x47FFFFFF	bank4 FIT Image	48 M

4.4.8.8 Programming a New U-Boot, RCW and FMan Ucode

The following three sections will discuss how to individually update U-Boot, RCW. For specific addresses, please refer to the QSPI Flash Memory Map as a reference.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Here are U-Boot commands to switch between the three different boot sources:

1. `cpld reset`: reset to boot from current bank
2. `cpld reset altbank`: reset to boot from alternate bank
3. `cpld reset sd`: reset to boot from SD card

Programming a New U-Boot to QSPI flash

By default, an existing U-Boot is run in QSPI bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp <uboot_image_addr> <uboot_file_name>.bin
=>sf probe 0:1
=>sf erase 100000 +$filesize
=>sf write <uboot_image_addr> 100000 $filesize
=>cpld reset altbank
```

NOTE

Using "sf probe 0:0" could program U-Boot to the current bank.

Programming a New RCW to QSPI flash

To program a new RCW, first switch to QSPI bank 0 by performing a hard reset or by typing *reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash offset 0x0. 0x0 is the offset address of RCW in the QSPI flash. Execute the following commands at the U-Boot prompt to program the RCW to QSPI flash and reset to alternate bank.

```
=>tftp <rcw_image_addr> <rcw_file_name>.bin
=>sf probe 0:1
=>sf erase 0 +$filesize
=>sf write <rcw_image_addr> 0 $filesize;
=>cpld reset altbank
```

NOTE

1. Using "sf probe 0:0" could program RCW to the current bank.
2. The RCW image needs to be swapped. Yocto Project generates the swapped RCW image, which is programmed into QSPI flash. For steps to generate RCW and U-Boot without using Yocto Project, refer to QSPI Deployment.

Programming a New U-Boot to SD card

To program U-boot, first boot the board to u-boot. Next, load the new U-Boot SD boot image(u-boot-with-spl-pbl.bin) to RAM by downloading it via TFTP and then copying it to SD card with blk offset 0x8. Execute the following commands at the U-Boot prompt to program the U-Boot to SD card and reset to sd boot.

```
=>tftp <uboot_image_addr> u-boot-with-spl-pbl.bin
=>mmc erase 8 0x800
=>mmc write <uboot_image_addr> 8 0x800
=>cpld reset sd
```

Program the image to SD card in Linux.

```
dd if=u-boot-with-spl-pbl-sd.bin of=/dev/sdb bs=512 seek=8
```

Programming a New FMan Microcode

Program a new microcode to QSPI flash 1(bank 4):

To program a new microcode, first switch to QSPI bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new microcode to RAM by downloading it via TFTP and then writing it to flash offset 0x300000. 0x300000 is the offset of ucode in the QSPI flash. Then execute the following commands at the U-Boot prompt to program the ucode to the boot device.

```
=>tftp <ucode_image_addr> <ucode_file_name>.bin
=>sf probe 0:1
=>sf erase 300000 +$filesize
=>sf write <ucode_image_addr> 300000 $filesize

NOTE: Using "sf probe 0:0" could program microcode to the current bank.
```

Program a new microcode to SD Card:

```
=>tftp <ucode_image_addr> <ucode_file_name>.bin
=>mmc write <ucode_image_addr> 820 50
```

4.4.8.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.8.9.1 FIT Image Deployment from TFTP

1. **Setting U-Boot Environment**

   Before performing FIT image deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters on page 194 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for FIT image deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200'
   =>saveenv
   ```

2. **Booting Up the System**

   Execute the following commands to TFTP the image to the board, then boot into Linux.

```
=>tftp a0000000 <fit_image_name>
=>bootm a0000000
```

   Now the board will boot into Linux .

4.4.8.9.2 FIT Image Deployment from Flash

1. **Setting U-Boot Environment**

   Before performing FIT image from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for deployment from flash:

```
=>setenv bootargs root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200
=>setenv bootcmd sf probe 0:0;sf read <fit_image_addr> 1000000 2800000;bootm <fit_image_addr>
=>saveenv
```

2. **Programming FIT image to QSPI Flash**

   The FIT image should be downloaded to the RAM address using TFTP then copied to flash offset 0x1000000 as per "QSPI memory map" in Flash Bank Usage. At the U-Boot prompt, use the following commands to program the image to QSPI current bank:

```
=>tftp <fit_image_addr> <fit_image_name>
=>sf probe 0:0
=>sf erase 0x1000000 +$filesize
=>sf write <fit_image_addr> 0x1000000 $filesize
```

3. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <fit_image_addr>
```

4.4.8.9.3 NFS Deployment

1. **Generating File System**

   Use Yocto to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
nfs_root_path board_ipaddress(rw,no_root_squash,async)
```

b. Restart the NFS service:

```
/etc/init.d/portmap restart
```

```
/etc/init.d/nfs-kernel-server  restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 194 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>: <your_gatewayip>:<your_netmask>:<board_name>:<ethx>:off
console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel FIT image to the board, then boot it up.

```
=>tftp a0000000  kernel.itb;
=>bootm a0000000:kernel@1 - a0000000:fdt@1
```

   Now the board will boot up with NFS filesystem.

4.4.8.9.4 SD Deployment

Partition SD Card

1. Insert SD card into the Linux Host PC.
2. Use the "fdisk" command to repartition the SD card.

```
# fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to creat the filesystem.

```
#mkfs.ext2 /dev/sdb1
```

NOTE

The first 2056 sectors of SD card must be remained for u-boot image

FIT Kernel Image and Root File System Deployment from SD Card

1. Insert SD card into the Linux Host PC.

2. Create temp director in host PC and mount the ext2 partition to the temp

```
#mkdir temp
#mount /dev/sdb1 temp
```

3. Copy the FIT Kernel Image to the SD card partition.

```
#cp  kernel.itb temp/
```

4. Copy the Root File System to the SD card partition.

```
#cp fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz  temp/
#tar xvfz fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz
#rm  fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz  temp/
```

5. Umount the temp director

```
#umount temp
```

U-Boot Image Deployment from SD Card

- Form Linux Host PC
  1. Insert SD card into Host.
  2. Use the "dd" command

```
# dd if=u-boot-with-spl-pbl.bin of=/dev/sdb  seek=8 bs=512
```

- Form ls1043ardb Board
  1. Insert SD card into target board and power on.
  2. Programming U-boot image to SD Card

```
=> tftpboot 82000000 u-boot-with-spl-pbl.bin
=> mmc write 82000000 8 800
=> cpld reset sd
```

Setting U-Boot Environment

- Execute the following commands at the U-Boot prompt

```
=> setenv bootcmd "ext2load mmc 0 a0000000 kernel.itb && bootm a0000000"
```

- Using the Ramdisk as the Root File Systemp

```
=> setenv bootargs  "root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200"
```

- Using the Ext2 Partition of SD card as the Root File Systemp

```
=> setenv bootargs  "root=/dev/mmcblk0p1 rw earlycon=uart8250,0x21c0500 console=ttyS0,115200"
```

- Saving the environment

```
=>saveenv
```

NOTE

The `kernel.itb` is the name of your FIT Image, you can use the `ext2ls` command to list it at the U-Boot prompt

4.4.8.9.5 QSPI Deployment

Build U-Boot image for QSPI without Yocto Project

The U-Boot image generated from code needs to be swapped for QSPI boot. Yocto generates the swapped U-Boot image, which is programmed into QSPI flash. Below are steps to build U-Boot without Yocto:

1. Compile QSPI boot image(enable QSPI):

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/bin/aarch64-linux-gnu-
```

```
$make ARCH=aarch64 ls1046ardb_qspi_defconfig
```

```
$make CROSS_COMPILE=${toolchain_path}/bin/aarch64-linux-gnu- -j4
```

2. Swap the bytes for QSPI boot:

```
$tclsh byte_swap.tcl rcw_1600_qspiboot.bin rcw_1600_qspiboot_swap.bin 8
```

   The byte_swap.tcl script is a shareable tool and can be found under rcw/tool/ directory.

3. Switch to QSPI altbank:

```
=>cpld reset altbank
```

   Or set switches referring to board configurations Switch Settings and power on the board from QSPI boot.

Build Linux image and FIT image for QSPI boot

1. Compile Linux kernel image:

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/bin/aarch64-linux-gnu-
```

```
$make ARCH=arm64  $make defconfig freescale.config
```

```
$make CROSS_COMPILE=${toolchain_path}/bin/aarch64-linux-gnu- -j4
```

2. Make it into kernel.itb:

```
$mkimage -f kernel-ls1046a-rdb.its kernel.itb
```

To boot kernel, please refer to FIT Image Deployment from TFTP and FIT Image Deployment from Flash.

4.4.8.10 Check 'Link Up' for Serial Ethernet Interfaces

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
1 - RealTek RTL8211F <--> FM1@DTSEC3
2 - RealTek RTL8211F <--> FM1@DTSEC4
3 - RealTek RTL8211F <--> FM1@DTSEC5
4 - RealTek RTL8211F <--> FM1@DTSEC6
FM_TGEC_MDIO:
0 - Aquantia AQR107 <--> FM1@TGEC1
```

The results from the above *mdio list* command show that U-Boot was able to see PHYs on each of the 4 DTSEC interfaces and on the 10GEC interface. If you see “Generic” reported, it is an indication that something is there but the ls1046ardb can’t communicate with the device/port.

Check Link Status for External PHY

In order to check the status of a SGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read FM1@DTSEC3 1
Reading from bus FSL_MDIO0
PHY at address 1:
1 - 0x79ad
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x79a9.

4.4.8.11 Basic Networking Ping Test

U-BOOT

The LS1046ARDB has two RGMII ports and two SGMII ports. The log below shows how to ping from FM1@DTSEC3 and FM1@DTSEC4 interfaces.

```
U-Boot 2016.01 (Aug 19 2016 - 16:59:27 +0800)

SoC:  LS1046E (0x87070010)
Clock Configuration:
       CPU0(A72):1600 MHz  CPU1(A72):1600 MHz  CPU2(A72):1600 MHz
       CPU3(A72):1600 MHz
       Bus:      600  MHz  DDR:      2100 MT/s  FMAN:     700  MHz
Reset Configuration Word (RCW):
       00000000: 0c150010 0e000000 00000000 00000000
       00000010: 11335559 40005012 40025000 c1000000
       00000020: 00000000 00000000 00000000 00238800
       00000030: 20124000 00003101 00000096 00000001
I2C:   ready
Model: LS1046A RDB Board
Board: LS1046ARDB, boot from QSPI vBank 0
CPLD:  V2.2
PCBA:  V2.0
```

```
SERDES Reference Clocks:
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
DRAM:  Initializing DDR....using SPD
Detected UDIMM 18ASF1G72AZ-2G3B1
8 GiB (DDR4, 64-bit, CL=15, ECC on)
       DDR Chip-Select Interleaving Mode: CS0+CS1
SEC0: RNG instantiated
PPA Firmware: Version 0.2
Using SERDES1 Protocol: 4403 (0x1133)
Using SERDES2 Protocol: 21849 (0x5559)
NAND:  512 MiB
MMC:   FSL_SDHC: 0
SF: Detected S25FL512S_256K with page size 256 Bytes, erase size 256 KiB, total 64 MiB
EEPROM: Invalid ID (5a 5a 5a 5a)
PCIe1: Root Complex no link, regs @ 0x3400000
PCIe2: Root Complex no link, regs @ 0x3500000
PCIe3: Root Complex no link, regs @ 0x3600000
In:    serial
Out:   serial
Err:   serial
SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI:  Net:   SF: Detected S25FL512S_256K with page size 256 Bytes, erase size 256 KiB, total 64 MiB
Fman1: Uploading microcode version 106.4.15
FM1@DTSEC3 [PRIME], FM1@DTSEC4, FM1@DTSEC5, FM1@DTSEC6, FM1@TGEC1, FM1@TGEC2
Hit any key to stop autoboot:  0
=> setenv serverip 10.192.208.233
=> setenv ipaddr 10.193.20.129
=> setenv ethaddr 00:e0:0c:00:89:00
=> ping $serverip
Using FM1@DTSEC3 device
host 10.192.208.233 is alive
=> setenv ethact FM1@DTSEC4
=> setenv ethaddr1 00:e0:0c:00:89:01
=> ping $serverip
Using FM1@DTSEC4 device
host 10.192.208.233 is alive
```

LINUX

Bring the kernel-ls1046a-rdb.itb via FMan interface to boot up Linux kernel:

```
=> tftp a0000000 kernel-ls1046a-rdb.itb;bootm a0000000
Using FM1@DTSEC3 device
TFTP from server 10.192.208.233; our IP address is 10.193.20.129
Filename 'kernel-ls1046a-rdb.itb'.
Load address: 0xa0000000
Loading: ################################################################
         ################################################################
         ################################################################
         ################################################################
         ################################################################
         ################################################################
         ################################################################
         ################################################################
         ################################################################
         ################################################################
```

```
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ################################
         2.5 MiB/s
done
Bytes transferred = 27185703 (19ed227 hex)
## Loading kernel from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'kernel@1' kernel subimage
     Description:  ARM64 Linux kernel
     Type:         Kernel Image
     Compression:  gzip compressed
     Data Start:   0xa00000dc
     Data Size:    5473947 Bytes = 5.2 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x80080000
     Entry Point:  0x80080000
   Verifying Hash Integrity ... OK
## Loading ramdisk from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'ramdisk@1' ramdisk subimage
     Description:  LS1046 Ramdisk
     Type:         RAMDisk Image
     Compression:  gzip compressed
     Data Start:   0xa053e5b8
     Data Size:    21686563 Bytes = 20.7 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: unavailable
     Entry Point:  unavailable
   Verifying Hash Integrity ... OK
## Loading fdt from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'fdt@1' fdt subimage
     Description:  Flattened Device Tree blob
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0xa053882c
     Data Size:    23809 Bytes = 23.3 KiB
     Architecture: AArch64
   Verifying Hash Integrity ... OK
   Loading fdt from 0xa053882c to 0x90000000
   Booting using the fdt blob at 0x90000000
```

```
   Uncompressing Kernel Image ... OK
   Using Device Tree in place at 0000000090000000, end 0000000090018d00

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 4.1.8-gdce5972 (gqy@titan) (gcc version 4.9.1 20140529 (prerelease)
(crosstool-NG linaro-1.13.1-4.9-2014.07 - Linaro GCC 4.9-2014.06) ) #12 SMP PREEMPT Wed Aug 24
12:21:27 CST 2016
[    0.000000] CPU: AArch64 Processor [410fd082] revision 2
[    0.000000] Detected PIPT I-cache on CPU0
[    0.000000] earlycon: Early serial console at MMIO 0x21c0500 (options '')
[    0.000000] bootconsole [uart0] enabled
[    0.000000] efi: Getting EFI parameters from FDT:
[    0.000000] efi: UEFI not found.
[    0.000000] Reserved memory: initialized node bman-fbpr, compatible id fsl,bman-fbpr
[    0.000000] Reserved memory: initialized node qman-fqd, compatible id fsl,qman-fqd
[    0.000000] Reserved memory: initialized node qman-pfdr, compatible id fsl,qman-pfdr
[    0.000000] cma: Reserved 16 MiB at 0x00000000ff000000
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv0.2 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] PERCPU: Embedded 18 pages/cpu @ffff80097fd6d000 s33664 r8192 d31872 u73728
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 2063880
[    0.000000] Kernel command line: console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,
0x21c0500
[    0.000000] log_buf_len individual max cpu contribution: 4096 bytes
[    0.000000] log_buf_len total cpu_extra contributions: 12288 bytes
[    0.000000] log_buf_len min size: 16384 bytes
[    0.000000] log_buf_len: 32768 bytes
[    0.000000] early log buf free: 14320(87%)
[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
[    0.000000] Dentry cache hash table entries: 1048576 (order: 11, 8388608 bytes)
[    0.000000] Inode-cache hash table entries: 524288 (order: 10, 4194304 bytes)
[    0.000000] software IO TLB [mem 0xfafff000-0xfefff000] (64MB) mapped at [ffff80007afff000-
ffff80007effefff]
[    0.000000] Memory: 8068852K/8386560K available (7754K kernel code, 582K rwdata, 3092K rodata,
476K init, 756K bss, 301324K reserved, 16384K cma-reserved)
[    0.000000] Virtual kernel memory layout:
[    0.000000]     vmalloc : 0xffff000000000000 - 0xffff7bffbfff0000   (126974 GB)
[    0.000000]     vmemmap : 0xffff7bffc0000000 - 0xffff7fffc0000000   (  4096 GB maximum)
[    0.000000]               0xffff7bffc2000000 - 0xffff7bffe7ff8000   (   607 MB actual)
[    0.000000]     fixed   : 0xffff7ffffabfd000 - 0xffff7ffffac00000   (    12 KB)
[    0.000000]     PCI I/O : 0xffff7ffffae00000 - 0xffff7ffffbe00000   (    16 MB)
[    0.000000]     modules : 0xffff7ffffc000000 - 0xffff800000000000   (    64 MB)
[    0.000000]     memory  : 0xffff800000000000 - 0xffff80097fe00000   ( 38910 MB)
[    0.000000]       .init : 0xffff800000b1a000 - 0xffff800000b91000   (   476 KB)
[    0.000000]       .text : 0xffff800000080000 - 0xffff800000b19974   ( 10855 KB)
[    0.000000]       .data : 0xffff800000ba2000 - 0xffff800000c33800   (   582 KB)
[    0.000000] Preemptible hierarchical RCU implementation.
[    0.000000]  Additional per-CPU info printed with stalls.
[    0.000000]  RCU restricting CPUs from NR_CPUS=64 to nr_cpu_ids=4.
[    0.000000] RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
[    0.000000] NR_IRQS:64 nr_irqs:64 0
[    0.000000] Architected cp15 timer(s) running at 25.00MHz (phys).
[    0.000000] clocksource arch_sys_counter: mask: 0xffffffffffffff max_cycles: 0x5c40939b5,
max_idle_ns: 440795202646 ns
[    0.000002] sched_clock: 56 bits at 25MHz, resolution 40ns, wraps every 4398046511100ns
[    0.008237] Console: colour dummy device 80x25
```

```
[    0.012708] Calibrating delay loop (skipped), value calculated using timer frequency.. 50.00
BogoMIPS (lpj=250000)
[    0.023127] pid_max: default: 32768 minimum: 301
[    0.027809] Security Framework initialized
[    0.031948] Mount-cache hash table entries: 16384 (order: 5, 131072 bytes)
[    0.038863] Mountpoint-cache hash table entries: 16384 (order: 5, 131072 bytes)
[    0.046512] Initializing cgroup subsys memory
[    0.050902] Initializing cgroup subsys hugetlb
[    0.055451] hw perfevents: enabled with arm/armv8-pmuv3 PMU driver, 7 counters available
[    0.063605] EFI services will not be available.
[    0.143758] CPU1: Booted secondary processor
[    0.143760] Detected PIPT I-cache on CPU1
[    0.163758] CPU2: Booted secondary processor
[    0.163761] Detected PIPT I-cache on CPU2
[    0.183767] CPU3: Booted secondary processor
[    0.183769] Detected PIPT I-cache on CPU3
[    0.183802] Brought up 4 CPUs
[    0.211728] SMP: Total of 4 processors activated.
[    0.216455] CPU: All CPU(s) started at EL2
[    0.220751] devtmpfs: initialized
[    0.226415] DMI not present or invalid.
[    0.230400] clocksource jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns:
19112604462750000 ns
[    0.240504] pinctrl core: initialized pinctrl subsystem
[    0.246350] NET: Registered protocol family 16
[    0.280834] cpuidle: using governor ladder
[    0.310840] cpuidle: using governor menu
[    0.314830] fsl-mc bus type registered
[    0.318648] MC object device driver fsl_mc_dprc registered
[    0.324196] MC object device driver fsl_mc_allocator registered
[    0.330165] Bman ver:0a02,02,01
[    0.335432] qman-fqd addr 0x9ff000000 size 0x800000
[    0.340335] qman-pfdr addr 0x9fc000000 size 0x2000000
[    0.345418] Qman ver:0a01,03,02,01
[    0.348875] vdso: 2 pages (1 code @ ffff800000ba9000, 1 data @ ffff800000ba8000)
[    0.356336] hw-breakpoint: found 6 breakpoint and 4 watchpoint registers.
[    0.363480] DMA: preallocated 256 KiB pool for atomic allocations
[    0.369707] Serial: AMBA PL011 UART driver
[    0.410976] vgaarb: loaded
[    0.413838] SCSI subsystem initialized
[    0.417925] usbcore: registered new interface driver usbfs
[    0.423486] usbcore: registered new interface driver hub
[    0.428863] usbcore: registered new device driver usb
[    0.434393] i2c i2c-0: IMX I2C adapter registered
[    0.439131] i2c i2c-0: can't use DMA
[    0.442853] i2c i2c-1: IMX I2C adapter registered
[    0.447588] i2c i2c-1: can't use DMA
[    0.451279] pps_core: LinuxPPS API ver. 1 registered
[    0.456269] pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti
[    0.465487] PTP clock support registered
[    0.469599] bman-fbpr addr 0x9fe000000 size 0x1000000
[    0.474692] Bman err interrupt handler present
[    0.479444] Bman portal initialised, cpu 0
[    0.483625] Bman portal initialised, cpu 1
[    0.487806] Bman portal initialised, cpu 2
[    0.491993] Bman portal initialised, cpu 3
[    0.496108] Bman portals initialised
[    0.500587] Qman err interrupt handler present
[    0.505348] QMan: Allocated lookup table at ffff0000001cd000, entry count 131073
[    0.646309] Qman portal initialised, cpu 0
```

```
[    0.650890] Qman portal initialised, cpu 1
[    0.655466] Qman portal initialised, cpu 2
[    0.660039] Qman portal initialised, cpu 3
[    0.664158] Qman portals initialised
[    0.667803] Bman: BPID allocator includes range 32:32
[    0.672902] Qman: FQID allocator includes range 256:256
[    0.678158] Qman: FQID allocator includes range 32768:32768
[    0.683784] Qman: CGRID allocator includes range 0:256
[    0.689085] Qman: pool channel allocator includes range 1025:15
[    0.695072] No USDPAA memory, no 'fsl,usdpaa-mem' in device-tree
[    0.701144] fsl-ifc 1530000.ifc: Freescale Integrated Flash Controller
[    0.707716] fsl-ifc 1530000.ifc: IFC version 1.4, 8 banks
[    0.713576] Switched to clocksource arch_sys_counter
[    0.721658] NET: Registered protocol family 2
[    0.726268] TCP established hash table entries: 65536 (order: 7, 524288 bytes)
[    0.733717] TCP bind hash table entries: 65536 (order: 8, 1048576 bytes)
[    0.740989] TCP: Hash tables configured (established 65536 bind 65536)
[    0.747612] UDP hash table entries: 4096 (order: 5, 131072 bytes)
[    0.753781] UDP-Lite hash table entries: 4096 (order: 5, 131072 bytes)
[    0.760467] NET: Registered protocol family 1
[    0.764943] RPC: Registered named UNIX socket transport module.
[    0.770896] RPC: Registered udp transport module.
[    0.775636] RPC: Registered tcp transport module.
[    0.780363] RPC: Registered tcp NFSv4.1 backchannel transport module.
[    0.786923] Trying to unpack rootfs image as initramfs...
[    0.792505] rootfs image is not initramfs (no cpio magic); looks like an initrd
[    0.818311] Freeing initrd memory: 21172K (ffff80002053f000 - ffff8000219ec000)
[    0.826013] kvm [1]: interrupt-controller@1440000 IRQ9
[    0.831246] kvm [1]: timer IRQ3
[    0.834412] kvm [1]: Hyp mode initialized successfully
[    0.840599] futex hash table entries: 1024 (order: 4, 65536 bytes)
[    0.846867] audit: initializing netlink subsys (disabled)
[    0.852307] audit: type=2000 audit(0.720:1): initialized
[    0.857853] HugeTLB registered 2 MB page size, pre-allocated 0 pages
[    0.864507] VFS: Disk quotas dquot_6.6.0
[    0.868469] VFS: Dquot-cache hash table entries: 512 (order 0, 4096 bytes)
[    0.875691] NFS: Registering the id_resolver key type
[    0.880784] Key type id_resolver registered
[    0.885033] Key type id_legacy registered
[    0.889127] fuse init (API version 7.23)
[    0.893182] 9p: Installing v9fs 9p2000 file system support
[    0.899088] io scheduler noop registered
[    0.903057] io scheduler cfq registered (default)
[    0.908251] find msi-controller /soc/msi-controller@1580000
[    0.914857] PCI host bridge /soc/pcie@3400000 ranges:
[    0.919941]    IO 0x4000010000..0x400001ffff -> 0x00000000
[    0.925468]   MEM 0x4040000000..0x407fffffff -> 0x40000000
[    0.931053] layerscape-pcie 3400000.pcie: PCI host bridge to bus 0000:00
[    0.937801] pci_bus 0000:00: root bus resource [bus 00-ff]
[    0.943317] pci_bus 0000:00: root bus resource [io  0x0000-0xffff]
[    0.949561] pci_bus 0000:00: root bus resource [mem 0x4040000000-0x407fffffff] (bus address
[0x40000000-0x7fffffff])
[    0.960317] pci 0000:00:00.0: bridge configuration invalid ([bus 00-00]), reconfiguring
[    0.968438] pci 0000:00:00.0: BAR 1: assigned [mem 0x4040000000-0x4043ffffff]
[    0.975624] pci 0000:00:00.0: BAR 0: assigned [mem 0x4044000000-0x4044ffffff]
[    0.982802] pci 0000:00:00.0: BAR 6: assigned [mem 0x4045000000-0x4045ffffff pref]
[    0.990426] pci 0000:00:00.0: PCI bridge to [bus 01]
[    0.995441] pcieport 0000:00:00.0: enabling device (0000 -> 0002)
[    1.001640] pcieport 0000:00:00.0: Signaling PME through PCIe PME interrupt
[    1.008818] PCI host bridge /soc/pcie@3500000 ranges:
```

```
[    1.013907]    IO 0x4800010000..0x480001ffff -> 0x00000000
[    1.019424]   MEM 0x4840000000..0x487fffffff -> 0x40000000
[    1.025007] layerscape-pcie 3500000.pcie: PCI host bridge to bus 0001:00
[    1.031747] pci_bus 0001:00: root bus resource [bus 00-ff]
[   1.037270] pci_bus 0001:00: root bus resource [io  0x10000-0x1ffff] (bus address [0x0000-0xffff])
[    1.046289] pci_bus 0001:00: root bus resource [mem 0x4840000000-0x487fffffff] (bus address
[0x40000000-0x7fffffff])
[    1.057026] pci 0001:00:00.0: bridge configuration invalid ([bus 00-00]), reconfiguring
[    1.065142] pci 0001:00:00.0: BAR 1: assigned [mem 0x4840000000-0x4843ffffff]
[    1.072321] pci 0001:00:00.0: BAR 0: assigned [mem 0x4844000000-0x4844ffffff]
[    1.079505] pci 0001:00:00.0: BAR 6: assigned [mem 0x4845000000-0x4845ffffff pref]
[    1.087125] pci 0001:00:00.0: PCI bridge to [bus 01]
[    1.092131] pcieport 0001:00:00.0: enabling device (0000 -> 0002)
[    1.098334] pcieport 0001:00:00.0: Signaling PME through PCIe PME interrupt
[    1.105508] PCI host bridge /soc/pcie@3600000 ranges:
[    1.110592]    IO 0x5000010000..0x500001ffff -> 0x00000000
[    1.116115]   MEM 0x5040000000..0x507fffffff -> 0x40000000
[    1.121692] layerscape-pcie 3600000.pcie: PCI host bridge to bus 0002:00
[    1.128458] pci_bus 0002:00: root bus resource [bus 00-ff]
[   1.133979] pci_bus 0002:00: root bus resource [io  0x20000-0x2ffff] (bus address [0x0000-0xffff])
[    1.142994] pci_bus 0002:00: root bus resource [mem 0x5040000000-0x507fffffff] (bus address
[0x40000000-0x7fffffff])
[    1.153738] pci 0002:00:00.0: bridge configuration invalid ([bus 00-00]), reconfiguring
[    1.161846] pci 0002:00:00.0: BAR 1: assigned [mem 0x5040000000-0x5043ffffff]
[    1.169032] pci 0002:00:00.0: BAR 0: assigned [mem 0x5044000000-0x5044ffffff]
[    1.176215] pci 0002:00:00.0: BAR 6: assigned [mem 0x5045000000-0x5045ffffff pref]
[    1.183834] pci 0002:00:00.0: PCI bridge to [bus 01]
[    1.188841] pcieport 0002:00:00.0: enabling device (0000 -> 0002)
[    1.195046] pcieport 0002:00:00.0: Signaling PME through PCIe PME interrupt
[    1.202340] Freescale LS2 console driver
[    1.206336] fsl-ls2-console: device fsl_mc_console registered
[    1.212151] fsl-ls2-console: device fsl_aiop_console registered
[    1.219546] Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
[    1.226540] msm_serial: driver initialized
[    1.230858] console [ttyS0] disabled
[    1.234470] 21c0500.serial: ttyS0 at MMIO 0x21c0500 (irq = 18, base_baud = 18750000) is a 16550A
[    1.243320] console [ttyS0] enabled
[    1.243320] console [ttyS0] enabled
[    1.250309] bootconsole [uart0] disabled
[    1.250309] bootconsole [uart0] disabled
[    1.258337] 21c0600.serial: ttyS1 at MMIO 0x21c0600 (irq = 18, base_baud = 18750000) is a 16550A
[    1.267291] 21d0500.serial: ttyS2 at MMIO 0x21d0500 (irq = 19, base_baud = 18750000) is a 16550A
[    1.276245] 21d0600.serial: ttyS3 at MMIO 0x21d0600 (irq = 19, base_baud = 18750000) is a 16550A
[    1.288701] brd: module loaded
[    1.293380] loop: module loaded
[    1.296585] at24 0-0052: 65536 byte 24c512 EEPROM, writable, 1 bytes/write
[    1.303471] at24 0-0053: 65536 byte 24c512 EEPROM, writable, 1 bytes/write
[   1.310790] ahci-qoriq 3200000.sata: AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl platform mode
[   1.319760] ahci-qoriq 3200000.sata: flags: 64bit ncq sntf pm clo only pmp fbs pio slum part ccc
sds apst
[    1.329738] scsi host0: ahci-qoriq
[    1.333229] ata1: SATA max UDMA/133 mmio [mem 0x03200000-0x0320ffff] port 0x100 irq 30
[    1.341947] nand: device found, Manufacturer ID: 0x2c, Chip ID: 0xac
[    1.348311] nand: Micron MT29F4G08ABBEAH4
[    1.352315] nand: 512 MiB, SLC, erase size: 256 KiB, page size: 4096, OOB size: 224
[    1.360623] Bad block table found at page 131008, version 0x01
[    1.367676] Bad block table found at page 130944, version 0x01
[    1.374551] fsl,ifc-nand 7e800000.nand: IFC NAND device at 0x7e800000, bank 0
[    1.381947] fsl-quadspi 1550000.quadspi: s25fs512s (65536 Kbytes)
[    1.388390] fsl-quadspi 1550000.quadspi: s25fs512s (65536 Kbytes)
```

```
[    1.395613] libphy: Fixed MDIO Bus: probed
[    1.399790] tun: Universal TUN/TAP device driver, 1.6
[    1.404846] tun: (C) 1999-2004 Max Krasnyansky
[    1.411315] libphy: Freescale XGMAC MDIO Bus: probed
[    1.417174] libphy: Freescale XGMAC MDIO Bus: probed
[    1.423299] libphy: Freescale XGMAC MDIO Bus: probed
[    1.428336] libphy: Freescale XGMAC MDIO Bus: probed
[    1.433361] libphy: Freescale XGMAC MDIO Bus: probed
[    1.438394] libphy: Freescale XGMAC MDIO Bus: probed
[    1.443419] libphy: Freescale XGMAC MDIO Bus: probed
[    1.448451] libphy: Freescale XGMAC MDIO Bus: probed
[    1.453478] libphy: Freescale XGMAC MDIO Bus: probed
[    1.458505] libphy: Freescale XGMAC MDIO Bus: probed
[    1.473182] Freescale FM module, FMD API version 21.1.0
[    1.480715] Freescale FM Ports module
[    1.484382] fsl_mac: fsl_mac: FSL FMan MAC API based driver
[    1.490043] fsl_mac 1ae4000.ethernet: FMan MEMAC
[    1.494665] fsl_mac 1ae4000.ethernet: FMan MAC address: 00:00:00:00:00:03
[    1.501503] fsl_mac 1ae6000.ethernet: FMan MEMAC
[    1.506123] fsl_mac 1ae6000.ethernet: FMan MAC address: 00:00:00:00:00:04
[    1.513007] fsl_mac 1ae8000.ethernet: FMan MEMAC
[    1.517627] fsl_mac 1ae8000.ethernet: FMan MAC address: 00:00:00:00:00:05
[    1.524512] fsl_mac 1aea000.ethernet: FMan MEMAC
[    1.529127] fsl_mac 1aea000.ethernet: FMan MAC address: 00:00:00:00:00:06
[    1.535973] fsl_mac 1af0000.ethernet: FMan MEMAC
[    1.540588] fsl_mac 1af0000.ethernet: FMan MAC address: 00:00:00:00:00:07
[    1.547547] fsl_mac 1af2000.ethernet: FMan MEMAC
[    1.552163] fsl_mac 1af2000.ethernet: FMan MAC address: 00:00:00:00:00:08
[    1.559026] fsl_dpa: FSL DPAA Ethernet driver
[    1.563471] fsl_dpa fsl,dpaa:ethernet@0: of_find_device_by_node(/soc/fman@1a00000/
ethernet@e0000) failed
[    1.572963] fsl_dpa: probe of fsl,dpaa:ethernet@0 failed with error -22
[    1.579616] fsl_dpa fsl,dpaa:ethernet@1: of_find_device_by_node(/soc/fman@1a00000/
ethernet@e2000) failed
[    1.589106] fsl_dpa: probe of fsl,dpaa:ethernet@1 failed with error -22
[    1.600629] fsl_dpa: fsl_dpa: Probed interface eth0
[    1.610449] fsl_dpa: fsl_dpa: Probed interface eth1
[    1.620543] fsl_dpa: fsl_dpa: Probed interface eth2
[    1.630908] fsl_dpa: fsl_dpa: Probed interface eth3
[    1.641565] fsl_dpa: fsl_dpa: Probed interface eth4
[    1.652419] fsl_dpa: fsl_dpa: Probed interface eth5
[    1.657362] fsl_advanced: FSL DPAA Advanced drivers:
[    1.662324] fsl_proxy: FSL DPAA Proxy initialization driver
[    1.668030] fsl_dpa_shared: FSL DPAA Shared Ethernet driver
[    1.673688] fsl_dpa_macless: FSL DPAA MACless Ethernet driver
[    1.679513] fsl_oh: FSL FMan Offline Parsing port driver
[    1.683593] ata1: SATA link down (SStatus 0 SControl 300)
[    1.690334] e1000: Intel(R) PRO/1000 Network Driver - version 7.3.21-k8-NAPI
[    1.697385] e1000: Copyright (c) 1999-2006 Intel Corporation.
[    1.703161] e1000e: Intel(R) PRO/1000 Network Driver - 2.3.2-k
[    1.708994] e1000e: Copyright(c) 1999 - 2014 Intel Corporation.
[    1.714940] sky2: driver version 1.30
[    1.718848] VFIO - User Level meta-driver version: 0.3
[    1.724110] vfio_fsl_mc_driver_init: Driver registration fails as no fsl_mc_bus found
[    2.932972] xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
[    2.938470] xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 1
[    2.946283] xhci-hcd xhci-hcd.0.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x00010010
[    2.955006] xhci-hcd xhci-hcd.0.auto: irq 27, io mem 0x02f00000
[    2.961164] hub 1-0:1.0: USB hub found
[    2.964927] hub 1-0:1.0: 1 port detected
```

```
[    2.968952] xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
[    2.974445] xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 2
[    2.982121] usb usb2: We don't know the algorithms for LPM for this host, disabling LPM.
[    2.990427] hub 2-0:1.0: USB hub found
[    2.994184] hub 2-0:1.0: 1 port detected
[    2.998220] xhci-hcd xhci-hcd.1.auto: xHCI Host Controller
[    3.003713] xhci-hcd xhci-hcd.1.auto: new USB bus registered, assigned bus number 3
[    3.011513] xhci-hcd xhci-hcd.1.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x00010010
[    3.020231] xhci-hcd xhci-hcd.1.auto: irq 28, io mem 0x03000000
[    3.026366] hub 3-0:1.0: USB hub found
[    3.030117] hub 3-0:1.0: 1 port detected
[    3.034144] xhci-hcd xhci-hcd.1.auto: xHCI Host Controller
[    3.039629] xhci-hcd xhci-hcd.1.auto: new USB bus registered, assigned bus number 4
[    3.047311] usb usb4: We don't know the algorithms for LPM for this host, disabling LPM.
[    3.055613] hub 4-0:1.0: USB hub found
[    3.059364] hub 4-0:1.0: 1 port detected
[    3.063400] xhci-hcd xhci-hcd.2.auto: xHCI Host Controller
[    3.068920] xhci-hcd xhci-hcd.2.auto: new USB bus registered, assigned bus number 5
[    3.076727] xhci-hcd xhci-hcd.2.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x00010010
[    3.085446] xhci-hcd xhci-hcd.2.auto: irq 29, io mem 0x03100000
[    3.091579] hub 5-0:1.0: USB hub found
[    3.095366] hub 5-0:1.0: 1 port detected
[    3.099384] xhci-hcd xhci-hcd.2.auto: xHCI Host Controller
[    3.104876] xhci-hcd xhci-hcd.2.auto: new USB bus registered, assigned bus number 6
[    3.112552] usb usb6: We don't know the algorithms for LPM for this host, disabling LPM.
[    3.120850] hub 6-0:1.0: USB hub found
[    3.124629] hub 6-0:1.0: 1 port detected
[    3.128663] ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
[    3.135220] ehci-pci: EHCI PCI platform driver
[    3.139684] ehci-platform: EHCI generic platform driver
[    3.145029] ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
[    3.151211] ohci-pci: OHCI PCI platform driver
[    3.155707] ohci-platform: OHCI generic platform driver
[    3.161061] usbcore: registered new interface driver usb-storage
[    3.167189] mousedev: PS/2 mouse device common for all mice
[    3.172941] rtc-pcf2127 1-0051: chip found, driver version 0.0.1
[    3.181682] rtc-pcf2127 1-0051: rtc core: registered rtc-pcf2127 as rtc0
[    3.188479] i2c /dev entries driver
[    3.194275] ina2xx 0-0040: power monitor ina220 (Rshunt = 1000 uOhm)
[    3.200675] 0-004c supply vcc not found, using dummy regulator
[    3.209254] imx2-wdt 2ad0000.wdog: timeout 60 sec (nowayout=0)
[    3.215370] sdhci: Secure Digital Host Controller Interface driver
[    3.221548] sdhci: Copyright(c) Pierre Ossman
[    3.225958] sdhci-pltfm: SDHCI platform and OF driver helper
[    3.231727] sdhci-esdhc 1560000.esdhc: No vmmc regulator found
[    3.237565] sdhci-esdhc 1560000.esdhc: No vqmmc regulator found
[    3.283583] mmc0: SDHCI controller on 1560000.esdhc [1560000.esdhc] using ADMA 64-bit
[    3.292162] platform caam_qi.0: Linux CAAM Queue I/F driver initialised
[    3.298788] caam 1700000.crypto: Instantiated RNG4 SH1
[    3.303933] caam 1700000.crypto: device ID = 0x0a11030100000000 (Era 8)
[    3.310546] caam 1700000.crypto: job rings = 4, qi = 1
[    3.317314] caam algorithms registered in /proc/crypto
[    3.322993] platform caam_qi.0: fsl,sec-v5.4 algorithms registered in /proc/crypto
[    3.331427] caam_jr 1710000.jr: registering rng-caam
[    3.336493] caam 1700000.crypto: fsl,sec-v5.4 algorithms registered in /proc/crypto
[    3.344195] MC object device driver fsl_dpaa2_caam registered
[    3.350438] usbcore: registered new interface driver usbhid
[    3.356101] usbhid: USB HID core driver
[    3.359999] fsl-mc bus not found, restool driver registration failed
[    3.366378] MC object device driver fsl_dpio_drv registered
```

```
[    3.372405] Freescale USDPAA process driver
[    3.376588] fsl-usdpaa: no region found
[    3.380419] Freescale USDPAA process IRQ driver
[    3.385008] MC object device driver fsl_dpaa2_eth registered
[    3.390677] MC object device driver dpaa2_mac registered
[    3.396003] MC object device driver dpaa2_ethsw registered
[    3.401496] MC object device driver dpaa2_evb registered
[    3.406863] MC object device driver fsl_dce_api registered
[    3.412357] MC object device driver dpaa2_rtc registered
[    3.417768] Initializing XFRM netlink socket
[    3.422073] NET: Registered protocol family 10
[    3.426910] sit: IPv6 over IPv4 tunneling driver
[    3.431749] NET: Registered protocol family 17
[    3.436204] NET: Registered protocol family 15
[    3.440667] 8021q: 802.1Q VLAN Support v1.8
[    3.444877] 9pnet: Installing 9P2000 support
[    3.449173] Key type dns_resolver registered
[    3.453670] registered taskstats version 1
[    3.457893] fsl_generic: FSL DPAA Generic Ethernet driver
[    3.466054] rtc-pcf2127 1-0051: setting system clock to 2016-08-23 22:21:29 UTC (1471990889)
[    3.474584] fdt: not creating '/sys/firmware/fdt': CRC check failed
[    3.481260] RAMDISK: gzip image found at block 0
[    4.242385] VFS: Mounted root (ext2 filesystem) readonly on device 1:0.
[    4.249050] devtmpfs: mounted
[    4.252209] Freeing unused kernel memory: 476K (ffff800000b1a000 - ffff800000b91000)
[    4.259991] Freeing alternatives memory: 48K (ffff800000b91000 - ffff800000b9d000)
INIT: version 2.88 booting
Starting udev
[    4.365202] udevd[1890]: starting version 182
[    4.393090] fsl_dpa fsl,dpaa:ethernet@2 fm1-mac3: renamed from eth0
[    4.493818] fsl_dpa fsl,dpaa:ethernet@8 fm1-mac9: renamed from eth4
[    4.493848] udevd[1893]: renamed network interface eth0 to fm1-mac3
[    4.563841] fsl_dpa fsl,dpaa:ethernet@5 fm1-mac6: renamed from eth3
[    4.563870] udevd[1898]: renamed network interface eth4 to fm1-mac9
[    4.663783] fsl_dpa fsl,dpaa:ethernet@3 fm1-mac4: renamed from eth1
[    4.663809] udevd[1896]: renamed network interface eth3 to fm1-mac6
[    4.723709] fsl_dpa fsl,dpaa:ethernet@4 fm1-mac5: renamed from eth2
[    4.730024] udevd[1894]: renamed network interface eth1 to fm1-mac4
[    4.773732] udevd[1895]: renamed network interface eth2 to fm1-mac5
[    4.820204] random: dd urandom read with 3 bits of entropy available
Populating dev cache
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
INIT: Entering runlevel: 5un-postinsts exists during rc.d purge
Configuring network interfaces... eth0: ERROR while getting interface flags: No such device
Starting OpenBSD Secure Shell server: sshd
  generating ssh RSA key...
  generating ssh ECDSA key...
  generating ssh DSA key...
  generating ssh ED25519 key...
done.
Starting network benchmark server: netserver.
Starting system log daemon...0
Starting kernel log daemon...0

QorIQ SDK (FSL Reference Distro) 1.9 ls1043ardb /dev/ttyS0

ls1043ardb login: root

root@ls1043ardb:~# ifconfig
lo        Link encap:Local Loopback
```

```
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@ls1043ardb:~# ifconfig -a
eth5      Link encap:Ethernet  HWaddr 00:00:00:00:00:08
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Memory:1af2000-1af2fff

fm1-mac3  Link encap:Ethernet  HWaddr 00:00:00:00:00:03
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Memory:1ae4000-1ae4fff

fm1-mac4  Link encap:Ethernet  HWaddr 00:00:00:00:00:04
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Memory:1ae6000-1ae6fff

fm1-mac5  Link encap:Ethernet  HWaddr 00:00:00:00:00:05
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Memory:1ae8000-1ae8fff

fm1-mac6  Link encap:Ethernet  HWaddr 00:00:00:00:00:06
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Memory:1aea000-1aeafff

fm1-mac9  Link encap:Ethernet  HWaddr 00:00:00:00:00:07
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Memory:1af0000-1af0fff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
```

```
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

sit0      Link encap:UNSPEC  HWaddr 00-00-00-00-3A-30-30-30-00-00-00-00-00-00-00-00
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@ls1043ardb:~#
root@ls1043ardb:~# ifconfig fm1-mac3 10.193.20.129
root@ls1043ardb:~# ping 10.192.208.233
PING 10.192.208.233 (10.192.208.233) 56(84) bytes of data.
64 bytes from 10.192.208.233: icmp_seq=1 ttl=63 time=0.303 ms
64 bytes from 10.192.208.233: icmp_seq=2 ttl=63 time=0.314 ms
64 bytes from 10.192.208.233: icmp_seq=3 ttl=63 time=0.281 ms
64 bytes from 10.192.208.233: icmp_seq=4 ttl=63 time=0.319 ms
64 bytes from 10.192.208.233: icmp_seq=5 ttl=63 time=0.318 ms
64 bytes from 10.192.208.233: icmp_seq=6 ttl=63 time=0.289 ms
64 bytes from 10.192.208.233: icmp_seq=7 ttl=63 time=0.267 ms
64 bytes from 10.192.208.233: icmp_seq=8 ttl=63 time=0.259 ms
64 bytes from 10.192.208.233: icmp_seq=9 ttl=63 time=0.318 ms
^C
--- 10.192.208.233 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 7998ms
rtt min/avg/max/mdev = 0.259/0.296/0.319/0.027 ms
root@ls1043ardb:~#
```

4.4.9 LS2085ARDB/LS2088ARDB

4.4.9.1 Introduction

This guide describes how to deploy software images onto the LS2085ARDB/LS2088ARDB board. This board supports multiple SoCs within a family of pin-compatible SoCs. These include:

- LS2080A (not supported in this release)
- LS2085A (a non-production SoC)
- LS2088A
- LS2084A
- Future planned SoCs

The images to be deployed are produced by the SDK's built process (bitbake). They are also available in pre-built form on the "images" ISO supplied with the SDK.

U-Boot is a very flexible boot loader and supports many possible boot devices that can contain images. This guide's main focus is NOR flash and using U-Boot's Ethernet to program it.

4.4.9.2 LS2085ARDB/LS2088ARDB Board Introduction

The LS2085ARDB/LS2088ARDB (reference design board) with a LS2088A SoC is a high-performance computing, evaluation, and development platform.

Most are configured with a socket for the SoC .

4.4.9.2.1 Overview

The figure below shows the LS2085ARDB/LS2088ARDB without a case:

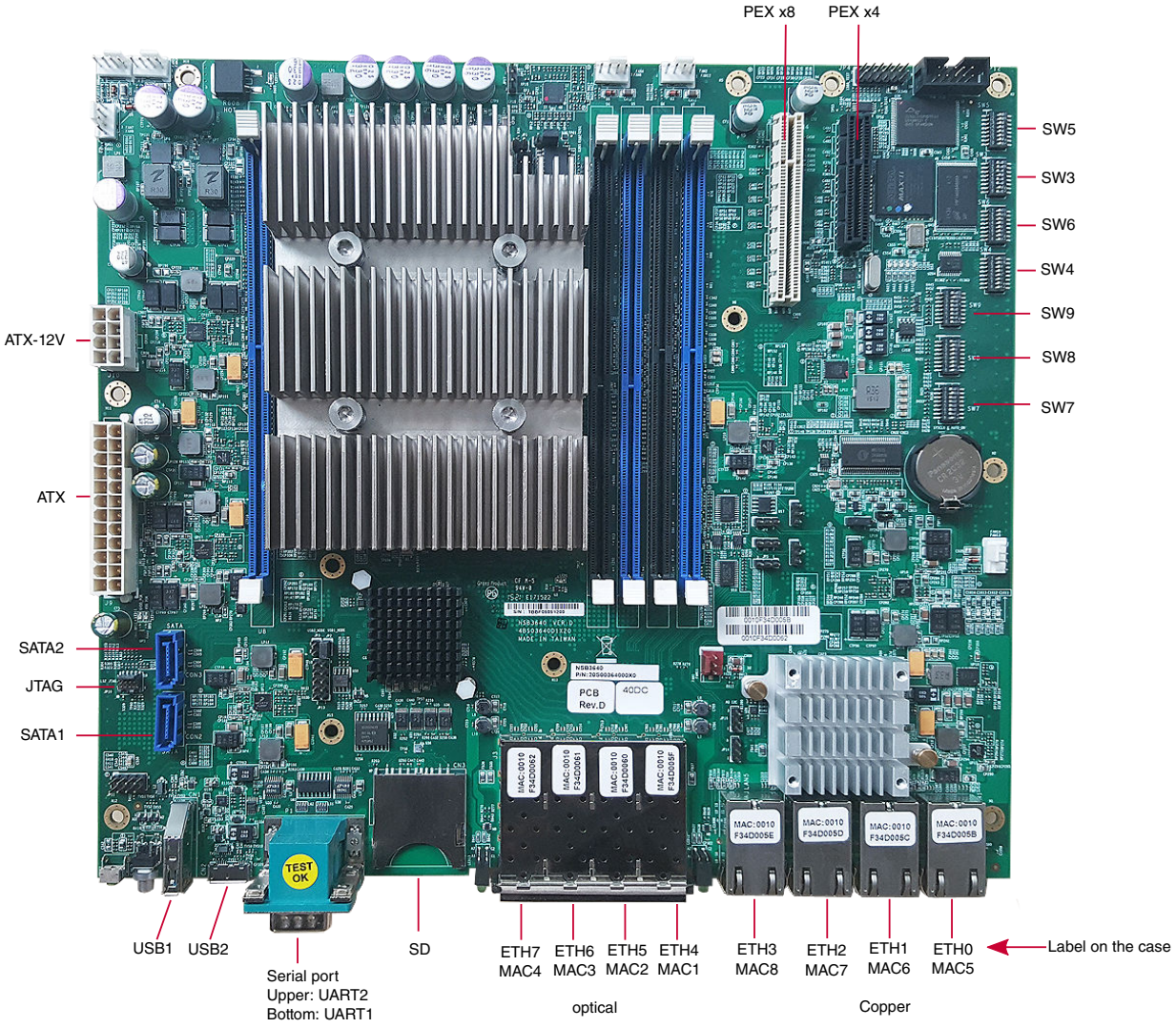


Figure 2. LS2085ARDB/LS2088ARDB

4.4.9.2.1.1 On-board front panel connection for 10G Ethernet

The RDB provides eight 10G interfaces using XFI SerDes protocols; four 10G interfaces are optical and the other four interfaces use copper connections.

The following figure shows the optical and copper interfaces available on the RDB. The optical ports are fitted with SFP modules and require fiber cables for connecting the interfaces. The copper interfaces take standard CAT6A/7 cables with RJ45 connectors.

The figure below labels the interfaces with their various identifiers. The DPMAC is the DPAA2 object that identifies the physical interface. As such, DPMAC1 to DPMAC8 represent the specific physical RDB interfaces shown below.

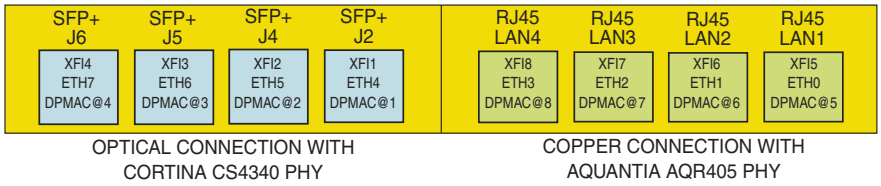


Figure 3. RDB optical and copper interfaces

When U-Boot starts, it prints the names of available interfaces using a syntax like "DPMAC5@xgmii" and "e1000#0" for the PCIe e1000 card. For example, suppose you plug a copper cable into the port labelled "DPMAC@5" in the figure. As described layer, you will set ethact in U-Boot to DPMAC5@xgmii. Note that the ETHX labels in the figure above are the physical labels on the chassis and are not the names of the interfaces in Linux. The interfaces in Linux are called niX.

The LS2088A SoC supports 80 Gbps total Ethernet bandwidth so all 8 ports can be used at the same time.

4.4.9.2.1.2 PCIe interface

The RDB board is normally provided with an e1000 PCIe Ethernet card fitted into a PCIe slot. This provides a demonstration of PCIe functionality and also provides a general purpose 1 Gbps copper Ethernet port that both Linux and U-Boot can use.

The PCIe NIC Ethernet card is connected to the board via an L-shaped adapter.

4.4.9.2.2 Board preparation

Use the serial port labeled UART2 for the console. It is the upper port. Connect the 10G interfaces according to your use case needs as described below.

Figure 4. LS2085ARDB/LS2088ARDB front panel

The following picture shows the rear of the board: the PCIe Ethernet card connector and power socket on the RDB.

Figure 5. LS2085ARDB/LS2088ARDB rear panel

Use the following steps to prepare the RDB for use:

1. Connect Ethernet cables to whichever Ethernet ports you want to use. If you are doing software development related to DPAA2, using the PCIe Ethernet can be convenient because it operates completely independently of the SoC's DPAA2 subsystem.
2. Attach a UART cable between the RDB top port and host computer.

3. Open a serial console tool on the host computer to communicate with the RDB.
4. Configure the host computer's serial port with the following settings:
   - Data rate: 115200 bps
   - Number of data bits: 8
   - Parity: None
   - Number of stop bits: 1
   - Flow control: Hardware/None
5. Connect the power supply to the board.

4.4.9.2.3 Default boot of the RDB

The RDB comes pre-loaded with software. The software is loaded in NOR flash on the RDB. Power on the RDB by pressing the power button on the front of the chassis and the U-Boot boot loader will automatically run and then load and start Linux using images located in vbank0 of the RDB's NOR flash. You can press a key while U-Boot is counting down to stop the Linux boot and get a U-Boot command prompt.

An example boot log is available in Detailed deployment steps on page 226.

4.4.9.3 LS2085ARDB/LS2088ARDB Details

4.4.9.3.1 On-board switch settings

The figure below shows default configuration switch settings for rev D - F boards. Earlier boards are not supported.

The bits on the switches are numbered from 1 to 8. In the figure, bit 1 is on the left.

Switch	Settings (1: ON, 0: OFF)
SW5	1111 1111
SW3	0001 0010
SW4	1111 1111
SW6	1111 1111
SW7	0100 0010 (100 MHz SYSCLOCK)
SW9	0100 0000
SW8	0101 1111

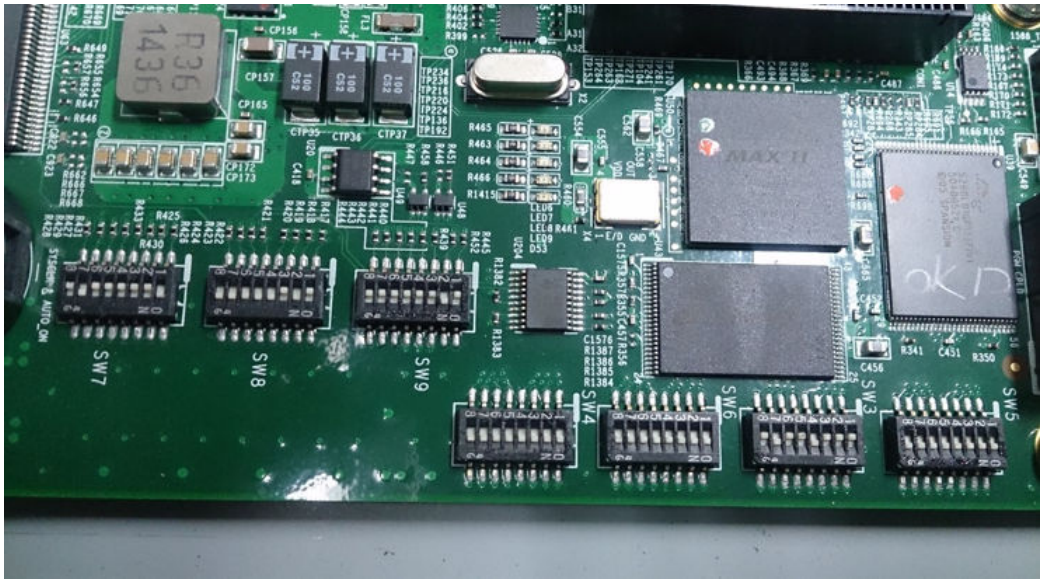


Figure 6. Switches on Rev D - F Boards

Refer to the *QorIQ LS2085A/LS2088A Reference Design Board Reference Manual* for detailed switch descriptions.

4.4.9.3.2 PBL, clocks, NOR Flash banks, and switches

This section describes features of the LS2085ARDB/LS2088ARDB that are relevant to deploying images. Understanding these features will help you to understand the deployment steps. The section concludes by outlining a deployment strategy.

This section explains:

- The connection between SYSCLK and PBL file changes.
- What a NOR flash bank is and why they are useful.
- The overall strategy of NOR flash deployments.

4.4.9.3.2.1 PBL files and SYSCLK

Pre-boot loader (PBL) files are automatically read by SoCs when they reset. The PBL file specifies important hardware configuration parameters including multipliers and dividers that are used to create the frequencies of the various clocks that the SoC requires. These clocks are formed by applying multipliers and dividers to a clock called SYSCLCK that is an input signal to the SoC.

For example, the clocks driving the ARM cores and the clock driving general SoC platform logic are derived from SYSCLK using multipliers and dividers specified in the PBL file. This means that a given PBL file is designed to be used with a particular SYSCLK frequency. The RDB must be configured to provide the SYSCLK that the PBL file you deploy requires.

4.4.9.3.2.2 RDB switch settings and SYSCLK frequency selection

The SYSCLK frequency is controlled by switch settings on the RDB. For revisions D – E of the LS2085ARDB/LS2088A RDB, SYSCLK is controlled by bits 1:3 of SW7. The available SYSCLK frequencies are:

Table 52. Available SYSCLK frequencies

Switch Setting	Frequency
off off off	66.666 MHz

Table continues on the next page...

Table 52. Available SYSCLK frequencies (continued)

Switch Setting	Frequency
off off on	83.333 MHz
off on off	100.000 MHz
off on on	125.000 MHz
on off off	133.333 MHz

You must set the switches to match what your PBL file expects. There is some complexity regarding when changes to switches take effect. The easy approach is to change the switches when the board is powered off. That will always work.

NOTE

The recommended frequency for LS2085ARDB/LS2088ARDB is 100MHz.

4.4.9.3.2.3 NOR Flash (virtual) banks

NOR flash is a simple and convenient destination for deploying images so it is frequently used. Many NXP development and reference boards provide a special feature that allows a single NOR flash to be divided into multiple parts called “banks”. This is done by board-level logic that modifies address signals. Because there is only one NOR flash physically, the banks are sometimes called "virtual" banks.

The benefit of this feature is that it allows more than one set of images to be independently deployed to the one NOR flash. This is very helpful during development because you can use the U-Boot image in one bank to program an image set into a different bank. If the new images are flawed, the old images are still functional to let you deploy corrected images.

The logic on the board usually allows the NOR flash to be divided into up to 8 banks, but almost always the feature is used to divide the flash into two halves only. The halves are called bank 0 and bank 4.

This is relevant to SYSCLK and PBLs because PBL files are one of the image types that must be deployed. It is possible to have different PBL files in the two banks, and they may require different SYSCLK frequencies. One must operate the board correctly in order to avoid using the wrong SYSCLK with a PBL.

4.4.9.3.2.4 Switch settings and NOR banks

On the LS2085ARDB/LS2088ARDB revisions D – F, switch SW9 bits 3:5 control which bank the SoC loads from when it powers up. The relevant values are:

Table 53. Switch Settings and NOR Banks

Switch Settings	NOR Banks
off off off	Load from NOR Flash Bank 0
on off off	Load from NOR Flash Bank 4

U-Boot prints which bank is loaded from. The output looks like following.

- For bank0

```
U-Boot 2016.01LS2088A-SDK+ge54c320 (Jul 17 2016 - 22:56:33 +0800)

SoC:  LS2088E Version:1.0 (0x87090010)
Clock Configuration:
       CPU0(A72):1800 MHz  CPU1(A72):1800 MHz  CPU2(A72):1800 MHz
       CPU3(A72):1800 MHz  CPU4(A72):1800 MHz  CPU5(A72):1800 MHz
       CPU6(A72):1800 MHz  CPU7(A72):1800 MHz
```

```
       Bus:      700  MHz  DDR:      1866.667 MT/s     DP-DDR:   1600 MT/s
Reset Configuration Word (RCW):
       00000000: 48303838 48480048 00000000 00000000
       00000010: 00000000 00000000 00a00000 00000000
       00000020: 00801180 00002581 00000000 00000000
       00000030: 00000c0b 00000000 00000000 00000000
       00000040: 00000000 00000000 00000000 00000000
       00000050: 00000000 00000000 00000000 00000000
       00000060: 00000000 00000000 00027000 00000000
       00000070: 412a0000 00000000 00000000 00000000
I2C:   ready
Model: Freescale Layerscape 2088A RDB Board
Board: LS2085A/LS2088A-RDB, Board Arch: V1, Board version: D, boot from vBank: 0
```

- For bank4

```
U-Boot 2016.01LS2088A-SDK+ge54c320 (Jul 17 2016 - 22:56:33 +0800)

SoC:  LS2088E Version:1.0 (0x87090010)
Clock Configuration:
       CPU0(A72):1800 MHz  CPU1(A72):1800 MHz  CPU2(A72):1800 MHz
       CPU3(A72):1800 MHz  CPU4(A72):1800 MHz  CPU5(A72):1800 MHz
       CPU6(A72):1800 MHz  CPU7(A72):1800 MHz
       Bus:      700  MHz  DDR:      1866.667 MT/s     DP-DDR:   1600 MT/s
Reset Configuration Word (RCW):
       00000000: 48303838 48480048 00000000 00000000
       00000010: 00000000 00000000 00a00000 00000000
       00000020: 00801180 00002581 00000000 00000000
       00000030: 00000c0b 00000000 00000000 00000000
       00000040: 00000000 00000000 00000000 00000000
       00000050: 00000000 00000000 00000000 00000000
       00000060: 00000000 00000000 00027000 00000000
       00000070: 412a0000 00000000 00000000 00000000
I2C:   ready
Model: Freescale Layerscape 2088A RDB Board
Board: LS2085A/LS2088A-RDB, Board Arch: V1, Board version: D, boot from vBank: 4
```

4.4.9.3.2.5 The `qixis_reset` command

U-Boot on the LS2085ARDB/LS2088ARDB has a useful command called `qixis_reset`. The command `qixis_reset` does a hard reset, loading from the bank specified by the switches.

If the switches are set to load from bank 0, then the command `qixis_reset altbank` will cause a reset into bank 4 (without a need to change any switches). This is useful if banks 0 and 4 contain PBL files that require the same SYSCLK.

4.4.9.3.2.6 NOR Flash banks and addresses

The “current” bank is the bank that was loaded and is currently running. The “alternate” bank is the other bank. For example, if you are running from bank 0 then bank 4 is the alternate bank. If you are running from bank 4, bank 0 is the alternate bank. The alternate bank if often called the "other" bank because it is the bank from which you are NOT booted.

Addressing is based on current and alternate/other banks, not banks 0 and 4. This is best explained by an example. Suppose the address of something in the current bank is `0x5_8000_0000`. The corresponding address in the alternate bank is changed in one bit, `0x5_8400_0000`. For example, suppose the byte at `0x5_8000_0000` have value `0xaa` and, while running U-Boot in the current bank, and you read the byte at `0x5_8400_0000` and see `0xbb`. Now, reboot after changing SW9 bits 3:5 so you are running U-Boot from what was the alternate bank and is now the current bank. You will see `0xbb` at `0x5_8000_0000` and `0xaa` at `0x5_8400_0000`.

This scheme is very useful, because it means you can always deploy images to the alternate bank using the same addresses, regardless of which bank happens to be "current". In other words, the steps to write an image into bank 4 using U-Boot running from bank 0 are the same as the steps to write an image to bank 0 using U-Boot running from bank 4.

4.4.9.3.2.7 Overall NOR deployment strategy

Let's take an example where you have a functioning U-Boot in bank 0 and you want to deploy a new image set that requires a different SYSCLK value than the one currently in use with the bank 0 images. It is always best to deploy into the alternate bank.

1. Use U-Boot commands to write the new images to the alternate bank, i.e. write them to the alternate bank addresses. The exact commands are presented in a later section.
2. Power off the board.
3. Change SW7 bits 1:3 to select the new SYSCLK.
4. Change SW9 bits 3:5 to load from bank 4.
5. Power on and test the new images. They are running with bank 4 as the "current" bank.
6. If you are happy with the new images and want to put them into bank 0 also, simply repeat step 1 to write them to the alternate bank (which is now bank 0).
7. Power off the board.
8. Change SW9 bits 3:5 to load from bank 0.
9. Power on. You are now running the new images from bank 0 and they are also deployed to bank 4.

If you are repeatedly (because you are developing) deploying new images that don't require a SYSCLK change, the following strategy can be convenient and is simpler.

1. Boot from bank 0 (i.e. power on with SW9 bits 3:5 selecting bank 0).
2. Load the new images into the alternate bank (bank 4).
3. Enter the `qixis_reset altbank` command. You will boot into your new images (in bank 4) without changing any switches. But take care to look at what U-Boot prints. If your new images were catastrophically bad, the board may automatically boot again into bank 0.
4. Test your new images and repeat this process as needed.

Note that this simpler procedure will fail if the new images require a change to SYSCLK. It will fail for the obvious reason that it contains no change to the SYSCLK frequency.

4.4.9.3.3 NOR Flash layout for bank 0 and bank 4

Images	Size	vbank0	vbank4
RCW + PBI	1MB	0x5_8000_0000	0x5_8400_0000
U-boot boot loader	1MB	0x5_8010_0000	0x5_8410_0000
U-boot Env	1MB	0x5_8020_0000	0x5_8420_0000
MC FW	4MB	0x5_8030_0000	0x5_8430_0000
MC DPL Blob	1MB	0x5_8070_0000	0x5_8470_0000
MC DPC Blob	1MB	0x5_8080_0000	0x5_8480_0000
Unused	4MB	0x5_8090_0000	0x5_8490_0000

Table continues on the next page...

Table continued from the previous page...

Reserved	1MB	0x5_80F0_0000	0x5_84F0_0000
Cortina PHY FW	1MB	0x5_8100_0000	0x5_8500_0000
FIT Images (Linux, device tree, and rfs)	40MB	0x5_8110_0000	0x5_8510_0000
Unused	7MB	0x5_8390_0000	0x5_8790_0000
Primary Protected Application (PPA)	2MB	0x5_80a0_0000	0x5_84a0_0000
Secure Boot Headers	3MB	0x5_80c0_0000	0x5_84c0_0000

4.4.9.4 Booting LS2085ARDB/LS2088ARDB

This section sumamrizes booting U-Boot and Linux using images that are already deployed to the board's NOR flash.

4.4.9.4.1 Booting U-Boot on the RDB

By default (per board switch settings), the boot loader (U-Boot) image located in NOR flash bank 0 runs on power on. Press any key while U-Boot is counting down to stop U-Boot from automatically running the "bootcmd" variable and booting Linux.

As the U-Boot boots to its prompt, users can use the commands listed below to deploy new images onto the RDB.

4.4.9.4.2 Booting Linux on the RDB

Booting Linux is controlled by the contents of U-Boot enviroment variable "bootcmd".

U-Boot passes the contents of U-Boot environment variable "bootargs" to Linux as boot-time kernel parameters. The commands in this variable are automatically run and Linux boots after U-Boot counts down a number of seconds given by U-Boot environment variable "bootdelay". Variable bootdelay can be set to -1 to avoid automatically booting Linux.

One can view the values of U-Boot enviroment variables using the U-Boot "printenv" command. The critical variables for booting Linux and their default values are shown below.

1. bootargs: contains parameters that are passed to the Linux kernel before it starts.

   ```
   printenv bootargs
   bootrgs=console=ttyS1,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0600,
   ramdisk_size=0x2000000 default_hugepagesz=2m hugepagesz=2m hugepages=256
   ```

   Notes on these kernel parameters:

   - Select ttyS1 as the console serial port (ttyS0 will not work on early LS2085A RDB/ LS2088A RDB boards, prior to Rev D).
   - Select file system to be a RAM disk populated from an image in the kernel itb file.
   - Allocate huge pages for user space (ODP) use. This is not needed unless you plan to use ODP. Please see ODP documentation for more information.

2. Environment variable "mcinitcmd" contains commands used during U-Boot for Management Complex load. It is not mandatory but very useful. Please refer U-Boot Environment Variables Relating to DPAA2 on page 237 for more information.

   ```
   printenv mcinitcmd
   mcinitcmd=fsl_mc start mc 0x580300000 0x580800000
   ```

3. Environment variable "bootcmd" contains commands used to boot Linux. The image needs to be copied to DDR first.

```
printenv bootcmd
bootcmd=cp.b $kernel_start $kernel_load $kernel_size && bootm $kernel_load
```

U-Boot enviroment variables such as "kernel_load" contain addresses used in the boot process. You can inspect them using the U-Boot "printenv" command. "bootcmd" command may change depending upon mcinitcmd env variable.

For information about the location of images in Flash, see NOR Flash layout for bank 0 and bank 4 on page 224.

4.4.9.5 Deploying Images

This section describes deploying image files on the RDB.

4.4.9.5.1 Updating images on the RDB

The update procedure assumes you have a working U-Boot image on the RDB. You will use U-Boot commands to download and flash new images into the "other" bank of the board's NOR flash. It is assumed that NOR flash is the boot device. This is controlled by switches on the board.

Images on the RDB can be updated by downloading new images from a host system connected to the RDB using any of the board's Ethernet ports, either one of the DPAA2 Ethernet ports on the front of the system or the PCIe Ethernet card on the back.

The following images must be programmed into the NOR flash in order for the U-Boot boot loader to function:

- PBL image (controls SerDes and PLL configurations, etc.)
- PPA Binary
- U-Boot image itself
- Cortina PHY firmware image (for the optical Ethernet ports).

U-Boot can read the other images needed to boot Linux from other devices such as disk drives or a TFTP server. U-Boot is very flexible. This document, however, assumes that these other images are also programmed into the NOR flash.

- Management Complex firmware, an itb format file.
- Management Complex data path control (DTC) file, a dtb format file.
- Management Complex data path layout (DPL) file, a dtb format file.

U-Boot has the capability to program images into the NOR flash. The general procedure is to TFTP the images from a TFTP server via Ethernet into RDB DDR and then program the images into NOR.

The section below describes the U-Boot commands to do this.

Recall that the NOR flash is deviced into banks. The commands below program the images into the "other" bank, i.e. if you have booted bank 0, the commands program bank 4. If you have booted from bank 4, the commands program bank 0. This process ensures that your board remains functional if you program erroneous images into the "other" bank because the good image remains in the current bank.

Normally, you boot from bank 0 and then program bank 4. Then, boot the images in bank 4 to test them.

If desired, you can use the same commands while booted from bank 4 to program bank 0.

4.4.9.5.2 Detailed deployment steps

This section describes in detail the steps needed to deploy a new image set on the target RDB board with the following assumptions: there is already a working U-Boot from previous early access releases in NOR flash bank 0. In summary, the steps are:

1. Place the images files onto the tftp server

2. Boot the RDB from bank 0.
3. Set up networking on the RDB (this assumes there is serial port access to the RDB).
4. Choose the files to be deployed onto the RDB's NOR flash bank 4 (actually the "other" bank).
5. Perform the deployment to bank 4 via TFTP.
6. Boot bank 4 and test the images in it.
7. Optionally, repeat the steps to deploy to bank 0.

Place the images to be deployed onto the TFTP server.

First, you must have a working TFTP server. There are so many ways to do it. The process depends on which operating system your server runs. Even if your server is running Linux, the process of settting up a TFTP server varies from Linux distribution to Linux distribution. Beware of firewall settings on your server.

The SDK images to put onto your TFTP server can be obtainedin two different ways:

1. Simply copy them from the "images" ISO that is available as part of the release.
2. Install the "sources" ISO onto your build server and build all of the images from source code. See "Getting Started with Yocto Project".

Boot from bank 0

Connect a serial cable to the upper RS-232 port and use settings 115200 baud, 8 data bits, 1 stop bit, no parity, no flow control.

Be sure SW9 bits 3:5 are off-off-off to boot from bank 0.

Power on the board.

Press a key using the serial port to stop U-Boot from booting Linux.

Set up networking on the RDB

The RDB must be on the same network as the TFTP server that contains the images you wish to flash. It is wise to test your RDB's networking before starting the image flashing process.

When U-Boot starts, it prints a list of available network interfaces, but the names U-Boot uses for the interfaces vary depending on which U-Boot version you are currently running.

Few early access releases' U-Boot prints something like this: "DPNI1, e1000#0 [PRIME]" where "e1000#0" is the e1000 PCIe card and "DPNI1" depends on your data path layout file but likely corresponds to DPMAC5 (right most copper port).

But later releases print the list of available interfaces using names like "DPMAC1@xgmii [PRIME], DPMAC2@xgmii, DPMAC3@xgmii, DPMAC4@xgmii, DPMAC5@xgmii, e1000#0".

Use any of the interfaces that U-Boot prints for deployment. Select the interface to use by setting the U-Boot "ethact" variable. The example below uses DPMAC5@xgmii.

Make sure the Management Complex (MC) firmware is started before doing U-Boot networking. Again, this varies depending on what version of U-Boot you are currently running. The MC will automatically load via the `mcinitcmd` U-Boot environment variable:

```
=> pri mcinitcmd
mcinitcmd=fsl_mc start mc 0x580300000 0x580800000
```

Unset the `mcinitcmd` if you do not intend to load the MC by default.

At this point, you can test U-Boot networking. Of course, select values that match your network, including the value of ethact.

```
=> setenv ethact DPMAC5@xgmii
=> setenv ipaddr 192.168.1.100
=> setenv netmask 255.255.255.0
=> setenv serverip 192.168.1.254
```

```
=> ping $serverip
Using DPMAC5@xgmii device
host 192.168.1.254 is alive
```

Choose the files you want to deploy

The table below summarizes the images to be programmed and their "other bank" address in the NOR flash.

NOTE

You must select image file names per the instructions below. The table contains symbolic names for the image files.

Binary	Description	**Other** (bank4) address
pbl_img	(RCW + PBL) binary (mandatory)	0x584000000
uboot_img	u-boot bootloader (mandatory)	0x584100000
mcfw_img	MC firmware image (mandatory)	0x584300000
mcdpc_img	Data Path Configuration file (mandatory)	0x584800000
mcdpl_img	Data Path Layout file (mandatory)	0x584700000
cor_img	Cortina PHY firmware (mandatory)	0x585000000
kern_img	Linux kernel + Platform Device Tree + Rootfs	0x585100000
ppa_img	PPA firmware	0x584a00000

Selecting a PBL (RCW) Image

The SDK release supports the following PBL files. Pick one of them according to your needs.

- **`ls2088ardb/ls2088ardb/FFFFFFFF_PP_HH_0x2a_0x41/PBL_0x2a_0x41_1600_600_1600_1600.bin`**
  - Core clock: 1600 MHz
  - Required board SYSCLK: 600 MHz (SW7[1:3] = off-on-off)
  - DDR: 1600 MHz
- **`ls2088ardb/ls2088ardb/FFFFFFFF_PP_HH_0x2a_0x41/PBL_0x2a_0x41_1800_700_1866_1600.bin`**
  - Core clock: 1800 MHz
  - Required board SYSCLK: 700 MHz (SW7[1:3] = off-on-off)
  - DDR: 1866 MHz
- **`ls2088ardb/ls2088ardb/FFFFFFFF_PP_HH_0x2a_0x41/PBL_0x2a_0x41_2000_800_2133_1600.bin` (RevF only)**
  - Core clock: 1000 MHz
  - Required board SYSCLK: 800 MHz (SW7[1:3] = off-on-off)
  - DDR: 2133 MHz

Choose specific files to deploy

The files to deploy are available on the images ISO or as a result of building from source using Yocto. Set U-Boot variables to the names of the files to deploy. The directories in the names will depend on how you set up your TFTP server.

```
=> setenv pbl_img sdk/ls2088ardb/rcw/rcw_0x2a_0x41/PBL_1800_700_1866_1600_0x2a_0x41.bin
=> setenv uboot_img sdk/ls2088ardb/u-boot-ls2088ardb.bin
=> setenv cor_img sdk/ls2088ardb/ls2-phy/CS4340/phy_firmware/cs4315-cs4340-PHY-ucode.txt
=> setenv mcfw_img sdk/ls2088ardb/mc_app/mc.itb
```

```
=> setenv mcdpc_img sdk/ls2088ardb/dpl-examples/dpc-0x2a41.dtb
=> setenv mcdpl_img sdk/ls2088ardb/dpl-examples/dpl-eth.0x2A_0x41.dtb
=> setenv kern_img sdk/ls2088ardb/kernel-ls2088ardb.itb
=> setenv ppa_img sdk/ls2088ardb/ppa.itb
```

Perform the deployment to bank 4 via TFTP

The following U-Boot commands TFTP images, erase old images, and program new images to the NOR flash addresses of the "other" bank.

```
# Flash PBL
=> tftp 0xa0000000 $pbl_img && erase 0x584000000 0x5840FFFFF && cp.b 0xa0000000
0x584000000 $filesize

# Erase old U-Boot environment
=> erase 0x584200000 +0x100000

# Flash U-Boot
=> tftp 0x80000000 $uboot_img && erase 0x584100000 +$filesize && cp.b 0x80000000
0x584100000 $filesize

# Flash Cortina (optical) PHY firmware
=> tftp 0x80000000 $cor_img && erase 0x585000000 +$filesize && cp.b 0x80000000
0x585000000 $filesize

# Flash MC-related images
=> tftp 0x80000000 $mcfw_img && erase 0x584300000 +0x300000 && cp.b 0x80000000 0x584300000
0x300000
=> tftp 0x80000000 $mcdpc_img && erase 0x584800000 +$filesize && cp.b 0x80000000
0x584800000 $filesize
=> tftp 0x80000000 $mcdpl_img && erase 0x584700000 +$filesize && cp.b 0x80000000
0x584700000 $filesize

# Flash FIT image with kernel, device tree, and file system image
=> tftp 0xa0000000 $kern_img && erase 0x585100000 +$filesize && cp.b 0xa0000000
0x585100000 $filesize

# Flash PPA image
=> tftp 0x80000000 $ppa_img && erase 0x584a00000 +$filesize && cp.b 0x80000000
0x584a00000 $filesize
```

Boot bank 4 and test the images in it

If the new images in bank 4 require a different SYSCLK value than the images in bank 0, you must use the following steps.

1. Power down the board.
2. Set SW7 bits 1:3 to select the SYSCLK frequency needed for the bank 4 images.
3. Set SW9 bits 3:5 to on-off-off to boot from bank 4. (bank 0 is off-off-off)
4. Power on the board.

If the new bank 4 images require the same SYSCLK frequency as the bank 0 images, there is a short cut. Enter the U-Boot command

```
=> qixis_reset altbank
```

to boot bank 4. You can enter

```
=> qixis_reset
```

from bank 4 U-Boot to boot bank 0 again.

After booting bank 4, press a key to stop Linux from automatically booting.

Use the U-Boot "printenv" to inspect the default U-Boot environment variable values. You may want to adjust some of them according to your names, e.g. the network settings variables such as "ipaddr" and "netmask".

Even if you make no changes, do a U-Boot "saveenv" command to update the values in NOR flash.

A complete U-Boot boot log follows.

```
U-Boot 2016.01LS2088A-SDK+g4cd72d5 (Oct 18 2016 - 03:54:37 +0800)

SoC:  LS2088E Version:1.0 (0x87090010)
Clock Configuration:
       CPU0(A72):1800 MHz  CPU1(A72):1800 MHz  CPU2(A72):1800 MHz
       CPU3(A72):1800 MHz  CPU4(A72):1800 MHz  CPU5(A72):1800 MHz
       CPU6(A72):1800 MHz  CPU7(A72):1800 MHz
       Bus:      700  MHz  DDR:      1866.667 MT/s     DP-DDR:   1600 MT/s
Reset Configuration Word (RCW):
       00000000: 48303838 48480048 00000000 00000000
       00000010: 00000000 00000000 00a00000 00000000
       00000020: 01001180 00002581 00000000 00000000
       00000030: 00000c0b 00000000 00000000 00000000
       00000040: 00000000 00000000 00000000 00000000
       00000050: 00000000 00000000 00000000 00000000
       00000060: 00000000 00000000 00027000 00000000
       00000070: 412a0000 00000000 00000000 00000000
I2C:   ready
Model: Freescale Layerscape 2085a RDB Board
Board: LS2085A/LS2088A-RDB, Board Arch: V1, Board version: D, boot from vBank: 4
FPGA: v1.19
SERDES1 Reference : Clock1 = 156.25MHz Clock2 = 156.25MHz
SERDES2 Reference : Clock1 = 100MHz Clock2 = 100MHz
DRAM:  Initializing DDR....using SPD
SPD DQ mapping error fixed
SPD DQ mapping error fixed
Detected UDIMM 18ASF1G72AZ-2G1A1
Detected UDIMM 18ASF1G72AZ-2G1A1
DP-DDR:  SPD DQ mapping error fixed
Detected UDIMM 18ASF1G72AZ-2G1A1
19 GiB
DDR    15 GiB (DDR4, 64-bit, CL=13, ECC on)
       DDR Controller Interleaving Mode: 256B
       DDR Chip-Select Interleaving Mode: CS0+CS1
DP-DDR 4 GiB (DDR4, 32-bit, CL=11, ECC on)
       DDR Chip-Select Interleaving Mode: CS0+CS1
SEC0: RNG instantiated
PPA Firmware: Version 0.2
Using SERDES1 Protocol: 42 (0x2a)
Using SERDES2 Protocol: 65 (0x41)
Flash: 128 MiB
NAND:  2048 MiB
MMC:   FSL_SDHC: 0
EEPROM: Invalid ID (ff ff ff ff)
PCIe1: disabled
PCIe2: disabled
PCIe3: Root Complex x1 gen1, regs @ 0x3600000
PCI:
     01:00.0    - 8086:107d - Network controller
PCIe3: Bus 00 - 01
```

```
PCIe4: Root Complex no link, regs @ 0x3700000
In:    serial
Out:   serial
Err:   serial
Debug Server FW timed out (boot status: 0x1)
Target spinup took 0 ms.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 1 device(s).
SCSI:  Net:   e1000: 00:15:17:8e:7c:ea
       DPMAC1@xgmii, DPMAC2@xgmii, DPMAC3@xgmii, DPMAC4@xgmii, DPMAC5@xgmii, DPMAC6@xgmii,
DPMAC7@xgmii, DPMAC8@xgmii, e1000#0 [PRIME]
Warning: e1000#0 MAC addresses don't match:
Address in SROM is         00:15:17:8e:7c:ea
Address in environment is  00:e0:0c:00:77:08

=>
```

Boot Linux

To boot Linux from the U-Boot command prompt, enter

```
=> boot
```

You can login as user "root" with no password.

A Linux boot log follow. The middle part of the log is deleted to save space.

```
=> boot
## Loading kernel from FIT Image at a0000000 ...
Using 'config@1' configuration
Trying 'kernel@1' kernel subimage
Description:  ARM64 Linux kernel
Created:      2016-07-17   0:33:13 UTC
Type:         Kernel Image     Compression:
gzip compressed     Data Start:   0xa00000dc
Data Size:    5398055 Bytes = 5.1 MiB
Architecture: AArch64     OS:           Linux
Load Address: 0x80080000
Entry Point:  0x80080000
Verifying Hash Integrity ... OK
## Loading ramdisk from FIT Image at a0000000 ...
Using 'config@1' configuration
Trying 'ramdisk@1' ramdisk subimage
Description:  LS2 Ramdisk
Created:      2016-07-17   0:33:13 UTC
Type:         RAMDisk Image
Compression:  uncompressed
Data Start:   0xa0529a8c
Data Size:    26384298 Bytes = 25.2 MiB
Architecture: AArch64     OS:
Linux     Load Address: unavailable
Entry Point:  unavailable
Verifying Hash Integrity ... OK
## Loading fdt from FIT Image at a0000000 ...
Using 'config@1' configuration
Trying 'fdt@1' fdt subimage
Description:  Flattened Device Tree blob
Created:      2016-07-17   0:33:13 UTC
Type:         Flat Device Tree
```

```
Compression:  uncompressed
Data Start:   0xa0525fb8
Data Size:    14928 Bytes = 14.6 KiB
Architecture: AArch64
Verifying Hash Integrity ... OK
Loading fdt from 0xa0525fb8 to 0x90000000
Booting using the fdt blob at 0x90000000
Uncompressing Kernel Image ... OK
reserving fdt memory region: addr=80000000 size=10000
Loading Device Tree to 000000009fff9000, end 000000009ffffa4f ... OK
Starting kernel ...
[    0.000000] Booting Linux on physical CPU 0x0

    <<<<<< MIDDLE OF LOG NOT SHOWN >>>>>>>>>>

Starting OpenBSD Secure Shell server: sshd
  generating ssh RSA key...
  generating ssh ECDSA key...
  generating ssh DSA key...
  generating ssh ED25519 key...
done.
Starting rpcbind daemon...rpcbind: cannot create socket for udp6
rpcbind: cannot create socket for tcp6
done.
starting statd: done
Starting network benchmark server: netserver.
Starting system log daemon...0
Starting kernel log daemon...0
Starting internet superserver: xinetd.

QorIQ SDK (FSL Reference Distro) 2.0 ls2088ardb /dev/ttyS1
ls2088ardb login: root
root@ls2088ardb:~#
```

Optionally, repeat the steps to deploy to bank 0

You can deploy images to bank 0 by booting U-Boot from bank 4 and using the same flash programming steps documented above. The addresses in NOR flash do not change because they always represent the "other" bank, i.e. the bank you have not booted.

4.4.9.5.3 Deploying the full file system

The images ISO contains a compressed tar file with a large number of Yocto packages built, including native GNU tools.

It is large enough that it must be deployed to a mass storage device such as a SATA drive, USB drive, or SD card.

This can be done by booting the normal RAM disk file system. Choose a mass storage device. This example will use /dev/sdc. The example target system happens to have two SATA drives and a USB drive (/dev/sdc). Then perform these steps.

On the RDB booted from the RAM disk:

```
# bring up networking (example uses rightmost copper port) and set
# the time and date using any method
ifconfig ni0 192.168.1.100/24
rdate -s 192.168.1.254

# Use umount to ummount any /dev/sdc partitions that happen to be mounted
# and then create a partition on /dev/sdc.
fdisk /dev/sdc
o    - new DOS partition table
n    - new partition
```

```
p    - primary
<cr> - default partition number
<cr> - default partition first sector
<cr> - default partition last sector
w   - write partition table and exit

# Be sure /dev/sdc1 did get get mounted.  Then make a file system on it
# and mount that file system.
umount /run/media/sdc1   (if needed)
mkfs.ext4 /dev/sdc1
mkdir /mnt/newroot
mount /dev/sdc1 /mnt/newroot
```

On the machine with the images ISO, scp the full file system tar file to the RDB.

```
scp full-rootfs/LS2088A_SDK_AARCH64_20160719_ROOTFS_Image.tar.gz root@192.168.1.100:/mnt/newroot
```

Back on the RDB, untar the file system.

```
cd /mnt/newroot
tar pzxf LS2088A_SDK_AARCH64_20160719_ROOTFS_Image.tar.gz
sync    # Might take some time for USB or SD
```

Reboot the RDB with bootargs and bootcmd set as follows.

```
setenv bootargs console=ttyS1,115200 root=/dev/sdc1 rootwait earlycon=uart8250,mmio,0x21c0600
ramdisk_size=0x2000000 default_hugepagesz=2m hugepagesz=2m hugepages=256 setenv bootcmd 'fsl_mc
apply dpl 0x580700000 && cp.b $kernel_start $kernel_load $kernel_size && bootm
$kernel_load:kernel@1 - $kernel_load:fdt@1'
```

4.4.9.6 AQ405 Firmware Update

This chapter is required only if a user needs to program or update the AQ405 firmware on the LS2085ARDB/LS2088ARDB. This is already programmed on boards received from the factory. Unless a firmware update is required, please ignore these steps.

LS2085ARDB/LS2088ARDB has the AQ405, which contains a copper PHY and it needs firmware in order to be programmed; this firmware is permanent and needs to be programmed once. Over time, AQ405 firmware upgrades may be required. The following sections provide instructions on how to check the firmware version and upgrade it if needed.

4.4.9.6.1 Check firmware version

Run the following U-Boot commands to check the AQ405 firmware version. The firmware version should be 0x203 0x52.

```
=> mdio read FSL_MDIO1 0  0x1e.0x20
Reading from bus FSL_MDIO1
PHY at address 0:
30.32 - 0x203

=> mdio read FSL_MDIO1 0  0x1e.0xc885
Reading from bus FSL_MDIO1
PHY at address 0:
 30.51333 - 0x52
```

4.4.9.6.2 Update new firmware

Assumption: The host machine used to download images is running a TFTP server, with server IP 192.168.1.1, and all images to be downloaded are available in the tftpboot folder. Also, the host machine must be connected to the RDB system using the PCIe NIC card as described in Board preparation on page 219.

1. Download AQ_API.bin from host machine:

```
tftp 0x80300000 AQ_API.bin
```

2. Download AQ28nm-FW_2.3.52_Freescale_T2085RDB_030415.cld from host machine:

```
tftp 0x80000000 AQ28nm-FW_2.3.52_Freescale_T2085RDB_030415.cld
```

NOTE

Download images in the order mentioned above (1 followed by 2)

3. Run the following commands to update the new firmware:

```
go 0x80300000 FSL_MDIO1 0x0 0x80000000 $filesize
```

Check for the following logs on U-Boot prompt:

```
=> go 0x80300000 FSL_MDIO1 0x0 0x80000000 $filesize
## Starting application at 0x80300000 ...
Actual U-Boot ABI version 8
argc = 5
## AQ Firmware update Program Starts
flashing firmware for AQR405
FLASH type = Atmel AT45DB041D
CRC check good on image file (0x75BC)
Bytes in file 0x45800
  Bytes: 0x100
  Bytes: 0x200
...
  Bytes: 0x45700
  Bytes: 0x45800
CRC check good on image file (0x75BC)

Starting read of FLASH data
  Bytes: 0x45800
OK (CRC 0x75BC)
Device burned and verified
Hit any key to exit ...

## Application terminated, rc = 0x0
```

4.4.9.7 Supported RCW Binaries

LS2088A

LS2088A supports following RCW binaries:

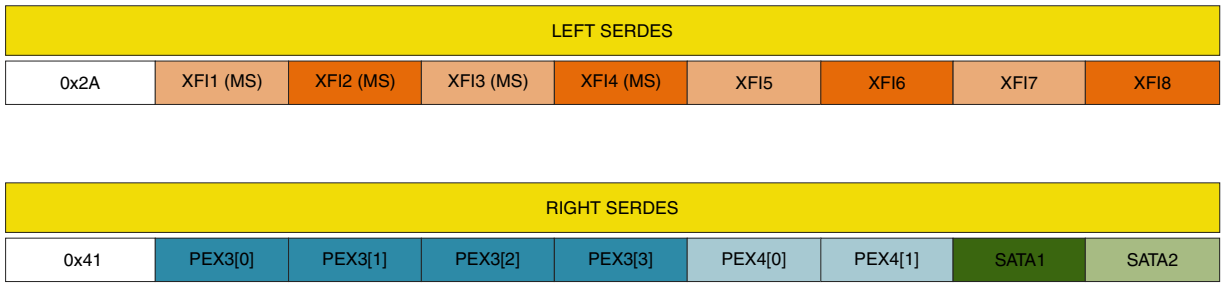
Table 54. RCW binary

SoC	Binary Name	Descripton	Required SYSCLK
LS2088A	PBL_0x2a_0x41_1600_600_1600_1600.bin	Core: 1600 MHz, Platform: 600 MHz, DDR: 1600 MHz, DP DDR 1600 MHz	100 MHz, SW7[1:3] = 010
LS2088A	BL_0x2a_0x41_1600_600_1600_1600.bin	Core: 1800 MHz, Platform: 700 MHz, DDR: 1866 MHz, DP DDR 1600 MHz	100 MHz, SW7[1:3] = 010
LS2088A	PBL_0x2a_0x41_2000_800_2133_1600.bin	Core: 1800MHz, Platform: 800MHz, DDR: 2133 MHz, DP DDR 1600 MHz	100 MHz, SW7[1:3] = 010

NOTE

The RCW's provided with the release enable the following features:

1. The figure below shows the SERDES configuration supported for LS2088A. Note that LS2088A supports upto 5 ports out of the 8 available on the RDB at a time



2. Boot location as NOR flash
3. Enables 4 UART without flow control
4. Enables I2C1, I2C2, I2C3, I2C4, SDHC, IFC, PCIe, SATA

4.4.9.7.1 Generating 500MHz RCW/PBL binary using QCVS tool

QCVS is a tool which provides a graphical user interface (GUI) for editing a PBL (RCW) files in decoded form. The PBL (RCW) file can be also be created from scratch or imported from an existing reset configuration word.

Please refer to QorIQ Configuration and Validation Suite for more information.

Steps for generating 500MHz RCW/PBL file using 533MHz PBL (RCW) file:

1. Create project and import the 533MHz file as outlined in *PBL Configuration using QCVS* (document: AN5260)
2. Update PLL configuration to set SYS_PLL_RAT to 10:1, CGA_PLLx_RAT to 18:1 and CGB_PLLy_RAT to 18:1 (here x,y can be 1, 2)

PLL CONFIGURATION (Bits 143-0)	
SYS_PLL_CFG [1-0]	0b00 - For all valid Platform PLL fr...
SYS_PLL_RAT [6-2]	0b01010 - 10:1
MEM_PLL_CFG [9-8]	0b00 - All valid DDR PLL frequencies
MEM_PLL_RAT [15-10]	0b001110 - 14:1
MEM2_PLL_CFG [17-16]	0b00 - All valid DDR PLL frequencies
MEM2_PLL_RAT [23-18]	0b001100 - 12:1
CGA_PLL1_CFG [25-24]	0b00 - All valid cluster group A PL...
CGA_PLL1_RAT [31-26]	0b010010 - 18:1 (Async mode)
CGA_PLL2_CFG [33-32]	0b00 - All valid cluster group A PL...
CGA_PLL2_RAT [39-34]	0b010010 - 18:1 (Async mode)
CGB_PLL1_CFG [49-48]	0b00 - All valid cluster group B PLL...
CGB_PLL1_RAT [55-50]	0b010010 - 18:1 (Async mode)
CGB_PLL2_CFG [57-56]	0b00 - All valid cluster group B PLL...
CGB_PLL2_RAT [63-58]	0b010010 - 18:1 (Async mode)
SYS_PLL_SPD [128]	0b0 - High speed operation (vco_d...
MEM_PLL_SPD [129]	0b0 - High speed operation (vco_d...
MEM2_PLL_SPD [130]	0b0 - High speed operation (vco_d...
CGA_PLL1_SPD [132]	0b0 - High speed operation (vco_d...
CGA_PLL2_SPD [133]	0b0 - High speed operation (vco_d...
CGB_PLL1_SPD [135]	0b0 - High speed operation (vco_d...
CGB_PLL2_SPD [136]	0b0 - High speed operation (vco_d...

3. Re-generate the PBL (RCW) file

4.4.9.7.2 Generating RCW for DSPI access

The SDK release does not provide a RCW file for the LS2088A DSPI access. If required, a user can generate the needed file be modifying the release PBL (RCW) file using the QCVS tool.

QCVS is a tool which provides a graphical user interface (GUI) for editing a PBL (RCW) files in decoded form. The PBL (RCW) file can be also be created from scratch or imported from an existing reset configuration word.

Refer to QorIQ Configuration and Validation Suite for more information.

Steps for generating RCW for DSPI access file:

1. Create project and import the release file (e.g. `PBL_1800_700_1866_1600_0x2a_0x41.bin`) as outlined in *PBL Configuration using QCVS* (document: AN5260)
2. Update RCW field configuration to set below table:

BIT(s)	Field Name	Setting
389-388	IIC3_BASE	11
393-392	SPI_BASE_BASE	00
395-394	SPI_PCS_BASE	00

3. Re-generate the PBL (RCW) file

4.4.9.8 U-Boot DPAA2 Commands

U-Boot contains a command called "fsl_mc". The primary purposes of this command are to load and start the Management Complex (MC) and to apply its Data Path Control (DPC) file.

The command is summarized below.

```
=> help fsl_mc
fsl_mc - DPAA2 command to manage Management Complex (MC)
Usage:
fsl_mc start mc [MC_FW_addr] [DPC_addr] - Start Management Complex
fsl_mc apply DPL [DPL_addr] - Apply DPL file
fsl_mc start aiop [AIOP_FW_addr] - Start AIOP
```

The addresses can be in NOR flash or DDR. The latter is used when MC files are loaded from a boot device like a disk drive.

Normally, users will not use fsl_mc to load and start the AIOP. Instead, this should be done using the "aiop_tool" Linux user space command. The fsl_mc command is used to load and start the AIOP only for low-level testing.

MC must be started before using DPAA2 networking in U-Boot, Linux, or both

DPAA2 networking depends on the MC, so DPAA2 networking cannot be used until the MC is loaded, supplied with a DPC file, and started. This is true for both U-Boot and Linux.

Here is an example that assumes the MC firmware image and Data Path Configuration (DPC) image are both in NOR flash.

fsl_mc start mc 0x580300000 0x580800000

U-Boot environment variable "mcinitcmd" can be used to automate loading and starting the MC, e.g. it can be set to contain the command above. Refer to the U-Boot Environment Variables Relating to DPAA2 on page 237 section for more information.

U-Boot DPAA2 networking does not rely on the Data Path Layout (DPL) file

U-Boot dynamically creates and destroys the objects it needs for DPAA2 networking. Users should not use the "fsl_mc apply DPL" command before using U-Boot networking.

Linux DPAA2 networking does reply on the DPL

Immediately before booting Linux use "fsl_mc apply DPL" command to apply the DPL (assuming DPL is present on NOR flash). It defines the initial set of DPAA2 objects for Linux. U-Boot networking is not available after executing this command.

```
=> fsl_mc apply DPL 0x580700000
```

As an example, the command above can be used in the U-Boot environment variable "bootcmd".

4.4.9.9 U-Boot Environment Variables Relating to DPAA2

1. DPAA2-specific Environment Variables

   - ***mcboottimeout***: Defines Management Complex boot timeout in milliseconds. If this variable is not defined the complile-time value, CONFIG_SYS_LS_MC_BOOT_TIMEOUT_MS will be the default. Normally, users do not need to set this variable because the default is acceptable.
   - ***mcmemsize***: Defines amount of system DDR to be use by the Management Complex. If this variable is not defined, the compile-time value CONFIG_SYS_LS_MC_DRAM_BLOCK_MIN_SIZE will be the default. Normally, users do not need to set this variable because the default is acceptable.
   - ***mcinitcmd***: Contains commands to load and start the Management Complex automatically before the U-Boot count down to boot starts. If this variable is defined, its contents are "run". The default value assumes that the Management Complex (MC) firmware and Data Path Control file are stored in NOR flash at fixed addresses. The default value is "fsl_mc start mc 0x580300000 0x580800000". Users may change this variable as needed to load the MC files from sources other than NOR into DDR and then start the MC using the fsl_mc command. For example, the files may be on a disk drive.

2. Impliciations for DPAA2 Environment variables that are not specific to DPAA2

   - ***bootcmd***: Contains commands that are automatically executed when the U-Boot "boot" command is run. This happens automatically when the user does not interrupt U-Boot's initial count down. In normal usage, bootcmd should contain the command to apply the Management Complex Data Path Layout (DPL) file because this must be done before booting Linux. The default value of bootcmd assumes that the DPL file is stored in NOR flash at a fixed address. The default is "fsl_mc apply dpl 0x580700000 && cp.b $kernel_start $kernel_load $kernel_size && bootm $kernel_load".

4.4.10 P2041RDB

4.4.10.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.10.2 Switch Settings

The RDB has user selectable switches for evaluating different boot options for the P2040/P2041 device. The table below lists the default switch settings. For a description of these settings, see P2041RDB Reference Manual, Section 22 Switch Settings.

Table 55. Default Switch Settings

	1	2	3	4	5	6	7	8
SW1	ON [1]	OFF [0]	ON [1]	ON [1]	OFF [0]	ON [1]	OFF [0]	OFF [0]
SW2	OFF	OFF	ON	OFF	OFF	ON	OFF	OFF
SW3	OFF	ON	OFF	ON	OFF	ON	ON	ON

Note that with the default SW1 settings, the following are selected.

SW1[1:5] = 10110:

RCW configuration source is eLBC GPCM 16-bit NOR Flash

SW1[6:8] = 100:

SYSCLK output frequency is 83.33MHz.

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 56. Boot Device Scenarios

Alternate Boot Devices	Change the following switch settings
SD card boot	SW1 [1:5] = 01100
SPI Flash boot	SW1 [1:5] = 10100
NAND Flash boot	SW1 [1:5] = 10010

4.4.10.3 U-Boot Environment Variables

4.4.10.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

If you plan to use optical 10G interfaces, you must add information to "hwconfig". This is important. The optical interface will operate erratically without it. It is easiest to do this using the U-Boot "*editenv*" command. Whatever you type will be appended to "hwconfig". The text you need to append is:

```
;fsl_fm1_xaui_phy:xfi
```

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;fsl_fm1_xaui_phy:xfi
```

4.4.10.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.10.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for P2041 version information (e.g., P2041E version 2.0) and also for the version of FMan microcode currently flashed on the P2041RDB (e.g. microcode version 106.1.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 1.0:

```
fsl_fman_ucode_p2041_r1.0_106_1_15.bin
```

- For silicon revision 1.1:

```
fsl_fman_ucode_p2041_r1.1_106_1_15.bin
```

- For silicon revision 2.0:

```
fsl_fman_ucode_p2041_r2.0_106_1_18.bin
```

NOTE

(i) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(ii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iii) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.10.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK 2.0 contains RCW binaries for use on the P2041RDB:

- RR_PH_0x19/rcw_5g_1200mhz.bin
- RR_PH_0x19/rcw_5g_1500mhz.bin
- RR_PX_0x09/rcw_14g_1500mhz.bin (default)

The RCW directories' names for the P2041RDB conform to the following naming convention:

```
ab_cd_e
```

Table 57. P2041RDB Naming Convention Legend

Slot	Convention
a	'R' if RGMII@DTSEC4 is supported 'N' if not available/not used
b	'R' if RGMII@DTSEC5 is supported 'N" if not available/not used
c [What is available in Slot 1] d [What is availabel in Slot 2 or SATA]	'N' if not available/not used 'P' if PCIe 'X' if XAUI 'R' if SRIO 'S' if SGMII 'S' if SGMII 'H' if SATA 'A' is AURORA
e	Hex value of serdes protocol value

For exmaple,

```
RR_PH_0x19
```

means:

- RGMII@DTSEC4
- RGMII@DTSEC5
- PCIE [Slot 1]
- SATA [Slot 2 not used]
- SERDES Protocol is 0x19

Below is a table that maps the different ports available on P2041RDB and their names in different environments.

Table 58. Port Map

Label on Panel of the P2041 RDB	Port in U-boot	Port in Linux	FMAN Address
GETH1-SGMII	FM1@DTSEC1	fm1-gb0	0xffe4e0000
GETH2-SGMII	FM1@DTSEC2	fm1-gb1	0xffe4e2000
GETH3-SGMII	FM1@DTSEC3	fm1-gb2	0xffe4e4000

Table continues on the next page...

Table 58. Port Map (continued)

GETH4-RGMII	FM1@DTSEC4	fm1-gb3	0xffe4e6000
GETH5-RGMII	FM1@DTSEC5	fm1-gb4	0xffe4e8000
No label (Supported by XAUI Riser Card)	FM1@TGEC1	fm1-10g	0xffe4f0000

4.4.10.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
0xfe0000000	0xfeffffffff	eLBC	256 MB
0xff4000000	0xff41fffff	Buffer manager software portal	2 MB
0xff4200000	0xff43fffff	Queue manager software portal	2 MB
0xfffdf0000	0xfffdf0fff	FPGA	4 KB
0xf00000000	0xf01ffffff	DCSR	32 MB
0xffa000000	0xffa0fffff	NAND	1 MB
0xc20000000	0xc2fffffff	Serial Rapid I/O 1	256 MB
0xc00000000	0xc1fffffff	PCI Express 1	512 MB
0xff8000000	0xff800ffff	PCI Express 1 I/O	64 KB
0xc40000000	0xc5fffffff	PCI Express 3	512 MB
0xff8020000	0xff802ffff	PCI Express 3 I/O	64 KB
0x00000000	0x7fffffff	Memory controller	2 GB

4.4.10.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration (for P2041RDB, SW2[2]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
CPU0:  P2041E, Version: 2.0, (0x82180120)
Core:  e500mc, Version: 3.2, (0x80230032)
Clock Configuration:
       CPU0:1500 MHz, CPU1:1500 MHz, CPU2:1500 MHz, CPU3:1500 MHz,
       CCB:750  MHz,
       DDR:666.667 MHz (1333.333 MT/s data rate) (Asynchronous), LBC:93.750 MHz
       FMAN1: 583.333 MHz
       QMAN:  375 MHz
       PME:   375 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 12600000 00000000 241c0000 00000000
       00000010: 648ea0c1 c3c02000 de800000 40000000
       00000020: 00000000 00000000 00000000 d0030f07
       00000030: 00000000 00000000 00000000 00000000
Board: P2041RDB, CPLD version: 4.0 vBank: 0
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>cpld reset
```

- Switch to alternate bank:

```
=>cpld reset altbank
```

The table below shows the P2041RDB NOR flash memory map.

Table 59. NOR Flash Memory Map

Range Start	Range End	Definition	Size
0xeff40000	0xefffffff	U-Boot (current bank)	768 KB
0xeff20000	0xeff3ffff	U-Boot env (current bank)	128 KB
0xeff00000	0xeff1ffff	FMAN Ucode (current bank)	128 KB
0xed300000	0xefefffff	Rootfs (alternate bank)	44 MB
0xed000000	0xed2fffff	Guest image #3 (alternate bank)	3 MB

Table continues on the next page...

Table 59. NOR Flash Memory Map (continued)

0xecd00000	0xecffffff	Guest image #2 (alternate bank)	3 MB
0xeca00000	0xeccfffff	Guest image #1 (alternate bank)	3 MB
0xec900000	0xec9fffff	HV config device tree (alternate bank)	1 MB
0xec800000	0xec8fffff	Hardware device tree (alternate bank)	1 MB
0xec700000	0xec7fffff	HV.uImage (alternate bank)	1 MB
0xec020000	0xec6fffff	Linux.uImage (alternate bank)	~7 MB
0xec000000	0xec01ffff	Rcw (alternate bank)	128 KB
0xebf40000	0xebffffff	U-Boot (alternate bank)	768 KB
0xebf20000	0xebf3ffff	U-Boot env (alternate bank)	128 KB
0xebf00000	0xebf1ffff	FMAN ucode (alternate bank)	128 KB
0xe9300000	0xebefffff	Rootfs (current bank)	44 MB
0xe9000000	0xe92fffff	Guest image #3 (current bank)	3 MB
0xe8d00000	0xe8ffffff	Guest image #2 (current bank)	3 MB
0xe8a00000	0xe8cfffff	Guest image #1 (current bank)	3 MB
0xe8900000	0xe89fffff	HV config device tree (current bank)	1 MB
0xe8800000	0xe88fffff	Hardware device tree (current bank)	1 MB
0xe8700000	0xe87fffff	HV.uImage (current bank)	1 MB
0xe8020000	0xe86fffff	Linux.uImage (current bank)	~7 MB

Table continues on the next page...

Table 59. NOR Flash Memory Map (continued)

0xe8000000	0xe801ffff	RCW (current bank)	128 KB

4.4.10.8 Programming a New U-Boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *cpld reset altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>cpld reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>cpld reset altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>. <fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
```

```
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>cpld reset altbank
```

4.4.10.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.10.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 1000000 <uImage_name>
   =>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 2000000 <platform_dtb_name>
   =>bootm 1000000 5000000 2000000
   ```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.10.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   =>saveenv
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.10.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

   b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
=>bootm 1000000 - c00000
```

   Now the board will boot up with NFS filesystem.

4.4.10.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

```
#fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to create the filesystems.

```
# mkfs.vfat /dev/sdb1
# mkfs.ext2 /dev/sdb2
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdb2 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

```
# cp uImage and uImage.dtb to /temp/boot folder
# umount /temp
```

8. Plug in the SD card to the target board and power on.

4.4.10.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

```
# fdisk /dev/sdx
```

3. Use the mkfs command to create the filesystems.

```
# mkfs.ext2 /dev/sdx1
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdx1 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure copy the target kernel image and dtb to sata.

```
# cp uImage and target.dtb to /temp/boot folder
# umount /temp
```

8. Connect the SATA hard drive to the target board and power on.

4.4.10.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

   TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

   ```
   =>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>erase e9300000 +$filesize
   =>cp.b 1000000 e9300000 $filesize
   ```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

   ```
   =>tftp 1000000 hv.uImage
   =>erase e8700000 +$filesize
   =>cp.b 1000000 e8700000 $filesize
   ```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

   ```
   =>tftp 1000000 <platform_name>-usdpaa.dtb
   =>erase e8800000 +$filesize
   =>cp.b 1000000 e8800000 $filesize
   ```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.10.10 Check 'Link Up' for Serial Ethernet Interfaces

If you are experiencing problems with your Ethernet interfaces, this section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
0 - Vitesse VSC8641 <--> FM1@DTSEC5
1 - Vitesse VSC8641 <--> FM1@DTSEC4
2 - Vitesse VSC8221 <--> FM1@DTSEC1
3 - Vitesse VSC8221 <--> FM1@DTSEC2
FM_TGEC_MDIO:
0 - Teranetics TN2020 <--> FM1@TGEC1
```

The results from the above *mdio list* command show that U-Boot was able to see PHYs on each of the 4 dTSEC interfaces and on the 10GEC interface. If you see “Generic” reported, it is an indication that something is there but the P2041 can’t communicate with the device/port.

Check SGMII Link Status for External PHY

In order to check the status of a SGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read FM1@DTSEC1 1
Reading from bus FSL_MDIO0
PHY at address 2:
1 - 0x796d
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7969.

Example:

```
=> mdio read FM1@DTSEC1  30
Reading from bus FSL_MDIO0
PHY at address 2:
```

```
30 - 0x25
```

The MAC interface (“fiber side”) link information is in Register #30 on the PHY. The last nibble is of interest. Bit #2 (from the left) is ‘Link Interlock Complete’ (Clause 37).

Check SGMII link Status for Internal PHY

The “TBI PHY” on-chip also has link information. The PHY address is programmed in TBI Physical Address Register (TBIPA) at offset 0x01C in the dTSEC register space.

```
=> md fe4e001c
fe4e001c: 00000008 00000000 00000000 00000000    ..............
```

The TBI PHY address is 0x8. In the TBI MII Register Set, the Status Register (SR) is register 1. So we check PHY=8, REG=1.

These additional registers in the MDIO register space may also be useful:

Offset 0x124 - MII Management Command Register (MIIMCOM)

Offset 0x12c - MII Management Control Register (MIIMCON)

Offset 0x130 - MII Management Status Register (MIISTAT)

Example:

```
=> mm fe4e1124
fe4e1124: 00000001 ? 1                   <==  Prepare for a read by writing a "1" to bit 31
fe4e1128: 0000021e ? 0801                <==  Write PHY address and Register address
fe4e112c: 00001240 ?
fe4e1130: 00000025 ? x
=> mm fe4e1124
fe4e1124: 00000001 ? 0                   <==  Prepare for a read by writing a  "0"  to bit 31
fe4e1128: 00000801 ? x
=> mm fe4e1124
fe4e1124: 00000000 ? 1                   <==  Transistion bit 31 from "0" to "1" in order to read
fe4e1128: 00000801 ?                     <==  Read PHY 8, REG 1
fe4e112c: 00001240 ?                     <==  Not used for read, only used for write cycles
fe4e1130: 0000017d ?                     <==  Data returned from read cycle
```

Here, 0x17d is the result of the read cycle. The ‘Link Status’ bit is bit #2 (from the left) of the last nibble. In this example, the last nibble is d=b'1101', therefore the 'Link Status' bit is "1".

Check XAUI link status for External PHY

In order to check the status of the XAUI link, you can use the *mdio read* command. Since this is a Clause 45 device, we pass three arguments to the *mdio read* command.

```
mdio read <PORT address> <DEVICE Address>.<REGISTER Address>
```

Example:

```
=> mdio read FM1@TGEC1  4.1
Reading from bus FM_TGEC_MDIO
PHY at address 0:
4.1 - 0x6
```

The MAC interface ("fiber side") link information is in Device#4, Register #1 on the PHY. The 'PHY XS Transmit Link Status' bit is bit #2 (from the left) of the last nibble.

Check XAUI Lane Synchronization

Check SERDES register 'Lane Alignment Status' bit to see if the SERDES lanes are correctly synchronized. The 'Lane Alignment Status' bit is bit #2 (from the left) of the 1st nibble. If the lanes are correctly synchronized, this bit is "1" and when the lane synchronization is lost or not acquired, this bit is a "0".

Example:

```
=>md fe0ea250 4
fe0ea250: c800000f 000f0000 00000000 00000000    ................
```

4.4.10.11 Basic Networking Ping Test

In Linux, in order to check that your network driver is set up, follow the commands below:

```
#ip addr show
#ip addr add <ip address of board>/24 brd + dev <port in Linux>
#ip link set <port in Linux> up
#ping <serverip>
```

If your network driver is not working, check your kernel messages. You should not see any errors reported by the FSL FMan MAC based driver:

```
fsl_mac: fsl_mac: FSL FMan MAC API based driver ()
fsl_mac ffe4e0000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e0000.ethernet: FMan MAC address: 00:04:9f:03:11:1e
fsl_mac ffe4e2000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e2000.ethernet: FMan MAC address: 00:04:9f:03:11:1f
fsl_mac ffe4e4000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e4000.ethernet: FMan MAC address: 00:04:9f:03:11:20
fsl_mac ffe4e6000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e6000.ethernet: FMan MAC address: 00:04:9f:03:11:21
fsl_mac ffe4e8000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e8000.ethernet: FMan MAC address: 00:04:9f:03:11:22
```

4.4.11 P3041/P5020DS

4.4.11.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.11.2 Switch Settings

The Development System has user selectable switches for evaluating different boot options for the P3041/P5020 device. The table below lists the default switch settings. For a description of these settings, see P3041/5020DS Hardware Getting Started Guide.

Table 60. Hydra P3041 Default Switch Settings

	1	2	3	4	5	6	7	8
SW1	OFF [0]	ON [1]	ON [1]	OFF [0]	ON [1]	OFF [0]	ON [1]	OFF [0]
SW2	OFF	OFF	ON	OFF	OFF	OFF	OFF	ON
SW3	OFF	ON	OFF	OFF	OFF	OFF	ON	ON
SW4	ON	ON	ON	ON	OFF	ON	OFF	ON
SW5	OFF	OFF	OFF	ON	OFF	ON	OFF	OFF
SW6	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON
SW7	OFF	OFF	OFF	OFF	ON	ON	ON	ON
SW8	ON	OFF	ON	OFF	OFF	OFF	ON	OFF
SW9	ON	ON	ON	OFF	OFF	OFF	OFF	OFF
SW10	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF

Table 61. Super Hydra P3041 Default Switch Settings

	1	2	3	4	5	6	7	8
SW1	OFF	ON	ON	OFF	ON	OFF	ON	OFF
SW2	OFF	OFF	ON	OFF	OFF	OFF	OFF	ON
SW3	OFF	ON	OFF	OFF	OFF	OFF	OFF	ON
SW4	ON	ON	ON	ON	OFF	ON	OFF	ON
SW5	OFF	OFF	OFF	ON	OFF	ON	OFF	OFF
SW6	OFF	OFF	OFF	OFF	OFF	OFF	OFF	ON
SW7	OFF	OFF	OFF	OFF	ON	ON	ON	ON
SW8	ON	OFF	ON	OFF	OFF	OFF	ON	OFF
SW9	ON	ON	ON	OFF	OFF	OFF	OFF	OFF
SW10	OFF	OFF	OFF	ON	OFF	OFF	OFF	OFF
SW11	OFF	OFF	ON	OFF	OFF	ON	OFF	OFF
SW12	ON	OFF	ON	ON	OFF	ON	OFF	ON
SW13	ON	ON	ON	OFF	OFF	ON	OFF	ON

Table 62. Hydra P5020 Default Switch Settings

	1	2	3	4	5	6	7	8
SW1	OFF	ON	ON	OFF	ON	OFF	ON	OFF
SW2	OFF	OFF	ON	OFF	OFF	OFF	OFF	ON

Table continues on the next page...

Table 62. Hydra P5020 Default Switch Settings (continued)

SW3	OFF	ON	OFF	OFF	OFF	ON	ON	OFF
SW4	ON	ON	ON	ON	ON	OFF	ON	ON
SW5	OFF	OFF	OFF	ON	OFF	ON	OFF	OFF
SW6	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON
SW7	OFF	OFF	OFF	OFF	ON	ON	ON	ON
SW8	ON	OFF	ON	OFF	OFF	ON	ON	OFF
SW9	ON	ON	ON	OFF	OFF	OFF	OFF	OFF
SW10	OFF	ON	OFF	ON	OFF	OFF	OFF	OFF

Table 63. Super Hydra P5020 Default Switch Settings

	1	2	3	4	5	6	7	8
SW1	OFF	ON	ON	OFF	ON	OFF	ON	OFF
SW2	OFF	OFF	ON	OFF	OFF	OFF	OFF	ON
SW3	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF
SW4	ON	ON	ON	ON	ON	OFF	ON	ON
SW5	OFF	OFF	OFF	ON	OFF	ON	OFF	OFF
SW6	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON
SW7	OFF	OFF	OFF	OFF	ON	ON	ON	ON
SW8	ON	OFF	ON	OFF	OFF	OFF	ON	OFF
SW9	ON	ON	ON	OFF	OFF	OFF	OFF	OFF
SW10	OFF	OFF	OFF	ON	OFF	OFF	OFF	OFF
SW11	OFF	OFF	OFF	OFF	OFF	ON	OFF	OFF
SW12	ON	OFF	ON	ON	OFF	ON	OFF	ON
SW13	ON	OFF	ON	OFF	OFF	ON	OFF	ON

NOTE

Please note the following for the Super Hydra P3041 and P5020 :

SW1-11 have ON = 1 and OFF = 0

SW12 and SW13 have ON = 0 and OFF = 1

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 64. Boot Device Scenarios

Alternate Boot Device	Change the following switch settings (OFF=0/ON=1)
SD card boot	SW1 [1:5] = 00110.
SPI Flash boot	SW1 [1:5] = 00101.
NAND flash boot	SW1[1:5] = 01001. SW7[1:4] =1001.

4.4.11.3 U-Boot Environment Variables

4.4.11.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

If you plan to use optical 10G interfaces, you must add information to "hwconfig". This is important. The optical interface will operate erratically without it. It is easiest to do this using the U-Boot "*editenv*" command. Whatever you type will be appended to "hwconfig". The text you need to append is:

```
;usb2:dr_mode=peripheral,phy_type=utmi
```

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;usb2:dr_mode=peripheral,phy_type=utmi
```

NOTE

USB2 has an OTG connector on P3041/P5020, but the OTG function is not available on these boards, so the default mode for USB2 is set to '*device*' by adding '*usb2:dr_mode=peripheral,phy_type=utmi*' in hwconfig. If host mode is needed, please remove the define for '*usb2...*' from hwconfig. Please avoid connecting USB2 to another host when USB2 is working in host mode. This might cause damage to the two hosts.

4.4.11.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.11.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for P3041/P5020 version information (e.g., P3041E version 2.0) and also for the version of FMan microcode currently flashed on the P3041DS/P5020DS (e.g. microcode version 106.1.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 1.0:

```
fsl_fman_ucode_p3041_r1.0_106_1_15.bin
fsl_fman_ucode_p5020_r1.0_106_1_15.bin
```

- For silicon revision 1.1:

```
fsl_fman_ucode_p3041_r1.1_106_1_15.bin
```

- For silicon revision 2.0:

```
fsl_fman_ucode_p3041_r2.0_106_1_18.bin
fsl_fman_ucode_p5020_r2.0_106_1_18.bin
```

NOTE

(i) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(ii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iii) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.11.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK contains RCW binaries for use on the P3041/P5020DS:

- RR_HXAPNRP_0x33/rcw_12g_1500mhz.bin (P3041DS)
- RR_HXAPNSP_0x36/rcw_15g_1500mhz.bin (P3041DS - default)
- RR_HXAPNRP_0x33/rcw_12g_2000mhz.bin (P5020DS)
- RR_HXAPNSP_0x36/rcw_15g_200mhz.bin (P5020DS - default)

The RCW directories' names for the P3041/P5020DS conform to the following naming convention:

```
ab_cdefghi_j
```

Table 65. P3041/P5020DS Naming Convention Legend

Slot	Convention
a	'R' if RGMII@DTSEC4 is supported 'N' if not available/not used
b	'R' if RGMII@DTSEC5 is supported 'N" if not available/not used
c [What is available in Slot 1] d [What is availabel in Slot 2] e [What is available in Slot 3] f [What is available in Slot 4] g [What is available in Slot 5] h [What is available in Slot 6] i [What is available in Slot 7]	'N' if not available/not used 'P' if PCIe 'X' if XAUI 'R' if SRIO 'S' if SGMII 'H' if SATA 'A' is AURORA
j	Hex value of serdes protocol value

For exmaple,

```
NR_HXAPNSP_0x36
```

means:

- No RGMII@DTSEC4
- RGMII@DTSEC5
- SATA [Slot 1 not used]
- XAUI on Slot 2
- AURORA on Slot 3
- PCIe on Slot 4
- Slot 5 is not used
- SGMII on Slot 6
- PCIe on Slot 7
- SERDES Protocol is 0x35

Below is a table that maps the different ports available on P3041/P5020DS and their names in different environments.

Table 66. Port Map

Label on P3041/P5020DS	Port in U-Boot	Port in Linux	FMan Address
N/A	FM1@DTSEC1	fm1-gb0	0xffe4e0000
N/A	FM1@DTSEC2	fm1-gb1	0xffe4e2000
N/A	FM1@DTSEC3	fm1-gb2	0xffe4e4000
N/A	FM1@DTSEC4	fm1-gb3	0xffe4e6000
N/A	FM1@DTSEC5	fm1-gb4	0xffe4e8000
N/A	FM1@TGEC1	fm1-10g	0xffe4f0000

4.4.11.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
0xfe0000000	0xfeffffffff	eLBC	256 MB
0xff4000000	0xff41fffff	Buffer manager software portal	2 MB
0xff4200000	0xff43fffff	Queue manager software portal	2 MB
0xfffdf0000	0xfffdf0fff	FPGA	4 KB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Definition	Size
0xf00000000	0xf01ffffff	DCSR	32 MB
0xffa000000	0xffa0fffff	NAND	1 MB
0xc20000000	0xc2fffffff	Serial Rapid I/O 1	256 MB
0xc00000000	0xc1fffffff	PCI Express 1	512 MB
0xff8000000	0xff800ffff	PCI Express 1 I/O	64 KB
0xc40000000	0xc5fffffff	PCI Express 3	512 MB
0xff8020000	0xff802ffff	PCI Express 3 I/O	64 KB
0x00000000	0x7fffffff	Memory controller	2 GB

4.4.11.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration (for P3041/P5020DS, SW7[0:3]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps bank 0 with the alternate bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
CPU0:  P3041E, Version: 2.0, (0x82190320)
Core:  e500mc, Version: 3.2, (0x80230032)
Clock Configuration:
       CPU0:1500 MHz, CPU1:1500 MHz, CPU2:1500 MHz, CPU3:1500 MHz,
       CCB:750  MHz,
       DDR:666.667 MHz (1333.333 MT/s data rate) (Asynchronous), LBC:93.750 MHz
       FMAN1: 583.333 MHz
       QMAN:  375 MHz
       PME:   375 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 12600000 00000000 241c0000 00000000
       00000010: d8984a01 03002000 de800000 41000000
       00000020: 00000000 00000000 00000000 10070000
       00000030: 00000000 00000000 00000000 00000000
Board: P3041DS, Sys ID: 0x1c, Sys Ver: 0x12, FPGA Ver: 0x05, vBank: 4
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>pixis reset
```

- Switch to alternate bank:

```
=>pixis_reset altbank
```

The table below shows the P3041/P5080DS NOR flash memory map

Table 67. NOR Flash Memory Map

Range Start	Range End	Definition	Size
0xeff40000	0xefffffff	U-Boot (current bank)	768 KB
0xeff20000	0xeff3ffff	U-Boot env (current bank)	128 KB
0xeff00000	0xeff1ffff	FMAN Ucode (current bank)	128 KB
0xed300000	0xefeffffff	Rootfs (alternate bank)	44 MB
0xed000000	0xed2fffff	Guest image #3 (alternate bank)	3 MB
0xecd00000	0xecffffff	Guest image #2 (alternate bank)	3 MB
0xeca00000	0xeccfffff	Guest image #1 (alternate bank)	3 MB
0xec900000	0xec9fffff	HV config device tree (alternate bank)	1 MB
0xec800000	0xec8fffff	Hardware device tree (alternate bank)	1 MB
0xec700000	0xec7fffff	HV.uImage (alternate bank)	1 MB
0xec020000	0xec6fffff	Linux.uImage (alternate bank)	~7 MB
0xec000000	0xec01ffff	Rcw (alternate bank)	128 KB
0xebf40000	0xebffffff	U-Boot (alternate bank)	768 KB

Table continues on the next page...

Table 67. NOR Flash Memory Map (continued)

0xebf20000	0xebf3ffff	U-Boot env (alternate bank)	128 KB
0xebf00000	0xebf1ffff	FMAN ucode (alternate bank)	128 KB
0xe9300000	0xebefffff	Rootfs (current bank)	44 MB
0xe9000000	0xe92fffff	Guest image #3 (current bank)	3 MB
0xe8d00000	0xe8ffffff	Guest image #2 (current bank)	3 MB
0xe8a00000	0xe8cfffff	Guest image #1 (current bank)	3 MB
0xe8900000	0xe89fffff	HV config device tree (current bank)	1 MB
0xe8800000	0xe88fffff	Hardware device tree (current bank)	1 MB
0xe8700000	0xe87fffff	HV.uImage (current bank)	1 MB
0xe8020000	0xe86fffff	Linux.uImage (current bank)	~7 MB
0xe8000000	0xe801ffff	RCW (current bank)	128 KB

4.4.11.8 Programming a New U-Boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *pixis reset* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
```

```
=>protect on <u-boot_start_addr> +$filesize
=>pixis_reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>pixis_reset altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>. <fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>pixis_reset altbank
```

4.4.11.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.11.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

```
=>tftp 1000000 <uImage_name>
=>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>tftp 2000000 <platform_dtb_name>
=>bootm 1000000 5000000 2000000
```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.11.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

```
=>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
=>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
=>saveenv
```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
```

```
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.11.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

   b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
```

```
=>bootm 1000000 - c00000
```

Now the board will boot up with NFS filesystem.

4.4.11.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

   ```
   #fdisk /dev/sdb
   ```

3. Use the mkfs.ext2 command to create the filesystems.

   ```
   # mkfs.vfat /dev/sdb1
   # mkfs.ext2 /dev/sdb2
   ```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

   ```
   # mkdir /temp
   # mount /dev/sdb2 /temp
   # cd /temp
   ```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

   ```
   # mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
   ```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

   ```
   # cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   ```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

   ```
   # cp uImage and uImage.dtb to /temp/boot folder
   # umount /temp
   ```

8. Plug in the SD card to the target board and power on.

4.4.11.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

   ```
   # fdisk /dev/sdx
   ```

3. Use the mkfs command to create the filesystems.

   ```
   # mkfs.ext2 /dev/sdx1
   ```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

   ```
   # mkdir /temp
   # mount /dev/sdx1 /temp
   # cd /temp
   ```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

   ```
   # mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
   ```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

   ```
   # cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
   # sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   ```

7. Make sure copy the target kernel image and dtb to sata.

   ```
   # cp uImage and target.dtb to /temp/boot folder
   # umount /temp
   ```

8. Connect the SATA hard drive to the target board and power on.

4.4.11.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

```
=>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>erase e9300000 +$filesize
=>cp.b 1000000 e9300000 $filesize
```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

```
=>tftp 1000000 hv.uImage
=>erase e8700000 +$filesize
=>cp.b 1000000 e8700000 $filesize
```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

```
=>tftp 1000000 <platform_name>-usdpaa.dtb
=>erase e8800000 +$filesize
=>cp.b 1000000 e8800000 $filesize
```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.12 P4080DS

4.4.12.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.12.2 Switch Settings

The P4080 Development System has user selectable switches for evaluating different boot options for the P4080 device. The table below lists the default switch settings. For a description of these settings, see P4080DS User Guide.

Table 68. Default Switch Settings

	1	2	3	4	5	6	7	8

Table continues on the next page...

Table 68. Default Switch Settings (continued)

SW1	OFF[0]	ON [1]	ON [1]	OFF [0]	ON [1]	OFF [0]	ON [1]	ON [1]
SW2	ON	ON	ON	ON	ON	ON	ON	ON
SW3	OFF	OFF	OFF	OFF	ON	ON	OFF	OFF
SW4	ON	ON	ON	ON	ON	ON	ON	OFF
SW5	ON	ON	ON	ON	ON	ON	ON	ON
SW6	OFF	OFF	OFF	OFF	OFF	OFF	ON	OFF
SW7	OFF	OFF	OFF	OFF	ON	ON	ON	ON
SW8	ON	OFF	ON	ON	OFF	OFF	OFF	OFF
SW9	ON	ON	ON	OFF	OFF	ON	OFF	OFF
SW10	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 69. Boot Device Scenarios

Alternate Boot Devices	**Change the following switch settings (0=OFF / 1=ON)**
SPI boot	SW1 [1:5] = 00101
SD Flash boot	SW1 [1:5] = 00110

4.4.12.3 U-Boot Environment Variables

The P4080DS is preloaded with U-Boot and Linux images. The following sections will guide the users on how to set the U-Boot environment and configure the U-Boot network parameters.

4.4.12.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

If you plan to use optical 10G interfaces, you must add information to "hwconfig". This is important. The optical interface will operate erratically without it. It is easiest to do this using the U-Boot "*editenv*" command. Whatever you type will be appended to "hwconfig". The text you need to append is:

```
;fsl_fm2_xaui_phy:xfi
```

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;fsl_fm2_xaui_phy:xfi
```

4.4.12.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.12.4 Frame Manager (FMan) Microcode

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for P4080 version information (e.g., P4080E version 3.0) and also for the version of FMan microcode currently flashed on the P4080DS (e.g. microcode version 106.2.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 2.0:

```
fsl_fman_ucode_p4080_r2.0_106_2_15.bin
```

- For silicon revision 3.0:

```
fsl_fman_ucode_p4080_r3.0_106_2_18.bin
```

NOTE

(i) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(ii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iii) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.12.5 RCW (Reset Configuration Word) and Ethernet Interfaces

The following section provides guidance on how to configure the RCW and Ethernet interfaces based on P4080DS capabilities.

QorIQ SDK 2.0 contains RCW binaries for use on the P4080DS:

- N_RRRSS_0x16/rcw_all_1500mhz_rev{2,3}.bin
- R_PPSXX_0xe/rcw_2sgmii_1500mhz_rev{2,3}.bin
- R_PPSXN_0x10/rcw_13g_1500mhz_rev{2,3}.bin (default)
- R_PPSXN_0x10/rcw_5g_1500mhz_rev{2,3}.bin

The RCW directories' names for the P4080DS conform to the following naming convention:

```
a_bcdef_g
```

Table 70. P4080DS Naming Convention Legend

Label	Convention
a	'R' if RGMII is supported 'N' if not available/not used

Table continues on the next page...

Table 70. P4080DS Naming Convention Legend (continued)

b [what is available in Slot 1] c [what is available in Slot 2] d [what is available in Slot 3] e [what is available in Slot 4] f [what is available in Slot 5]	For the Slots (b...f) 'N' if not available/not used 'P' if PCIe 'X' if XAUI 'R' if SRIO 'S' if SGMII 'A' is AURORA
g	Hex value of serdes protocol value

For example,

```
R_PPSXX_0xe
```

means:

- RGMII is enabled
- PCIe on slot 1
- PCIe on slot 2
- SGMII on slot 3
- XAUI on slot 4
- XAUI on slot 5

Below is a table that maps the different ports available on P4080DS and their names in different environments.

Table 71. Port Map

Label on ATX chassis	Port in U-boot	Port in Linux	FMAN Address
N/A	FM1@DTSEC2	fm1-gb1	0xffe4e2000
N/A	FM1@TGEC1	fm1-10g	0xffe4f0000
N/A	FM2@DTSEC3	fm2-gb2	0xffe5e4000
N/A	FM2@DTSEC4	fm2-gb3	0xffe5e6000
N/A	FM2@TGEC1	fm2-10g	0xffe5f0000

4.4.12.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
0xfe0000000	0xfeffffff	eLBC	256 MB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Definition	Size
0xff4000000	0xff41fffff	Buffer manager software portal	2 MB
0xff4200000	0xff43fffff	Queue manager software portal	2 MB
0xfffdf0000	0xfffdf0fff	FPGA	4 KB
0xf00000000	0xf01ffffff	DCSR	32 MB
0xffa000000	0xffa0fffff	NAND	1 MB
0xc20000000	0xc2fffffff	Serial Rapid I/O 1	256 MB
0xc00000000	0xc1fffffff	PCI Express 1	512 MB
0xff8000000	0xff800ffff	PCI Express 1 I/O	64 KB
0xc40000000	0xc5fffffff	PCI Express 3	512 MB
0xff8020000	0xff802ffff	PCI Express 3 I/O	64 KB
0x00000000	0x7fffffff	Memory controller	2 GB

4.4.12.7 Flash Bank Usage

The NOR flash on the board can be seen as eight flash banks. The board DIP switch (for P4080DS, SW7[0:3]) configuration preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps bank 0 with the alternate bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log.

```
CPU0:  P4080E, Version: 3.0, (0x82080030)
Core:  e500mc, Version: 3.1, (0x80230031)
Clock Configuration:
       CPU0:1499.985 MHz, CPU1:1499.985 MHz, CPU2:1499.985 MHz, CPU3:1499.985 MHz,
       CPU4:1499.985 MHz, CPU5:1499.985 MHz, CPU6:1499.985 MHz, CPU7:1499.985 MHz,
       CCB:799.992 MHz,
       DDR:649.994 MHz (1299.987 MT/s data rate) (Asynchronous), LBC:99.999 MHz
       FMAN1: 599.994 MHz
       FMAN2: 599.994 MHz
```

```
       QMAN:  399.996 MHz
       PME:   599.994 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 105a0000 00000000 1e1e181e 0000cccc
       00000010: 3842440c 3c3c2000 de800000 e1000000
       00000020: 00000000 00000000 00000000 008b6000
       00000030: 00000000 00000000 00000000 00000000
Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0c, vBank: 4
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>pixis_reset
```

- Switch to alternate bank:

```
=>pixis_reset altbank
```

The table below shows the P4080DS NOR flash memory map.

Table 72. NOR Flash Memory Map

Range Start	Range End	Definition	Size
0xeff40000	0xefffffff	U-Boot (current bank)	768KB
0xeff20000	0xeff3ffff	U-Boot env (current bank)	128KB
0xeff00000	0xeff1ffff	FMAN Ucode (current bank)	128KB
0xed300000	0xefefffff	Rootfs (alternate bank)	44MB
0xed000000	0xed2fffff	Guest image #3 (alternate bank)	3MB
0xecd00000	0xecffffff	Guest image #2 (alternate bank)	3MB
0xeca00000	0xeccfffff	Guest image #1 (alternate bank)	3MB
0xec900000	0xec9fffff	HV config device tree (alternate bank)	1MB
0xec800000	0xec8fffff	Hardware device tree (alternate bank)	1MB

Table continues on the next page...

Table 72. NOR Flash Memory Map (continued)

0xec700000	0xec7fffff	HV.uImage (alternate bank)	1MB
0xec020000	0xec6fffff	Linux.uImage (alternate bank)	~7MB
0xec000000	0xec01ffff	RCW (alternate bank)	128KB
0xebf40000	0xebffffff	U-Boot (alternate bank)	768KB
0xebf20000	0xebf3ffff	U-Boot env (alternate bank)	128KB
0xebf00000	0xebf1ffff	FMAN ucode (alternate bank)	128KB
0xe9300000	0xebefffff	Rootfs (current bank)	44MB
0xe9000000	0xe92fffff	Guest image #3 (current bank)	3MB
0xe8d00000	0xe8ffffff	Guest image #2 (current bank)	3MB
0xe8a00000	0xe8cfffff	Guest image #1 (current bank)	3MB
0xe8900000	0xe89fffff	HV config device tree (current bank)	1MB
0xe8800000	0xe88fffff	Hardware device tree (current bank)	1MB
0xe8700000	0xe87fffff	HV.uImage (current bank)	1MB
0xe8020000	0xe86fffff	Linux.uImage (current bank)	~7MB
0xe8000000	0xe801ffff	RCW (current bank)	128K

4.4.12.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *pixis_reset altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>pixis_reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>pixis_reset altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>. <fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>pixis_reset altbank
```

4.4.12.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.12.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

```
=>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
=>saveenv
```

NOTE

ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

```
=>tftp 1000000 <uImage_name>
=>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>tftp 2000000 <platform_dtb_name>
=>bootm 1000000 5000000 2000000
```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.12.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

```
=>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
=>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
=>saveenv
```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
```

```
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.12.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

   b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
```

```
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
=>bootm 1000000 - c00000
```

   Now the board will boot up with NFS filesystem.

4.4.12.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

```
#fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to create the filesystems.

```
# mkfs.vfat /dev/sdb1
# mkfs.ext2 /dev/sdb2
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdb2 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

```
# cp uImage and uImage.dtb to /temp/boot folder
# umount /temp
```

8. Plug in the SD card to the target board and power on.

4.4.12.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

```
# fdisk /dev/sdx
```

3. Use the mkfs command to create the filesystems.

```
# mkfs.ext2 /dev/sdx1
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdx1 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure copy the target kernel image and dtb to sata.

```
# cp uImage and target.dtb to /temp/boot folder
# umount /temp
```

8. Connect the SATA hard drive to the target board and power on.

4.4.12.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

   TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

   ```
   =>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>erase e9300000 +$filesize
   =>cp.b 1000000 e9300000 $filesize
   ```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

   ```
   =>tftp 1000000 hv.uImage
   =>erase e8700000 +$filesize
   =>cp.b 1000000 e8700000 $filesize
   ```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

   ```
   =>tftp 1000000 <platform_name>-usdpaa.dtb
   =>erase e8800000 +$filesize
   =>cp.b 1000000 e8800000 $filesize
   ```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

   ```
   =>boot
   ```

4.4.13 P5040DS

4.4.13.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.13.2 Switch Settings

The Development System has user selectable switches for evaluating different boot options for the P5040 device. The table below lists the default switch settings. For a description of these settings, see P5040DS Hardware Getting Started Guide.

Table 73. Default Switch Settings

	1	2	3	4	5	6	7	8
SW1	OFF	ON	ON	OFF	ON	OFF	ON	OFF
SW2	ON	OFF	OFF	OFF	ON	ON	OFF	ON
SW3	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF
SW4	ON	ON	ON	ON	ON	OFF	OFF	ON
SW5	OFF	OFF	OFF	ON	OFF	ON	OFF	OFF
SW6	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON
SW7	OFF	OFF	OFF	OFF	ON	ON	ON	ON
SW8	ON	OFF	ON	OFF	OFF	OFF	ON	OFF
SW9	ON	ON	ON	OFF	OFF	OFF	OFF	OFF
SW10	OFF	OFF	OFF	ON	OFF	OFF	OFF	OFF
SW11	OFF	OFF	ON	ON	ON	OFF	OFF	ON
SW12	ON	OFF	ON	ON	OFF	ON	OFF	ON
SW13	OFF	ON	ON	ON	ON	OFF	OFF	ON

NOTE

SW1-11 have ON = 1 and OFF = 0

SW12 and SW13 have ON = 0 and OFF = 1

Note that with the default settings, the following are selected:

SW1[1:5] = 01101:

RCW configuration source is eLBC GPCM 16-bit NOR flash.

SW1[6:8] = 010:

1.5V DDR3 technology

Continued Boot

Disabled NAND Flash ECC

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 74. Boot Device Scenarios

Alternate Boot Devices	Change these DIP switch settings
SD card booting	SW1 [1:5] = 00110
SPI booting Scenario	SW1 [1:5] = 00101
NAND booting Scenario	SW1[1:5] = 01001 SW7[1:4] = 1001

4.4.13.3 U-Boot Environment Variables

4.4.13.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

If you plan to use optical 10G interfaces, you must add information to "hwconfig". This is important. The optical interface will operate erratically without it. It is easiest to do this using the U-Boot "*editenv*" command. Whatever you type will be appended to "hwconfig". The text you need to append is:

```
;fsl_fm2_xaui_phy:xfi
```

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;fsl_fm2_xaui_phy:xfi
```

4.4.13.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
```

```
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.13.4 Frame Manager Microcode (FMan Ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for P5040 version information (e.g., P5040E version 2.1) and also for the version of FMan microcode currently flashed on the P5040DS (e.g. microcode version 106.1.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 1.0:

```
fsl_fman_ucode_p5040_r1.0_106_1_15.bin
```

- For silicon revision 2.0:

```
fsl_fman_ucode_p5040_r2.0_106_1_15.bin
```

- For silicon revision 2.1:

```
fsl_fman_ucode_p5040_r2.1_106_1_18.bin
```

NOTE

(i)Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(ii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iii) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.13.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK 2.0 contains the following RCW binary for use on the P5040DS:

- RR_XXSNSpP_0x02/rcw_26g_2267mhz.bin (default)

The RCW directories' names for the P5040DS conform to the following naming convention:

```
ab_cdefghi_j
```

Table 75. P5040DS Naming Convention Legend

Slot	Convention
a	'R' if FM1@DTSEC5 is supported 'N' if not available/not used
b	'R' if FM2@DTSEC5 is supported 'N" if not available/not used
c [What is available in Slot 1] d [What is available in Slot 2] e [What is available in Slot 3] f [What is available in Slot 4] g [What is available in Slot 5] h [What is available in Slot 6] i [What is available in Slot 7]	'N' if not available/not used 'p' if PCIe x2 'P' if PCIe x4 'X' if XAUI 'R' if SRIO 'S' if SGMII 'H' if SATA 'A' is AURORA
j	Hex value of serdes protocol value

For example,

```
RR_XXSNSpP_0x02
```

means:

- RGMII FM1@DTSEC5
- RGMII FM2@DTSEC5
- XAUI on slots 1 and 2
- PCIe on slots 6 and 7

- SGMII on slots 3 and 5
- SERDES Protocol is 0x02

Below is a table that maps the different ports available on P5040DS and their names in different environments.

Table 76. Port Map

Label on P5040DS	Port in U-boot	Port in Linux	FMAN Address
N/A	FM1@DTSEC3	fm1-gb2	0xffe4e4000
N/A	FM1@DTSEC4	fm1-gb3	0xffe4e6000
N/A	FM1@DTSEC5	fm1-gb4	0xffe4e8000
N/A	FM1@TGEC1	fm1-10g	0xffe4ef0000
N/A	FM2@DTSEC3	fm2-gb2	0xffe5e4000
N/A	FM2@DTSEC4	fm2-gb3	0xffe5e6000
N/A	FM2@DTSEC5	fm2-gb4	0xffe5e8000
N/A	FM2@TGEC1	fm2-10g	0xffe5ef0000

4.4.13.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
0xfe0000000	0xfeffffffff	eLBC	256 MB
0xff4000000	0xff41fffff	Buffer manager software portal	2 MB
0xff4200000	0xff43fffff	Queue manager software portal	2 MB
0xfffdf0000	0xfffdf0fff	FPGA	4 KB
0xf00000000	0xf01ffffff	DCSR	32 MB
0xffa000000	0xffa0fffff	NAND	1 MB
0xc20000000	0xc2fffffff	Serial Rapid I/O 1	256 MB
0xc00000000	0xc1fffffff	PCI Express 1	512 MB
0xff8000000	0xff800ffff	PCI Express 1 I/O	64 KB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Definition	Size
0xc40000000	0xc5fffffff	PCI Express 3	512 MB
0xff8020000	0xff802ffff	PCI Express 3 I/O	64 KB
0x00000000	0x7fffffff	Memory controller	2 GB

4.4.13.7 Flash Bank Usage

The NOR flash on the board can be seen as eight flash banks. The board DIP switch configuration (for P5040DS, SW7[1:4]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps bank 0 with the alternate bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
CPU0:  P5040E, Version: 2.1, (0x820c0021)
Core:  e5500, Version: 1.2, (0x80240012)
Clock Configuration:
       CPU0:2266.667 MHz, CPU1:2266.667 MHz, CPU2:2266.667 MHz, CPU3:2266.667 MHz,
       CCB:800  MHz,
       DDR:800  MHz (1600 MT/s data rate) (Asynchronous), LBC:100  MHz
       FMAN1: 600 MHz
       FMAN2: 600 MHz
       QMAN:  400 MHz
       PME:   400 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 8cd80000 00000000 a2a29200 00440000
       00000010: 089c4400 00283000 de800000 61000000
       00000020: 00000000 00000000 00000000 10070000
       00000030: 00000000 00000000 00000000 00000000
Board: P5040DS, Sys ID: 0x20, Sys Ver: 0x02, FPGA Ver: 0x03, vBank: 4
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>pixis_reset
```

- Switch to alternate bank:

```
=>pixis_reset altbank
```

The table below shows the P5040DS NOR flash memory map.

Table 77. NOR flash Memory Map

Range Start	Range End	Definition	Size
0xeff40000	0xefffffff	U-Boot (current bank)	768KB
0xeff20000	0xeff3ffff	U-Boot env (current bank)	128KB
0xeff00000	0xeff1ffff	FMAN Ucode (current bank)	128KB
0xed300000	0xefeffffff	Rootfs (alternate bank)	44MB
0xed000000	0xed2fffff	Guest image #3 (alternate bank)	3MB
0xecd00000	0xecffffff	Guest image #2 (alternate bank)	3MB
0xeca00000	0xeccfffff	Guest image #1 (alternate bank)	3MB
0xec900000	0xec9fffff	HV config device tree (alternate bank)	1MB
0xec800000	0xec8fffff	Hardware device tree (alternate bank)	1MB
0xec700000	0xec7fffff	HV.uImage (alternate bank)	1MB
0xec020000	0xec6fffff	Linux.uImage (alternate bank)	~7MB
0xec000000	0xec01ffff	RCW (alternate bank)	128KB
0xebf40000	0xebffffff	U-Boot (alternate bank)	768KB
0xebf20000	0xebf3ffff	U-Boot env (alternate bank)	128KB
0xebf00000	0xebf1ffff	FMAN ucode (alternate bank)	128KB
0xe9300000	0xebefffff	Rootfs (current bank)	44MB
0xe9000000	0xe92fffff	Guest image #3 (current bank)	3MB

Table continues on the next page...

Table 77. NOR flash Memory Map (continued)

0xe8d00000	0xe8ffffff	Guest image #2 (current bank)	3MB
0xe8a00000	0xe8cfffff	Guest image #1 (current bank)	3MB
0xe8900000	0xe89fffff	HV config device tree (current bank)	1MB
0xe8800000	0xe88fffff	Hardware device tree (current bank)	1MB
0xe8700000	0xe87fffff	HV.uImage (current bank)	1MB
0xe8020000	0xe86fffff	Linux.uImage (current bank)	~7MB
0xe8000000	0xe801ffff	RCW (current bank)	128K

4.4.13.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *pixis_reset altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>pixis_reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location

of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>pixis_reset altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *pixis_reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>. <fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>pixis_reset altbank
```

4.4.13.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.13.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 1000000 <uImage_name>
   =>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 2000000 <platform_dtb_name>
   =>bootm 1000000 5000000 2000000
   ```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.13.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   =>saveenv
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

   ```
   =>tftp 1000000 <uImage name>
   =>protect off <kernel_start_addr> +$filesize
   =>erase <kernel_start_addr> +$filesize
   =>cp.b 1000000 <kernel_start_addr> $filesize
   =>protect on <kernel_start_addr> +$filesize
   ```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

   ```
   =>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>protect off <ramdisk_start_addr> +$filesize
   =>erase <ramdisk_start_addr> +$filesize
   =>cp.b 2000000 <ramdisk_start_addr> $filesize
   =>protect on <ramdisk_start_addr> +$filesize
   ```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

   ```
   =>tftp c00000 <platform dtb name>
   =>protect off <dtb_start_addr> +$filesize
   =>erase <dtb_start_addr> +$filesize
   =>cp.b c00000 <dtb_start_addr> $filesize
   =>protect on <dtb_start_addr> +$filesize
   ```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

   ```
   =>boot
   ```

or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.13.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   <nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/nfs restart
   ```

   NOTE

   nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>:
   <your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
   =>setenv netdev <ethx>
   =>saveenv
   ```

   NOTE

   <ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

   ```
   =>tftp 1000000 <uImage name>
   =>tftp c00000 <platform dtb name>
   =>bootm 1000000 - c00000
   ```

   Now the board will boot up with NFS filesystem.

4.4.13.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

   ```
   #fdisk /dev/sdb
   ```

3. Use the mkfs.ext2 command to create the filesystems.

   ```
   # mkfs.vfat /dev/sdb1
   # mkfs.ext2 /dev/sdb2
   ```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

   ```
   # mkdir /temp
   # mount /dev/sdb2 /temp
   # cd /temp
   ```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

   ```
   # mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
   ```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

   ```
   # cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   ```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

   ```
   # cp uImage and uImage.dtb to /temp/boot folder
   # umount /temp
   ```

8. Plug in the SD card to the target board and power on.

4.4.13.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

   ```
   # fdisk /dev/sdx
   ```

3. Use the mkfs command to create the filesystems.

   ```
   # mkfs.ext2 /dev/sdx1
   ```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

   ```
   # mkdir /temp
   # mount /dev/sdx1 /temp
   # cd /temp
   ```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

   ```
   # mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
   ```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

   ```
   # cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
   # sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   ```

7. Make sure copy the target kernel image and dtb to sata.

   ```
   # cp uImage and target.dtb to /temp/boot folder
   # umount /temp
   ```

8. Connect the SATA hard drive to the target board and power on.

4.4.13.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

```
=>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>erase e9300000 +$filesize
=>cp.b 1000000 e9300000 $filesize
```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

```
=>tftp 1000000 hv.uImage
=>erase e8700000 +$filesize
=>cp.b 1000000 e8700000 $filesize
```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

```
=>tftp 1000000 <platform_name>-usdpaa.dtb
=>erase e8800000 +$filesize
=>cp.b 1000000 e8800000 $filesize
```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.14 T1023RDB

4.4.14.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.14.2 Switch Settings

The T1023RDB has user selectable switches for evaluating different boot options for the T1023 device. The table below lists the default switch settings. For a description of these settings, see T1023RDB User Guide.

Table 78. Default Switch Settings (NOR boot on Rev-C)

	1	2	3	4	5	6	7	8

Table continues on the next page...

Table 78. Default Switch Settings (NOR boot on Rev-C) (continued)

SW1	ON [0]	ON [0]	ON [0]	OFF [1]	ON [0]	OFF [1]	OFF [1]	OFF [1]
SW2	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF
SW3	OFF	OFF	OFF	ON	ON	ON	ON	OFF

Note that with the default settings, the following are selected:

on Rev-B: SW2[1:8] = '00100010', SW3[1] ='1' for SPI boot (NOR flash is not available on Rev-B board.)

on Rev-C: SW1[1:8] = '00010111', SW2[1] ='1' for NOR boot

```
For Rev-C:
   SW3[5] = '0' for bank0(or 'switch bank0' by software)
   SW3[5] = '1' for bank4(or 'switch bank4' by software)

   SW3[3] = '1' for SD card(or 'switch sd' by software)
   SW3[3] = '0' for eMMC (or 'switch emmc' by software)
```

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 79. Boot Device Scenarios

Alternate Boot Devices	**Change the following switch settings (0 = ON, 1 = OFF)**
NOR boot (default boot on Rev-C)	SW1 [1:8] = 0001 0111 SW2 [1:8] = 1111 1111 SW3 [1:8] = 1110 0001
NAND boot	SW1 [1:8] = 1000 0010 SW2 [1:8] = 1011 1101 SW3 [1:8] = 0111 0001 on Rev-B: shunt rework header pin 1-3 and 2-4
SPI boot (default boot on Rev-B)	SW1 [1:8] = 0010 0010 SW2 [1:8] = 1011 1101 SW3 [1:8] = 0110 0001 on Rev-B: shunt rework header pin 1-2 and 3-4
SD boot	SW1 [1:8] = 0010 0000 SW2 [1:8] = 0011 1101 SW3 [1:8] = 0110 0001

4.4.14.3 U-Boot Environment Variables

4.4.14.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=null
```

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=null
```

4.4.14.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
```

```
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.14.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for T1023 version information (e.g., T1023E version 1.0) and also for the version of FMan microcode currently flashed on the T1023RDB (e.g. microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 1.0:

```
fsl_fman_ucode_t1024_r1.0_106_4_18.bin
fsl_fman_ucode_t1024_r1.0_107_4_2.bin
fsl_fman_ucode_t1024_r1.0_108_4_9.bin
```

NOTE

(i) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(ii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

4.4.14.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK contains RCW binaries for use on T1023RDB:

- RNSS_NPP_77/rcw_77_1400MHz.bin (default)
- RNSS_NPP_77/rcw_77_1200MHz.bin

The RCW directories' names for the T1023RDB conform to the following naming convention:

```
abcd_efg_h
```

Table 80. T1023RDB Naming Convention Legend

Slot	Convention
a	'R' if RGMII1 @DTSEC4 is supported 'N' if not available/not used
b	'R' if RGMII2 @DTSEC3 is supported 'N" if not available/not used

Table continues on the next page...

Table 80. T1023RDB Naming Convention Legend (continued)

c	'S' if FM1@DTSEC1 1G SGMII is supported 'N" if not available/not used
d	'S' if FM1@DTSEC3 2.5G SGMII is supported 'N" if not available/not used
e [What is availabel in Mini-PCI connector 1] f [What is availabel in Mini-PCI connector 2] g	'N' if not available/not used 'P' if PCIe is available 'X' if XAUI is available 'R' if SRIO is available 'S' if SGMII is available 'H' if SATA is available 'A' is AURORA is available
h	Hex value of SerDes protocol

For example,

```
RNSS_NPP_77
```

means:

- 1G RGMII@DTSEC4
- 1G SGMII@DTSEC1
- 2.5G SGMII@DTSEC3
- PCIE [Mini-PCI connector 1]
- PCIE [Mini-PCI connector 2]
- SerDes Protocol is 0x77

The QorIQ SDK release supports SerDes 0x77 default configuration.

- `RNSS_NPP_77 (1x 2.5G SGMII + 1x 1G SGMII + 1x 1G RGMII + 2x PCIe)`

When using the default configuration, the network interfaces, which include one 1Gbps RGMII ports, one 1Gbps SGMII ports and one 2.5G SGMII port labeled as the following in U-Boot:

FM1@DTSEC4

FM1@DTSEC1

FM1@DTSEC3

Below is a table that maps the different ports available on T1023 RDB and their names in different environments.

Table 81. Port Map

Label on 1U box	Port in U-boot	Port in Linux	FMan Address	Comments

Table continues on the next page...

Table 81. Port Map (continued)

ETH1	FM1@DTSEC4	fm1-mac4	0xffe4e6000	1Gbps RGMII (RTL8211F PHY)
ETH2	FM1@DTSEC1	fm1-mac1	0xffe4e0000	1Gbps SGMII (RTL8211F PHY)
ETH3	FM1@DTSEC3	fm1-mac3	0xffe4e4000	2.5Gbps SGMII (AQR105 PHY)

4.4.14.6 System Memory Map

System memory map on T1023RDB

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
0xF_FF80_0000	0xF_FF80_FFFF	IFC - NAND Flash	64KB
0xF_FE00_0000	0xF_FEFF_FFFF	CCSRBAR	16MB
0xF_F802_0000	0xF_F802_FFFF	PCI Express 3 I/O Space	64KB
0xF_F801_0000	0xF_F801_FFFF	PCI Express 2 I/O Space	64KB
0xF_F800_0000	0xF_F800_FFFF	PCI Express 1 I/O Space	64KB
0xF_F600_0000	0xF_F7FF_FFFF	Queue manager software portal	32MB
0xF_F400_0000	0xF_F5FF_FFFF	Buffer manager software portal	32MB
0xF_E800_0000	0xF_EFFF_FFFF	IFC - NOR Flash	128MB
0xF_0000_0000	0xF_003F_FFFF	DCSR	4MB
0xC_2000_0000	0xC_2FFF_FFFF	PCI Express 3 Mem Space	256MB
0xC_1000_0000	0xC_1FFF_FFFF	PCI Express 2 Mem Space	256MB
0xC_0000_0000	0xC_0FFF_FFFF	PCI Express 1 Mem Space	256MB
0x0_0000_0000	0x0_7FFF_FFFF	DDR Memory	2GB

4.4.14.7 Flash Bank Usage

The NOR flash (only on Rev-C) on the board can be seen as 8 flash banks. The board DIP switch configuration (for T1023RDB, SW3[5:7]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log (only Rev-C board supports NOR flash, NOR is unavailable on Rev-B):

```
CPU0:  T1023E, Version: 1.0, (0x85490010)
Core:  e5500, Version: 2.1, (0x80241021)
Single Source Clock Configuration
Clock Configuration:
       CPU0:1400 MHz, CPU1:1400 MHz,
       CCB:400  MHz,
       DDR:800  MHz (1600 MT/s data rate) (Asynchronous), IFC:100  MHz
       FMAN1: 700 MHz
       QMAN:  400 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 0810000e 00000000 00000000 00000000
       00000010: 3b800003 00000012 ec02f000 21000000
       00000020: 00000000 00000000 00000000 00022800
       00000030: 00000130 04020200 00000000 00000006
Board: T1023RDB, RevC, boot from NOR vBank0
SERDES Reference Clocks:
SD1_CLK1=100.00MHZ, SD1_CLK2=125.00MHz
I2C:   ready
SPI:   ready
DRAM:  Detected UDIMM Fixed DDR4 on board
2 GiB (DDR4, 32-bit, CL=11, ECC off)
Flash: 128 MiB
L2:    256 KiB enabled
Corenet Platform Cache: 256 KiB enabled
Using SERDES1 Protocol: 119 (0x77)
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=> switch bank0
or set DIP-switch SW3[5:7]=000
```

- Switch to alternate bank:

```
=> switch bank4
or set DIP-switch SW3[5:7]=100
```

The table below shows the T1023RDB NOR flash memory map.

Table 82. NOR Flash Memory Map

Range Start	Range End	Definition	Size

Table continues on the next page...

Table 82. NOR Flash Memory Map (continued)

0xeff40000	0xefffffff	U-Boot (current bank)	768 KB
0xeff20000	0xeff3ffff	U-Boot env (current bank)	128 KB
0xeff00000	0xeff1ffff	FMAN Micro code (current bank)	128 KB
0xed300000	0xefdfffff	Rootfs (alternate bank)	44 MB
0xed000000	0xed2fffff	Guest image #3 (alternate bank)	3 MB
0xecd00000	0xecffffff	Guest image #2 (alternate bank)	3 MB
0xeca00000	0xeccfffff	Guest image #1 (alternate bank)	3 MB
0xec900000	0xec9fffff	HV config device tree(alt bank)	1 MB
0xec800000	0xec8fffff	Hardware device tree(alternate bank)	1 MB
0xec700000	0xec7fffff	HV.uImage (alternate bank)	1 MB
0xec020000	0xec6fffff	Linux.uImage (alternate bank)	~7 MB
0xec000000	0xec01ffff	RCW (alternate bank)	128 KB
0xebf40000	0xebffffff	U-Boot (alternate bank)	768 KB
0xebf20000	0xebf3ffff	U-Boot env (alternate bank)	128 KB
0xebf00000	0xebf1ffff	FMan Micro code (alternate bank)	128 KB
0xe9300000	0xebdfffff	Rootfs (current bank)	44 MB
0xe8900000	0xe92fffff	Guest image #3 (current bank)	3 MB
0xe8d00000	0xe8ffffff	Guest image #2 (current bank)	3 MB

Table continues on the next page...

Table 82. NOR Flash Memory Map (continued)

0xe8a00000	0xe8cfffff	Guest image #1 (current bank)	3 MB
0xe8900000	0xe89fffff	HV config device tree(cur bank)	1 MB
0xe8800000	0xe88fffff	Hardware device tree (current bank)	1 MB
0xe8700000	0xe87fffff	HV.uImage (current bank)	1MB
0xe8020000	0xe86fffff	Linux.uImage (current bank)	~7 MB
0xe8000000	0xe801ffff	RCW (current bank)	128 KB

4.4.14.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>reset
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
```

```
=>protect on <rcw_start_addr> +$filesize
=>reset
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <ucode_start_addr>. <ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <ucode_start_addr> +$filesize
=>erase <ucode_start_addr> +$filesize
=>cp.b 1000000 <ucode_start_addr> $filesize
=>protect on <ucode_start_addr> +$filesize
=>reset
```

4.4.14.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.14.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 1000000 <uImage_name>
   =>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 2000000 <platform_dtb_name>
   =>bootm 1000000 5000000 2000000
   ```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.14.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

```
=>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
=>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
=>saveenv
```

Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.14.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   <nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/nfs restart
   ```

   NOTE

   nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>:
   <your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
   =>setenv netdev <ethx>
   =>saveenv
   ```

   NOTE

   <ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

   ```
   =>tftp 1000000 <uImage name>
   =>tftp c00000 <platform dtb name>
   =>bootm 1000000 - c00000
   ```

   Now the board will boot up with NFS filesystem.

4.4.14.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

   ```
   #fdisk /dev/sdb
   ```

3. Use the mkfs.ext2 command to create the filesystems.

   ```
   # mkfs.vfat /dev/sdb1
   # mkfs.ext2 /dev/sdb2
   ```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

   ```
   # mkdir /temp
   # mount /dev/sdb2 /temp
   # cd /temp
   ```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

   ```
   # mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
   ```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

   ```
   # cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   ```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

   ```
   # cp uImage and uImage.dtb to /temp/boot folder
   # umount /temp
   ```

8. Plug in the SD card to the target board and power on.

4.4.14.9.5 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

```
=>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>erase e9300000 +$filesize
=>cp.b 1000000 e9300000 $filesize
```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

```
=>tftp 1000000 hv.uImage
=>erase e8700000 +$filesize
=>cp.b 1000000 e8700000 $filesize
```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

```
=>tftp 1000000 <platform_name>-usdpaa.dtb
=>erase e8800000 +$filesize
=>cp.b 1000000 e8800000 $filesize
```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.14.10 Check 'Link Up' for Serial Ethernet Interfaces

If you are experiencing problems with your Ethernet interfaces, this section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
1 - RealTek RTL8211FS <--> FM1@DTSEC4
3 - RealTek RTL8211FS <--> FM1@DTSEC1
FM_TGEC_MDIO:
2 - Aquantia AQR105 <--> FM1@DTSEC3
```

The results above show that U-Boot was able to see PHYs on each of the 3 DTSEC interfaces.

Check RGMII Link Status for External PHY

In order to check the status of a RGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read FM1@DTSEC4 1
Reading from bus FSL_MDIO0
PHY at address 2:
1 - 0x796d
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7969.

4.4.14.11 Basic Networking Ping Test

In Linux, in order to check that your network driver is set up, follow the commands below:

```
#ip addr show
#ip addr add <ip address of board>/24 dev <port in Linux>
#ip link set <port in Linux> up
#ping <serverip>
```

If your network driver is not working, check your kernel messages. You should not see any errors reported by the FSL FMan MAC based driver:

```
fsl_mac: fsl_mac: FSL FMan MAC API based driver ()
fsl_mac ffe4e0000.ethernet: FMan MEMAC
fsl_mac ffe4e0000.ethernet: FMan MAC address: 00:04:9f:03:9e:5a
fsl_mac ffe4e4000.ethernet: FMan MEMAC
fsl_mac ffe4e4000.ethernet: FMan MAC address: 00:04:9f:03:9e:5c
fsl_mac ffe4e6000.ethernet: FMan MEMAC
fsl_mac ffe4e6000.ethernet: FMan MAC address: 00:15:17:be:d6:2d
```

4.4.15 T1024RDB

4.4.15.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.15.2 Switch Settings

The T1024RDB has user selectable switches for evaluating different boot options for the T1024 device. The table below lists the default switch settings. For a description of these settings, see T1024RDB User Guide.

Table 83. Default Switch Settings

	1	2	3	4	5	6	7	8
SW1	ON [0]	ON [0]	ON [0]	OFF [1]	ON [0]	ON [0]	OFF [1]	OFF [1]
SW2	OFF	ON	OFF	OFF	OFF	OFF	OFF	OFF
SW3	ON	OFF	OFF	ON	ON	ON	ON	OFF

Note that with the default settings, the following are selected.

SW1[1:8] = 0001 0011:

NOR Boot mode is selected with default RCW 0x95 in bank0.

SW2[7] = 1:

T1024 uses DDR3L (1.35V).

For 2.5G SGMII: Turn off the board, switch from bank0 to bank4 to use RCW 0x135, set SW3[5] = OFF(1) and SW3[2] = OFF(1)

For 10G Base-T: Turn off the board, switch from bank4 to bank0 to use RCW 0x95, and set SW3[5] = ON(0) and SW3[2] = ON(0)

NOTE: Ignore the above switch settings for SW3[5] and SW3[2], when board is turned on while switching to bank0 using cpld reset or switching to bank4 using cpld reset altbank.

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 84. Boot Device Scenarios

Alternate Boot Devices	**Change the following switch settings** **(0 = ON, 1 = OFF)**
NAND boot	SW1 [1:8] = 1000 1000 SW2 [1:8] = 1011 1111 SW3 [1:8] = 0111 0001
SPI boot	SW1 [1:8] = 0010 0010 SW2 [1:8] = 1011 1111 SW3 [1:8] = 0110 0001
SD Flash boot	SW1 [1:8] = 0010 0000 SW2 [1:8] = 0011 1111 SW3 [1:8] = 0110 0001

4.4.15.3 U-Boot Environment Variables

4.4.15.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

4.4.15.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.15.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for T1024 version information (e.g., T1024E version 1.0) and also for the version of FMan microcode currently flashed on the T1024RDB (e.g. microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 1.0(the version info please refer to sdk release image):

```
fsl_fman_ucode_t1024_r1.0_106_4_18.bin (*)
fsl_fman_ucode_t1024_r1.0_107_4_2.bin
fsl_fman_ucode_t1024_r1.0_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.15.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK contains RCW binaries for use on T1024RDB:

- RRX_PPP_95/rcw_95_1400MHz.bin (default RCW in bank0 for 10G XFI)
- RRX_PPP_95/rcw_95_1200MHz.bin
- RNS_NPP_135/rcw_135_1400MHz.bin (alternate RCW in bank4 for 2.5G SGMII)
- RNS_NPP_135/rcw_135_1200MHz.bin

The RCW directories' names for the T1024RDB conform to the following naming convention:

```
abc_def_g
```

Table 85. T1024RDB Naming Convention Legend

Slot	Convention
a	'R' if RGMII1 @DTSEC4 is supported 'N' if not available/not used

Table continues on the next page...

Table 85. T1024RDB Naming Convention Legend (continued)

b	'R' if RGMII2 @DTSEC3 is supported 'N" if not available/not used
c	'X' if FM1@TGEC1, XFI 10G Base-T is supported 'S' if FM1@DTSEC3 2.5G SGMII is supported 'N" if not available/not used
d [What is available in PCIe Slot] e [What is availabel in Mini-PCI connector 1] f [What is availabel in Mini-PCI connector 2]	'N' if not available/not used 'P' if PCIe 'X' if XAUI 'R' if SRIO 'S' if SGMII 'H' if SATA 'A' is AURORA
g	Hex value of serdes protocol value

For example,

```
RRX_PPP_95
```

means:

- RGMII@DTSEC4
- RGMII@DTSEC3
- XFI 10G, FM1@TGEC1
- PCIE [Slot 1]
- PCIE [Mini-PCI connector 1]
- PCIE [Mini-PCI connector 2]
- SERDES Protocol is 0x95

The QorIQ SDK release supports SerDes 0x95 default configuration.

```
• RRX_PPP_95 (1x 10G XFI + 2x 1G RGMII + 3x PCIe)
```

The QorIQ SDK release also supports SerDes 0x135 for 2.5G SGMII configuration.

```
• RNS_NPP_135 (1x 2.5G SGMII + 1x 1G RGMII + 2x PCIe)
```

When using the default configuration, the network interfaces, which include two 1Gbps RGMII ports and one 10G XFI 10G Base-T port labeled as the following in U-Boot:

FM1@DTSEC3

FM1@DTSEC4

FM1@TGEC1

Below is a table that maps the different ports available on T1024RDB and their names in different environments.

Table 86. Port Map

Label on 1U box	Port in U-boot	Port in Linux	FMan Address	Comments
ETH0	FM1@DTSEC4	fm1-mac4	0xffe4e6000	1Gbps RGMII (RTL8211E PHY)
ETH1	FM1@DTSEC3	fm1-mac3	0xffe4e4000	1Gbps RGMII (RTL8211E PHY)
ETH2	FM1@TGEC1	fm1-mac1	0xffe4e0000	10Gbps Base-T (AQR105 PHY, use rcw_0x95)
ETH3	FM1@DTSEC3	fm1-mac3	0xffe4e4000	2.5Gbps SGMII (AQR105 PHY, use rcw_0x135)

4.4.15.6 System Memory Map

System memory map on T1024RDB

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
0xF_FFDF_0000	0xF_FFDF_0FFF	IFC - CPLD	4KB
0xF_FF80_0000	0xF_FF80_FFFF	IFC - NAND Flash	64KB
0xF_FE00_0000	0xF_FEFF_FFFF	CCSRBAR	16MB
0xF_F802_0000	0xF_F802_FFFF	PCI Express 3 I/O Space	64KB
0xF_F801_0000	0xF_F801_FFFF	PCI Express 2 I/O Space	64KB
0xF_F800_0000	0xF_F800_FFFF	PCI Express 1 I/O Space	64KB
0xF_F600_0000	0xF_F7FF_FFFF	Queue manager software portal	32MB
0xF_F400_0000	0xF_F5FF_FFFF	Buffer manager software portal	32MB
0xF_E800_0000	0xF_EFFF_FFFF	IFC - NOR Flash	128MB
0xF_0000_0000	0xF_003F_FFFF	DCSR	4MB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Definition	Size
0xC_2000_0000	0xC_2FFF_FFFF	PCI Express 3 Mem Space	256MB
0xC_1000_0000	0xC_1FFF_FFFF	PCI Express 2 Mem Space	256MB
0xC_0000_0000	0xC_0FFF_FFFF	PCI Express 1 Mem Space	256MB
0x0_0000_0000	0x0_FFFF_FFFF	DDR Memory	4GB

4.4.15.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration (for T1024RDB, SW3[5:7]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
CPU0:  T1024E, Version: 1.0, (0x85480010)
Core:  e5500, Version: 2.1, (0x80241021)
Single Source Clock Configuration
Clock Configuration:
       CPU0:1400 MHz, CPU1:1400 MHz,
       CCB:400  MHz,
       DDR:800  MHz (1600 MT/s data rate) (Asynchronous), IFC:100  MHz
       QE:200  MHz
       FMAN1: 700 MHz
       QMAN:  400 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 0810000e 00000000 00000000 00000000
       00000010: 4a800001 80000012 ec027000 21000000
       00000020: 00000000 00000000 00000000 00030810
       00000030: 00000000 0b005a08 00000000 00000006
Board: T1024RDB, Board rev: 0x03 CPLD ver: 0x05, boot from NOR vBank0
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=> cpld reset
or set DIP-switch SW3[5:7]=000
```

- Switch to alternate bank:

```
=> cpld reset altbank
or set DIP-switch SW3[5:7]=100
```

The table below shows the T1024RDB NOR flash memory map.

Table 87. NOR Flash Memory Map

Range Start	Range End	Definition	Size
0xeff40000	0xefffffff	U-Boot (current bank)	768 KB
0xeff20000	0xeff3ffff	U-Boot env (current bank)	128 KB
0xeff00000	0xeff1ffff	FMAN Micro code (current bank)	128 KB
0xefe00000	0xefe3ffff	QE firmware (current bank)	256KB
0xed300000	0xefefffff	Rootfs (alternate bank)	43 MB
0xec800000	0xec8fffff	Hardware device tree(alternate bank)	1 MB
0xec020000	0xec7fffff	Linux.uImage (alternate bank)	7 MB + 875 KB
0xec000000	0xec01ffff	RCW (alternate bank)	128 KB
0xebf40000	0xebffffff	U-Boot (alternate bank)	768 KB
0xebf20000	0xebf3ffff	U-Boot env (alternate bank)	128 KB
0xebf00000	0xebf1ffff	FMan Micro code (alternate bank)	128 KB
0xebe00000	0xebe3ffff	QE firmware (alt bank)	256KB
0xe9300000	0xebefffff	Rootfs (current bank)	43 MB
0xe8800000	0xe88fffff	Hardware device tree (current bank)	1 MB
0xe8020000	0xe86fffff	Linux.uImage (current bank)	7 MB + 875 KB
0xe8000000	0xe801ffff	RCW (current bank)	128 KB

4.4.15.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is

no need to switch into the alternate bank after each configuration, i.e. type the command *cpld reset altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>cpld reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>cpld reset altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <ucode_start_addr>. <ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <ucode_start_addr> +$filesize
=>erase <ucode_start_addr> +$filesize
=>cp.b 1000000 <ucode_start_addr> $filesize
=>protect on <ucode_start_addr> +$filesize
=>cpld reset altbank
```

4.4.15.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.15.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

```
=>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
=>saveenv
```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

```
=>tftp 1000000 <uImage_name>
=>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>tftp 2000000 <platform_dtb_name>
=>bootm 1000000 5000000 2000000
```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.15.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

```
=>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
=>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
=>saveenv
```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.15.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

   b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters

on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
=>bootm 1000000 - c00000
```

   Now the board will boot up with NFS filesystem.

4.4.15.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

```
#fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to create the filesystems.

```
# mkfs.vfat /dev/sdb1
# mkfs.ext2 /dev/sdb2
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdb2 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

```
# cp uImage and uImage.dtb to /temp/boot folder
# umount /temp
```

8. Plug in the SD card to the target board and power on.

4.4.15.10 Check 'Link Up' for Serial Ethernet Interfaces

If you are experiencing problems with your Ethernet interfaces, this section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
2 - RealTek RTL8211E <--> FM1@DTSEC4
6 - RealTek RTL8211E <--> FM1@DTSEC3
FM_TGEC_MDIO:
1 - Generic 10G PHY <--> FM1@TGEC1
```

The results from the above mdio list command show that U-Boot was able to see PHYs on each of the 2 dTSEC and on the 10GEC interface. “Generic 10G PHY” is the normal display when using AQR105 PHY on T1024RDB.

Check RGMII Link Status for External PHY

In order to check the status of a RGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read FM1@DTSEC4 1
Reading from bus FSL_MDIO0
PHY at address 2:
1 - 0x796d
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7969.

4.4.15.11 Basic Networking Ping Test

In Linux, in order to check that your network driver is set up, follow the commands below:

```
#ip addr show
#ip addr add <ip address of board>/24 dev <port in Linux>
#ip link set <port in Linux> up
#ping <serverip>
```

If your network driver is not working, check your kernel messages. You should not see any errors reported by the FSL FMan MAC based driver:

```
fsl_mac: fsl_mac: FSL FMan MAC API based driver ()
fsl_mac ffe4e0000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e0000.ethernet: FMan MAC address: 00:04:9f:03:11:1e
fsl_mac ffe4e2000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e2000.ethernet: FMan MAC address: 00:04:9f:03:11:1f
fsl_mac ffe4e4000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e4000.ethernet: FMan MAC address: 00:04:9f:03:11:20
fsl_mac ffe4e6000.ethernet: FMan dTSEC version: 0x08240101
fsl_mac ffe4e6000.ethernet: FMan MAC address: 00:04:9f:03:11:21
```

4.4.16 T1040D4RDB/T1042D4RDB

4.4.16.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.16.2 Switch Settings

The RDB has user selectable switches for evaluating different boot options for T104xD4RDB development system. The table below lists the default switch settings for NOR flash boot.

Table 88. Default Switch Settings

	1	2	3	4	5	6	7	8
SW1	ON [0]	ON [0]	ON [0]	OFF [1]	ON [0]	ON [0]	OFF [1]	OFF [1]
SW2	OFF	ON	OFF	OFF	OFF	ON	ON	OFF
SW3	OFF	OFF	OFF	ON	ON	ON	ON	OFF

NOTE

* PCIe slots modes: All the PCIe devices work as Root Complex.

* For reduce SoC core personalities for T1040/T1042 which is T1020/T1022, set switch SW3[8]=ON

* For single source clocking mode, set SW3[1] = ON and use appropriate RCW

* For `cfg_dram_type` to work with DDR4(1.2V), set SW2[7] = 0; to work with DDR3L(1.35V), set SW2[7] = 1.

Table 89. Frequency settings in MHz

SYSCLK	CPU Core (e5500)	CCB	DDR	FMan	DDRCLK	QMAN	PME
100	1400	600	800 (1600MT/s)	600	66.66	300	300

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 90. Boot Device Scenarios

Alternate Boot Devices	Change these DIP switch settings (0=ON/1=OFF)
a) NAND boot	• SW1 [1:8] = 1000 1000 • SW2 [1:8] = 0011 1001 • SW3 [1:8] = 1111 0001
a) SPI boot	• SW1 [1:8] = 0010 0010 • SW2 [1:8] = 1011 1001 • SW3 [1:8] = 1110 0001
b) SD boot	• SW1 [1:8] = 0010 0000 • SW2 [1:8] = 0011 1001 • SW3 [1:8] = 1110 0001

NOTE

* For reduce SoC core personalities for T1040/T1042 which is T1020/T1022, set switch SW3[8]=0

* For single source clocking mode, set SW3[1] = 0 and use appropriate RCW

4.4.16.3 U-Boot Environment Variables

4.4.16.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.

- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The "hwconfig" default setting enables DDR chip select interleaving as well as USB1 and USB2 configuration:

```
hwconfig=fsl_ddr:bank_intlv=cs0_cs1;usb1:dr_mode=host,phy_type=utmi;usb2:dr_mode=host,phy_
type=utmi
```

NOTE

For USB gadget set "dr_mode=peripheral"

4.4.16.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.16.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for T1040 version information (e.g., T1040E version 1.1) and also for the version of FMan microcode currently flashed on the T104XD4RDB (e.g. microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

For silicon revision1.0:

```
fsl_fman_ucode_t1040_r1.0_106_4_15.bin (*)
fsl_fman_ucode_t1040_r1.0_107_4_2.bin
```

For silicon revision 1.1:

```
fsl_fman_ucode_t1040_r1.1_106_4_18.bin (*)
fsl_fman_ucode_t1040_r1.1_107_4_2.bin
fsl_fman_ucode_t1040_r1.1_108_5_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.16.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK 2.0 contains the following RCW binary for use on the T1040D4RDB/T1020D4RDB:

```
t1040d4rdb/ RR_P_66/rcw_1400MHz.rcw (default)
t1040d4rdb/ RR_P_66/rcw_1400MHz_single_source.rcw
t1040d4rdb/ RR_P_66/rcw_sben_1400MHz.rcw
t1040d4rdb/ RR_P_66/rcw_sben_1400MHz_single_source.rcw
```

QorIQ SDK 2.0 contains the following RCW binary for use on the T1042D4RDB/T1022D4RDB:

```
t1042d4rdb/RR_P_86/rcw_1400MHz.rcw (default)
t1042d4rdb/RR_P_86/rcw_1400MHz_single_source.rcw
t1042d4rdb/RR_P_86/rcw_sben_1400MHz.rcw
t1042d4rdb/RR_P_86/rcw_sben_1400MHz_single_source.rcw
```

The RCW directories' names for the T1040D4RDB/T1042D4RDB conform to the following naming convention:

```
ab_c_d
```

Table 91. T1040D4RDB/T1042D4RDB RCW Directories Naming Convention Legend

Label	Convention

Table continues on the next page...

Table 91. T1040D4RDB/T1042D4RDB RCW Directories Naming Convention Legend (continued)

a	'R' if RGMII@MAC4 or RGMII@MAC2 is supported 'N' if not available/not used
b	'R' if RGMII@MAC5 is supported 'N" if not available/not used
c [What is available in Slot 1]	'P' if PCIe 'S' if SGMII 'Q' if QSGMII
d	Hex value of serdes1 protocol value

Note: For SerDes protocol 0x66, RGMII is available on FM@MAC4 and FM@MAC5

For example,

```
RR_P_66
```

means:

- RGMII@mac4
- RGMII@mac5
- One PCIe slot
- SERDES Protocol is 0x66

The RCW filenames for the T1040D4RDB/T1042D4RDB conform to the following naming convention:

```
rcw_x.rcw
rcw_x_clockmode.rcw
```

Table 92. T1040D4RDB/T1042D4RDB RCW Files Naming Convention Legend

Label	Convention
x	Core frequency
clockmode	"single_source" for single source clocking. If "single_source" is not present, it means separate clock source for sysclk and ddrclk

NOTE

* RR_P_66 is the default configuration for T1040D4RDB/T1020D4RDB.

* RR_P_86 is the default configuration for T1042D4RDB/T1022D4RDB

When using the default configuration, the network interfaces, are labeled as the following in U-Boot:

For T1040D4RDB:

FM1@DTSEC3 (SGMII)

FM1@DTSEC4 (RGMII)

FM1@DTSEC5 (RGMII)

For T1042D4RDB:

FM1@DTSEC1 (SGMII)

FM1@DTSEC2 (SGMII)

FM1@DTSEC3 (SGMII)

FM1@DTSEC4 (RGMII)

FM1@DTSEC5 (RGMII)

Below is a table that maps the different ports available on T1040D4RDB and their names in different environments.

Table 93. Port Map for T1040D4RDB

Label on Panel of T1040 RDB	Port in U-Boot	Port in Linux	FMan Address
ETH0	FM1@DTSEC4	fm1-gb3	0xfe4e6000
ETH1	FM1@DTSEC5	fm1-gb4	0xfe4e8000
ETH2	FM1@DTSEC3	fm1-gb2	0xfe4e4000
ETH3	Not supported in U-Boot		0xfe4e0000
ETH4	Not supported in U-Boot		
ETH5	Not supported in U-Boot		
ETH6	Not supported in U-Boot		
ETH7	Not supported in U-Boot		0xfe4e2000
ETH8	Not supported in U-Boot		
ETH9	Not supported in U-Boot		
ETH10	Not supported in U-Boot		

NOTE

ETH3-ETH10 belong to the switch. They are not visible as interfaces in Linux. Switch control software is required to control them.

4.4.16.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
`0xfffdf0000`	`0xfffdfffff`	IFC - CPLD	4 KB
`0xfff800000`	`0xfff80ffff`	IFC - NAND Flash	64 KB
`0xffe000000`	`0xffeffffff`	CCSRBAR	16 MB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Definition	Size
0xff8030000	0xff803ffff	PCI Express 4 I/O Space	64 KB
0xff8020000	0xff802ffff	PCI Express 3 I/O Space	64 KB
0xff8010000	0xff801ffff	PCI Express 2 I/O Space	64 KB
0xff8000000	0xff800ffff	PCI Express 1 I/O Space	64 KB
0xff6000000	0xff7ffffff	Queue manager software portal	32 MB
0xff4000000	0xff5ffffff	Buffer manager software portal	32 MB
0xfe8000000	0xfefffffff	IFC - NOR Flash	128 MB
0xf00000000	0xf003fffff	DCSR	4 MB
0xc30000000	0xc3fffffff	PCI Express 4 Mem Space	265 MB
0xc20000000	0xc2fffffff	PCI Express 3 Mem Space	265 MB
0xc10000000	0xc1fffffff	PCI Express 2 Mem Space	265 MB
0xc00000000	0xc0fffffff	PCI Express 1 Mem Space	265 MB
0x000000000	0x0fffffffff	DDR	2 GB

4.4.16.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration (for T1040RDB, SW3[5:7]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

For T1040D4RDB:

```
CPU0:  T1040E, Version: 1.1, (0x85280011)
Core:  e5500, Version: 2.1, (0x80241021)
```

```
Clock Configuration:
       CPU0:1400 MHz, CPU1:1400 MHz, CPU2:1400 MHz, CPU3:1400 MHz,
       CCB:600  MHz,
       DDR:800  MHz (1600 MT/s data rate) (Asynchronous), IFC:150  MHz
       QE:300  MHz
       FMAN1: 600 MHz
       QMAN:  300 MHz
       PME:   300 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 0c18000e 0e000000 00000000 00000000
       00000010: 66000002 40000002 ec027000 01000000
       00000020: 00000000 00000000 00000000 00030810
       00000030: 00000000 0342580f 00000000 00000000
Board: T1040D4RDB
Board rev: 0x01 CPLD ver: 0x03, vBank: 4
```

For T1042D4RDB:

```
CPU0:  T1042E, Version: 1.1, (0x85280211)
Core:  e5500, Version: 2.1, (0x80241021)
Clock Configuration:
       CPU0:1400 MHz, CPU1:1400 MHz, CPU2:1400 MHz, CPU3:1400 MHz,
       CCB:600  MHz,
       DDR:800  MHz (1600 MT/s data rate) (Asynchronous), IFC:150  MHz
       QE:300  MHz
       FMAN1: 600 MHz
       QMAN:  300 MHz
       PME:   300 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 0c18000e 0e000000 00000000 00000000
       00000010: 86000002 40000002 ec027000 01000000
       00000020: 00000000 00000000 00000000 00030810
       00000030: 00000000 01fe580f 00000000 00000000
Board: T1042D4RDB
Board rev: 0x01 CPLD ver: 0x02, vBank: 4
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>cpld reset
```

- Switch to alternate bank:

```
=>cpld reset altbank
```

The table below shows the T1040D4RDB/T1042D4RDB NOR flash memory map.

Table 94. NOR flash memory map

Range Start	Range End	Definition	Size
0xEFF40000	0xEFFFFFFF	U-Boot (current bank)	768 KB
0xEFF20000	0xEFF3FFFF	U-Boot env (current bank)	128 KB
0xEFF00000	0xEFF0FFFF	FMAN Ucode (current bank)	64 KB
0xEFF10000	0xEFF1FFFF	QE Ucode (current bank)	64 KB
0xED300000	0xEFEFFFFF	rootfs (alt bank)	43MB
0xEC800000	0xEC8FFFFF	Hardware device tree (alternate bank)	1 MB
0xEC700000	0xEC7FFFFF	HV.uImage (alternate bank)	1 MB
0xEE020000	0XEC7FFFFF	Linux.uImage (alt bank)	7MB+875KB
0xEC000000	0xEC01FFFF	RCW (alternate bank)	128 KB
0xEBF40000	0xEBFFFFFF	U-Boot (alternate bank)	768 KB
0xEBF20000	0xEBF3FFFF	U-Boot env (alternate bank)	128 KB
0xEBF00000	0xEBF1FFFF	FMAN ucode (alternate bank)	64 KB
0xEBF10000	0XEBF1FFFF	QE Ucode (alt bank)	64 KB
0xE9300000	0xEBEFFFFF	rootfs (current bank)	43MB
0XE8800000	0XE88FFFFF	Hardware device tree (current bank)	1 MB
0xE8020000	0xE86FFFFF	Linux.uImage (current bank)	7MB+875KB
0xE8000000	0xE801FFFF	RCW (current bank)	128 KB

4.4.16.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *cpld reset altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>cpld reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>cpld reset altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>. <fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>cpld reset altbank
```

4.4.16.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.16.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

```
=>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
=>saveenv
```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

```
=>tftp 1000000 <uImage_name>
=>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>tftp 2000000 <platform_dtb_name>
=>bootm 1000000 5000000 2000000
```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.16.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

```
=>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
=>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
=>saveenv
```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
```

```
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.16.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

   b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
=>bootm 1000000 - c00000
```

   Now the board will boot up with NFS filesystem.

4.4.16.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

```
#fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to create the filesystems.

```
# mkfs.vfat /dev/sdb1
# mkfs.ext2 /dev/sdb2
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdb2 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

```
# cp uImage and uImage.dtb to /temp/boot folder
# umount /temp
```

8. Plug in the SD card to the target board and power on.

4.4.16.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

```
# fdisk /dev/sdx
```

3. Use the mkfs command to create the filesystems.

```
# mkfs.ext2 /dev/sdx1
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdx1 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure copy the target kernel image and dtb to sata.

```
# cp uImage and target.dtb to /temp/boot folder
# umount /temp
```

8. Connect the SATA hard drive to the target board and power on.

4.4.16.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

   TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

   ```
   =>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>erase e9300000 +$filesize
   =>cp.b 1000000 e9300000 $filesize
   ```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

   ```
   =>tftp 1000000 hv.uImage
   =>erase e8700000 +$filesize
   =>cp.b 1000000 e8700000 $filesize
   ```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

   ```
   =>tftp 1000000 <platform_name>-usdpaa.dtb
   =>erase e8800000 +$filesize
   =>cp.b 1000000 e8800000 $filesize
   ```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.16.10 Check 'Link Up' for Serial Ethernet Interfaces

If you are experiencing problems with your Ethernet interfaces, this section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
0 - RealTek RTL8211E <--> FM1@DTSEC3
1 - RealTek RTL8211E <--> FM1@DTSEC4
2 - RealTek RTL8211E <--> FM1@DTSEC5
```

The results from the above *mdio list* command show that U-Boot was able to see PHYs on each of the 3 dTSEC interfaces. If you see “Generic” reported, it is an indication that something is there but the T1040 can’t communicate with the device/port.

Check SGMII Link Status for External PHY

In order to check the status of a SGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read FM1@DTSEC3 1
Reading from bus FSL_MDIO0
PHY at address 0:
1 - 0x796d
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7969.

Check SGMII link Status for Internal PHY

The SGMII PCS on-chip also has link information. In this section we'll check for a valid link for FM1 EMAC3 (listed as FM1@DTSEC3 in U-Boot). For this check, if you want to see a valid link reported, connect the interface to a link partner. For FM1 EMAC3 (FM1@DTSEC3), the register we want to read is SGMII Status Register (MDIO_SGMII_SR). In particular, we will check MDIO_SGMII_SR[LINK_STAT]. The register MDIO_SGMII_SR is in the SerDes SGMII MDIO register space at

offset 0x1. Note that for SGMII PCS we are using a Clause 22 device. We will use the following registers in the in the Ethernet MAC controller register space to perform the read:

Offset 0x34 Management interface control register (MDIO_CTL)

Offset 0x38 Management interface data register (MDIO_DATA)

We will program MDIO_CTL so that a read of MDIO_DATA will return the contents of MDIO_SGMII_SR. The first step is to identify the values needed to write to MDIO_CTL:

Since 0x1 is the Offset of MDIO_SGMII_SR is the SGMII MDIO register space set MDIO_CTL[DEV_ADDR] = 0x1.

The SGMII MDIO register space is selected when the associated SGMIInCR1[MDEV_PORT] matches the Ethernet MAC PHY address (MDIO_CTL[PHY_ADDR]). So we will set:

```
MDIO_CTL[DEV_ADDR] = 0x1
MDIO_CTL[PHY_ADDR] = SGMIInCR1[MDEV_PORT]
```

NOTE

SGMII Protocol Control Register 1 (SGMIInCR1, n = A, B, ..., F) is in the SerDes register space.

In T1040RM see table 31.1.1.5 "MDIO port mapping for FM" for the mapping of PCD MDIO Slave to MAC MDIO Master. For this example, we're using MAC MDIO master FM1- MAC3 and PCS MDIO slave SGMIIC (i.e., SGMIInCR1 where n=C). Read SGMIICCR1:

```
=> md fe0ea624
fe0ea624: 000008bf 00000000 00000000 80000000 ................
```

So SGMIIACR1[MDEV_PORT] = 0.

So we will program MDIO_CTL with the following values:

```
MDIO_CTL[DEV_ADDR] = 0x1
MDIO_CTL[PHY_ADDR] = SGMIICCR1[MDEV_PORT] = 0
```

Read of FM1-EMAC3 MDIO_SGMII_SR:

```
=> mm fe4e5034 <== FM1 (offset 0x400000) MDIO-3 for EMAC3 (offset 0xe5000) MDIO_CTL
(offset 0x34)
fe4e5034: 00000003 ? 8001 <== Set MDIO_CTL[READ] = 1 MDIO_CTL[PHY_ADDR] = 0 and
MDIO_CTL[DEV_ADDR] = 1
fe4e1038: 00001001 ? x
=> md fe4e5030 <== Read MDIO_DATA (offset 0x38) for contents of
MDIO_SGMII_SR
fe4e5030: 40001408 00000001 0000002d 0000002d @...............
```

The above shows that MDIO_SGMII_SR = 0x0000002d and so MDIO_SGMII_SR[LINK_STAT] = 1 indicating that the link is valid.

4.4.16.11 Basic Networking Ping Test

In Linux, in order to check that your network driver is set up, follow the commands below:

```
#ip addr show
#ip addr add <ip address of board>/24 brd + dev <port in Linux>
#ip link set <port in Linux> up
#ping <serverip>
```

If your network driver is not working, check your kernel messages. You should not see any errors reported by the FSL FMan MAC API based driver:

```
fsl_mac: fsl_mac: FSL FMan MAC API based driver ()
fsl_mac ffe4e0000.ethernet: FMan MEMAC
fsl_mac ffe4e0000.ethernet: FMan MAC address: 00:e0:0c:00:81:00
fsl_mac ffe4e2000.ethernet: FMan MEMAC
fsl_mac ffe4e2000.ethernet: FMan MAC address: 00:e0:0c:00:81:01
fsl_mac ffe4e4000.ethernet: FMan MEMAC
fsl_mac ffe4e4000.ethernet: FMan MAC address: 00:e0:0c:00:81:02
fsl_mac ffe4e6000.ethernet: FMan MEMAC
fsl_mac ffe4e6000.ethernet: FMan MAC address: 00:e0:0c:00:81:03
fsl_mac ffe4e8000.ethernet: FMan MEMAC
fsl_mac ffe4e8000.ethernet: FMan MAC address: 00:e0:0c:00:81:04
```

4.4.17 T2080QDS

4.4.17.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.17.2 Switch Settings

The QDS has user selectable switches for evaluating different boot options for the T2080QDS device. The table below lists the default switch settings. For a description of these settings, see T2080QDS Reference Manual.

Table 95. T2080QDS Default Switch Settings

Switch	1	2	3	4	5	6	7	8
SW1	OFF	OFF	OFF	ON	OFF	OFF	ON	ON
SW2	ON	ON	ON	ON	ON	ON	ON	OFF
SW3	OFF	OFF	ON	OFF	ON	ON	ON	OFF
SW4	ON	OFF	ON	ON	ON	ON	ON	ON
SW5	ON	OFF	OFF	OFF	ON	ON	ON	ON
SW6	OFF	OFF	OFF	OFF	ON	ON	ON	ON
SW7	OFF	ON	ON	OFF	OFF	ON	ON	OFF
SW8	OFF	OFF	OFF	ON	OFF	ON	ON	OFF
SW9	ON	OFF	ON	OFF	OFF	OFF	ON	OFF
SW10	ON	ON	ON	OFF	ON	OFF	OFF	OFF
SW11	ON	ON	ON	ON	ON	ON	OFF	OFF
SW12	ON	ON	ON	ON	ON	ON	ON	ON

4.4.17.3 U-Boot Environment Variables

4.4.17.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=auto;usb1:dr_mode=host,phy_type=utmi
```

VID support for T2080QDS

The fuse status register provides the values from on-chip voltage ID efuses programmed at the factory. These values define the voltage requirements for the chip, specifically VDD (core voltage). U-Boot reads FUSESR and translates the values into the appropriate commands to set the voltage output value of an external voltage regulator. B4860QDS has a a IR36021 programmable digital Power Controller, which is programmed as per the value read from the fuses.

There is an environment variable also provided to override the fuse specified voltage.

```
"t208xqds_vdd_mv"
```

For example:

```
=> setenv t208xqds_vdd_mv 1050
=> save
=> reset
```

The commands above will override the Voltage with 1.05V.

NOTE

you also can use u-boot command "vdd_override" to set the VDD. However, The core voltage should be the VID value or Recommended value.

4.4.17.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
```

```
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.17.4 Frame Manager (FMan) Microcode

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for T2080 version information (e.g., T2080E version 1.1) and also for the version of FMan microcode currently flashed on the T2080QDS (e.g. microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 1.1:

```
fsl_fman_ucode_t2080_r1.1_106_4_18.bin (*)
fsl_fman_ucode_t2080_r1.1_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.17.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK 2.0 contains RCW binaries for use on the T2080QDS:

- RR_PNNPPH_66_15/rcw_66_15_1800MHz.bin (default RCW, Core/Platform/DDR = 1800MHz/600MHz/1866MT/s)

- RR_PNSNNR_6C_2D/rcw_6c_2d_1800MHz.bin

The RCW directories' names for the T2080QDS conform to the following naming convention:

```
ab_cdefgh_i_j
```

Table 96. T2080QDS Naming Convention Legend

Slot	Convention
a	'R' if RGMII1@DTSEC3 is supported 'N' if not available/not used
b	'R' if RGMII2@DTSEC4 is supported 'N" if not available/not used
c [What is available in Slot 1] d [What is availabel in Slot 2] e [What is available in Slot 3] f [What is available in Slot 4] g [What is available in Slot 5] h [What is available between SATA or SRIO]	'N' if not available/not used 'P' if PCIe 'X' if XAUI 'R' if SRIO 'S' if SGMII 'F' if XFI 'H' if SATA 'A' is AURORA
i	'hex value of serdes1 protocol value'
j	'hex value of serdes2 protocol value'

For exmaple,

```
RR_PNNPPH_66_15
```

means:

- RGMII1@DTSEC3
- RGMII2@DTSEC4
- PCIE [Slot 1]
- Slot 2 and 3 are not used
- PCIE [Slot 4]
- PCIE [Slot 5]
- SATA
- serdes1 protocol is 0x66
- serdes2 protocol is 0x15

When using the default configuration, the network interfaces, which include RGMII1, RGMII2 and 4 XFI interfaces, are labeled as the following in U-Boot:

FM1@DTSEC3(MAC3)

FM1@DTSEC4(MAC4)

FM1@TGEC1(MAC9)

FM1@TGEC2(MAC10)

FM1@TGEC3(MAC1)

FM1@TGEC4(MAC2)

4.4.17.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Discription	Size
0xF_FFDF_0000	0xF_FFDF_0FFF	IFC - QIXIS FPGA	4 KB
0xF_FF80_0000	0xF_FF8F_FFFF	IFC - NAND Flash	1 MB
0xF_FE00_0000	0xF_FEFF_FFFF	CCSR BAR	16 MB
0xF_F803_0000	0xF_F803_FFFF	PCI Express 4 I/O Space	64 KB
0xF_F802_0000	0xF_F802_FFFF	PCI Express 3 I/O Space	64 KB
0xF_F801_0000	0xF_F801_FFFF	PCI Express 2 I/O Space	64 KB
0xF_F800_0000	0xF_F800_FFFF	PCI Express 1 I/O Space	64 KB
0xF_F600_0000	0xF_F7FF_FFFF	Qman	32 MB
0xF_F400_0000	0xF_F5FF_FFFF	Bman	32 MB
0xF_E800_0000	0xF_EFFF_FFFF	IFC - NOR Flash	128 MB
0xF_0000_0000	0xF_003F_FFFF	DCSR	4 MB
0xC_4000_0000	0xC_4FFF_FFFF	PCI Express 4 Mem Space	256 MB
0xC_3000_0000	0xC_3FFF_FFFF	PCI Express 3 Mem Space or Serial Rapid I/O 2	256 MB
0xC_2000_0000	0xC_2FFF_FFFF	PCI Express 2 Mem Space or Serial Rapid I/O 1	256 MB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Discription	Size
0xC_0000_0000	0xC_1FFF_FFFF	PCI Express 1 Mem Space	512 MB
0x0_0000_0000	0x0_FFFF_FFFF	Memory controller	4 GB

4.4.17.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration (for T2080QDS, SW6[1:4]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, please refer to the U-Boot log.

```
CPU0:  T2080E, Version: 1.1, (0x85380011)
Core:  e6500, Version: 2.0, (0x80400120)
Clock Configuration:
       CPU0:1800 MHz, CPU1:1800 MHz, CPU2:1800 MHz, CPU3:1800 MHz,
       CCB:600  MHz,
       DDR:933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:150  MHz
       FMAN1: 700 MHz
       QMAN:  300 MHz
       PME:   600 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 0c070012 0e000000 00000000 00000000
       00000010: 6c2d0002 00000000 ec027000 c1000000
       00000020: 00000000 00000000 00000000 000307fc
       00000030: 00000000 00000000 00000000 00000004
Board: T2080QDS, Sys ID: 0x28, Board Arch: V1, Board Version: A, boot from vBank4
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>qixis
```

- Switch to alternate bank:

```
=>qixis altbank
```

The table below shows the T2080QDS NOR flash memory map.

Table 97. NOR flash memory map

Range Start	Range End	Definition	Size

Table continues on the next page...

Table 97. NOR flash memory map (continued)

0xEFF40000	0xEFFFFFFF	U-Boot (current bank)	768KB
0xEFF20000	0xEFF3FFFF	U-Boot env (current bank)	128KB
0xEFF00000	0xEFF1FFFF	FMAN Ucode (current bank)	128KB
0xED300000	0xEFEFFFFF	Rootfs (alternate bank)	44MB
0xED000000	0xED2FFFFF	Guest image #3 (alternate bank)	3MB
0xECD00000	0xECFFFFFF	Guest image #2 (alternate bank)	3MB
0xECA00000	0xECCFFFFF	Guest image #1 (alternate bank)	3MB
0xEC900000	0xEC9FFFFF	HV config device tree (alternate bank)	1MB
0xEC800000	0xED2FFFFF	Hardware device tree (alternate bank)	11MB
0xEC700000	0xEC7FFFFF	HV.uImage (alternate bank)	1MB
0xEC020000	0xEC7FFFFF	Linux.uImage (alternate bank)	~7MB
0xEC000000	0xEC01FFFF	RCW (alternate bank)	128KB
0xEBF40000	0xEBFFFFFF	U-Boot (alternate bank)	768KB
0xEBF20000	0xEBF3FFFF	U-Boot env (alternate bank)	128KB
0xEBF00000	0xEBF1FFFF	FMAN ucode (alternate bank)	128KB
0xE9300000	0xEBEFFFFF	Rootfs (current bank)	44MB
0xE9000000	0xE92FFFFF	Guest image #3(current bank)	3MB
0xE8D00000	0xE8FFFFFF	Guest image #2 (current bank)	3MB

Table continues on the next page...

Table 97. NOR flash memory map (continued)

0xE8A00000	0xE8CFFFFF	Guest image #1 (current bank)	3MB
0xE8900000	0xE89FFFFF	HV config device tree (current bank)	1MB
0xE8800000	0xE92FFFFF	Hardware device tree (current bank)	11MB
0xE8700000	0xE87FFFFF	HV.uImage (current bank)	1MB
0xE8020000	0xE87FFFFF	Linux.uImage (current bank)	~7MB
0xE8000000	0xE801FFFF	RCW (current bank)	128KB

4.4.17.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *qixis* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *qixis*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>qixis altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *qixis*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
```

```
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>qixis altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *qixis*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>. <fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>qixis altbank
```

4.4.17.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.17.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 1000000 <uImage_name>
   =>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 2000000 <platform_dtb_name>
   =>bootm 1000000 5000000 2000000
   ```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.17.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   =>saveenv
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

   ```
   =>tftp 1000000 <uImage name>
   =>protect off <kernel_start_addr> +$filesize
   =>erase <kernel_start_addr> +$filesize
   =>cp.b 1000000 <kernel_start_addr> $filesize
   =>protect on <kernel_start_addr> +$filesize
   ```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

   ```
   =>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>protect off <ramdisk_start_addr> +$filesize
   =>erase <ramdisk_start_addr> +$filesize
   =>cp.b 2000000 <ramdisk_start_addr> $filesize
   =>protect on <ramdisk_start_addr> +$filesize
   ```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

   ```
   =>tftp c00000 <platform dtb name>
   =>protect off <dtb_start_addr> +$filesize
   =>erase <dtb_start_addr> +$filesize
   =>cp.b c00000 <dtb_start_addr> $filesize
   =>protect on <dtb_start_addr> +$filesize
   ```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

   ```
   =>boot
   ```

   or

   ```
   => bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
   ```

4.4.17.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

   ```
   <nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
   ```

   b. Restart the NFS service:

   ```
   /etc/init.d/nfs restart
   ```

   NOTE

   nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

   ```
   =>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
   ip=<board_ipaddr>:<tftp_serverip>:
   <your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
   =>setenv netdev <ethx>
   =>saveenv
   ```

   NOTE

   <ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

   ```
   =>tftp 1000000 <uImage name>
   =>tftp c00000 <platform dtb name>
   =>bootm 1000000 - c00000
   ```

   Now the board will boot up with NFS filesystem.

4.4.17.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

   ```
   #fdisk /dev/sdb
   ```

3. Use the mkfs.ext2 command to create the filesystems.

   ```
   # mkfs.vfat /dev/sdb1
   # mkfs.ext2 /dev/sdb2
   ```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

   ```
   # mkdir /temp
   # mount /dev/sdb2 /temp
   # cd /temp
   ```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

   ```
   # mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
   ```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

   ```
   # cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   ```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

   ```
   # cp uImage and uImage.dtb to /temp/boot folder
   # umount /temp
   ```

8. Plug in the SD card to the target board and power on.

4.4.17.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

   ```
   # fdisk /dev/sdx
   ```

3. Use the mkfs command to create the filesystems.

   ```
   # mkfs.ext2 /dev/sdx1
   ```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

   ```
   # mkdir /temp
   # mount /dev/sdx1 /temp
   # cd /temp
   ```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

   ```
   # mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
   ```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

   ```
   # cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
   # sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   # rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
   ```

7. Make sure copy the target kernel image and dtb to sata.

   ```
   # cp uImage and target.dtb to /temp/boot folder
   # umount /temp
   ```

8. Connect the SATA hard drive to the target board and power on.

4.4.17.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

```
=>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>erase e9300000 +$filesize
=>cp.b 1000000 e9300000 $filesize
```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

```
=>tftp 1000000 hv.uImage
=>erase e8700000 +$filesize
=>cp.b 1000000 e8700000 $filesize
```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

```
=>tftp 1000000 <platform_name>-usdpaa.dtb
=>erase e8800000 +$filesize
=>cp.b 1000000 e8800000 $filesize
```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.18 T2080RDB

4.4.18.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.18.2 Switch Settings

The RDB has user selectable switches for evaluating different boot options for the T2080 device. The table below lists the default switch settings. For a description of these settings, see T2080RDB User Guide.

Table 98. Default Switch Settings

	1	2	3	4	5	6	7	8

Table continues on the next page...

Table 98. Default Switch Settings (continued)

SW1	ON [0]	ON [0]	ON [0]	OFF [1]	ON [0]	ON [0]	OFF [1]	OFF [1]
SW2	OFF	ON	OFF	OFF	OFF	OFF	OFF	OFF
SW3	OFF	OFF	OFF	ON	ON	ON	ON	OFF

Note that with the default settings, the following are selected.

(SW1[1:8] SW2[1]) = 000100111:

NOR Boot mode is selected

SW3[1] = 1:

SYSCLK clock source

SW3[5:7] = 000:

NOR flash bank 0 select.

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 99. Boot Device Scenarios

Alternate Boot Devices	**Change the following switch settings (0=ON / 1=OFF)**
NAND boot	SW1 [1:8] = 1000 0010 SW2 [1:8] = 1011 1111 SW3 [1:8] = 1111 0001
SPI boot	SW1 [1:8] = 0010 0010 SW2 [1:8] = 1011 1111 SW3 [1:8] = 1110 0001
SD Flash boot	SW1 [1:8] = 0010 0000 SW2 [1:8] = 0011 1111 SW3 [1:8] = 1110 0001

4.4.18.3 U-Boot Environment Variables

4.4.18.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=auto;usb1:dr_mode=host,phy_type=utmi
```

If you plan to use optical 10G interfaces, you must add information to "hwconfig". This is important. The optical interface will operate erratically without it. It is easiest to do this using the U-Boot "*editenv*" command. Whatever you type will be appended to "hwconfig". The text you need to append is:

```
;fsl_fm1_xaui_phy:xfi
```

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;fsl_fm1_xaui_phy:xfi
```

4.4.18.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.18.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for T2080 version information (e.g., T2080E version 1.1) and also for the version of FMan microcode currently flashed on the T2080RDB (e.g. microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

For silicon revision 1.1:

```
fsl_fman_ucode_t2080_r1.1_106_4_18.bin (*)
fsl_fman_ucode_t2080_r1.1_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.18.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK 2.0 contains RCW binaries for use on the T2080RDB:

- RRFFXX_P_66_15/rcw_66_15_1800MHz.bin (default RCW, Core/Platform/DDR = 1800MHz/600MHz/1866MT/s)
- RRFFXX_P_66_1F/rcw_66_1f_1800MHz.bin
- RRFFXX_P_66_27/rcw_66_27_1800MHz.bin

The RCW directories' names for the T2080RDB conform to the following naming convention:

```
abcdef_g_h_i
```

Table 100. T2080RDB Naming Convention Legend

Label	Convention
a	'R' indicates RGMII1 @ DTSEC3 is supported
b	'R' indicates RGMII2 @ DTSEC4 is supported
c,d	'F' indicates 10G SFP+ fiber ports are supported
e,f	'X' indicates 10G Base-T copper ports are supported

Table continues on the next page...

Table 100. T2080RDB Naming Convention Legend (continued)

g	'P' indicates PCIe4 x4 in the PCI slot
h	SerDes1 protocol value (in hexadecimal)
i	SerDes2 protocol value (in hexadecimal).

For example,

```
RRFFXX_P_66_1F
```

means:

- RGMII1 @ DTSEC3
- RGMII2 @ DTSEC4
- 10G SFP+ fiber ports
- 10G Base-T copper ports
- PCIe4 x4
- SerDes1 protocol 0x66
- SerDes2 protocol 0x1F

Below is a table that maps the different ports available on T2080RDB and their names in different environments.

Table 101. Port Map

Label on 1U box	Port in U-boot	Port in Linux	FMAN Address	Comments
ETH0	FM1@GTEC1	fm1-mac9	0xfe4f0000	10G Base-T SFP+ (Cortina4315)
ETH1	FM1@GTEC2	fm1-mac10	0xfe4f2000	10G Base-T SFP+ (Cortina4315)
ETH2	FM1@GTEC3	fm1-mac1	0xfe4e0000	10G Base-T (AQ1202)
ETH3	FM1@GTEC4	fm1-mac2	0xfe4e2000	10G Base-T (AQ1202)
ETH4	FM1@DTSEC3	fm1-mac3	0xfe4e4000	1G RGMII (RTL8211E)
ETH5	FM1@DTSEC4	fm1-mac4	0xfe4e6000	1G RGMII (RTL8211E)

4.4.18.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
`0xfffdf0000`	`0xfffdf0fff`	IFC - CPLD	4 KB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Definition	Size
0xfff800000	0xfff80ffff	IFC - NAND Flash	64 KB
0xffe000000	0xffeffffff	CCSR	16 MB
0xff8030000	0xff803ffff	PCI Express 4 I/O Space	64 KB
0xff8020000	0xff802ffff	PCI Express 3 I/O Space	64 KB
0xff8010000	0xff801ffff	PCI Express 2 I/O Space	64 KB
0xff8000000	0xff800ffff	PCI Express 1 I/O Space	64 KB
0xff6000000	0xff7ffffff	QMan	32 MB
0xff4000000	0xff5ffffff	BMan	32 MB
0xfe8000000	0xfeffffffff	IFC - NOR Flash	128 MB
0xf00000000	0xf003fffff	DCSR	4 MB
0xc40000000	0xc4fffffff	PCI Express 4 Mem Space	256 MB
0xc30000000	0xc3fffffff	PCI Express 3 Mem Space Rapid I/O 2	256 MB
0xc20000000	0xc2fffffff	PCI Express 2 Mem Space Rapid I/O 1	256 MB
0xc00000000	0xc1fffffff	PCI Express 1 Mem Space	512 MB
0x000000000	0x0fffffff	DDR	4 GB

4.4.18.7 Flash Bank Usage

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration (for T2080RDB, SW3[5:7]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software (or via the dip switch) and effectively swaps bank 0 with the alternate bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
CPU0:  T2080E, Version: 1.1, (0x85380011)
Core:  e6500, Version: 2.0, (0x80400120)
Clock Configuration:
       CPU0:1799.820 MHz, CPU1:1799.820 MHz, CPU2:1799.820 MHz, CPU3:1799.820 MHz,
       CCB:599.940 MHz,
       DDR:933.310 MHz (1866.620 MT/s data rate) (Asynchronous), IFC:149.985 MHz
       FMAN1: 699.930 MHz
       QMAN:  299.970 MHz
       PME:   599.940 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 1207001b 15000000 00000000 00000000
       00000010: 66150002 00000000 ec027000 c1000000
       00000020: 00800000 00000000 00000000 000307fc
       00000030: 00000000 00000000 00000000 00000004
Board: T2080RDB, Board rev: 0x01 CPLD ver: 0x03, boot from NOR vBank0
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>cpld reset
```

- Switch to alternate bank:

```
=>cpld reset altbank
```

The table below shows the T2080RDB NOR flash memory map.

Table 102. NOR Flash Memory Map

Range Start	Range End	Definition	Size
0xeff40000	0xefffffff	U-Boot (current bank)	768 KB
0xeff20000	0xeff3ffff	U-Boot env (current bank)	128 KB
0xeff00000	0xeff1ffff	FMAN Ucode (current bank)	128 KB
0xefe00000	0xefe3ffff	PHY CS4315 firmware	256 KB
0xed300000	0xefefffff	Rootfs (alternate bank)	44 MB
0xED000000	0xED2FFFFF	Guest image #3(alternate bank)	3 MB

Table continues on the next page...

Table 102. NOR Flash Memory Map (continued)

0xECA00000	0xECCFFFFF	Guest image #2 (alternate bank)	3 MB
0xECA00000	0xECCFFFFF	Guest image #1 (alternate bank)	3 MB
0xEC900000	0xEC9FFFFF	HV config device tree (alternate bank)	1 MB
0xec800000	0xec8fffff	Hardware device tree (alternate bank)	1 MB
0xEC700000	0xEC7FFFFF	HV.uImage (alternate bank)	1MB
0xec020000	0xec7fffff	Linux.uImage (alternate bank)	~7 MB
0xec000000	0xec01ffff	RCW (alternate bank)	128 KB
0xebf40000	0xebffffff	U-Boot (alternate bank)	768 KB
0xebf20000	0xebf3ffff	U-Boot env (alternate bank)	128 KB
0xebf00000	0xebf1ffff	FMAN Ucode (alternate bank)	128 KB
0xebe00000	0xebe3ffff	PHY CS4315 firmware (alternate bank)	256 KB
0xe9300000	0xebefffff	Rootfs (current bank)	44 MB
0xE9000000	0xE92FFFFF	Guest image #3 (current bank)	3 MB
0xE8D00000	0xE8FFFFFF	Guest image #2 (current bank)	3 MB
0xE8A00000	0xE8CFFFFF	Guest image #1 (current bank)	3 MB
0xE8900000	0xE89FFFFF	HV config device tree (current bank)	1 MB
0xe8800000	0xe88fffff	Hardware device tree (current bank)	1 MB
0xE8700000	0xE87FFFFF	HV.uImage (current bank)	1 MB

Table continues on the next page...

Table 102. NOR Flash Memory Map (continued)

0xe8020000	0xe87fffff	Linux.uImage (current bank)	~7 MB
0xe8000000	0xe801ffff	RCW (current bank)	128 KB

4.4.18.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *cpld reset altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>cpld reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>cpld reset altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>.

<fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>cpld reset altbank
```

4.4.18.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.18.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 1000000 <uImage_name>
   =>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 2000000 <platform_dtb_name>
   =>bootm 1000000 5000000 2000000
   ```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.18.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   ```

```
=>saveenv
```

Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.18.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
=>bootm 1000000 - c00000
```

Now the board will boot up with NFS filesystem.

4.4.18.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

```
#fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to create the filesystems.

```
# mkfs.vfat /dev/sdb1
# mkfs.ext2 /dev/sdb2
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdb2 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

```
# cp uImage and uImage.dtb to /temp/boot folder
# umount /temp
```

8. Plug in the SD card to the target board and power on.

4.4.18.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

```
# fdisk /dev/sdx
```

3. Use the mkfs command to create the filesystems.

```
# mkfs.ext2 /dev/sdx1
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdx1 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure copy the target kernel image and dtb to sata.

```
# cp uImage and target.dtb to /temp/boot folder
# umount /temp
```

8. Connect the SATA hard drive to the target board and power on.

4.4.18.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

   TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

   ```
   =>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>erase e9300000 +$filesize
   =>cp.b 1000000 e9300000 $filesize
   ```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

   ```
   =>tftp 1000000 hv.uImage
   =>erase e8700000 +$filesize
   =>cp.b 1000000 e8700000 $filesize
   ```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

Target Deployment - for hv-2p mode deployment:

```
=>tftp 1000000 <platform_name>-usdpaa.dtb
=>erase e8800000 +$filesize
=>cp.b 1000000 e8800000 $filesize
```

Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.19 T4240QDS

4.4.19.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.19.2 Switch Settings

The Development System has user selectable switches for evaluating different boot options for the T4240 device. The table below lists the default switch settings. For a description of these settings, see T4240QDS User guide.

Table 103. T4240QDS Default Switch Settings

Switch	1	2	3	4	5	6	7	8
SW1	OFF	OFF	OFF	ON	OFF	ON	ON	ON
SW2	ON	ON	ON	ON	ON	ON	ON	OFF
SW3	OFF	OFF	OFF	OFF	ON	ON	OFF	OFF
SW4	OFF	ON	OFF	ON	OFF	OFF	OFF	OFF
SW5	ON	ON	ON	OFF	OFF	OFF	ON	OFF
SW6	OFF	OFF	OFF	OFF	ON	ON	ON	ON
SW7	ON	ON	ON	ON	ON	OFF	ON	OFF
SW8	ON	ON	OFF	OFF	ON	ON	OFF	ON
SW9	OFF	OFF	OFF	ON	ON	ON	ON	ON

NOTE

- PCIe slots modes: All the PCIe devices work as Root Complex.
- Boot location: NOR flash.
- RCW source: NOR flash.

4.4.19.3 U-Boot Environment Variables

4.4.19.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, is as follows:

```
hwconfig=fsl_ddr:ctlr_intlv=3way_4KB,bank_intlv=auto;fsl_fm1_xaui_phy:xfi;fsl_fm2_xaui_phy:xfi;usb
1:dr_mode=host,phy_type=utmi
```

4.4.19.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.19.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for T4240 version information (e.g., T4240E version 2.0) and also for the version of FMan microcode currently flashed on the T4240QDS (e.g. microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 2.0:

```
fsl_fman_ucode_t4240_r2.0_106_4_18.bin (*)
fsl_fman_ucode_t4240_r2.0_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.19.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK contains RCW binaries for use on the T4240QDS:

- RR_SSFFPRPH_27_55_5_11/rcw_27_55_5_11_1800MHz.bin
- RR_XXQQPRPH_1_37_5_11/rcw_1_37_5_11_1800MHz.bin
- RR_XXSSPRPH_1_27_5_11/rcw_1_27_5_11_1800MHz.bin (default)
- RR_XXXXPRPR_1_1_5_5/rcw_1_1_5_5_1800MHz.bin
- RR_SSFFPRPH_27_55_5_11/rcw_27_55_5_11_1666MHz_rev2.bin
- RR_XXQQPRPH_1_37_5_11/rcw_1_37_5_11_1666MHz_rev2.bin
- RR_XXSSPRPH_1_27_5_11/rcw_1_27_5_11_1666MHz_rev2.bin
- RR_XXXXPRPR_1_1_5_5/rcw_1_1_5_5_1666MHz_rev2.bin

The RCW directories' names for the T4240QDS conform to the following naming convention:

```
ab_cdefghij_k_l_m_n
```

Table 104. T4240QDS Naming Convention Legend

Slot	Convention

Table continues on the next page...

Table 104. T4240QDS Naming Convention Legend (continued)

a	'R' if RGMII@FM2-DTSEC5 is supported 'N' if not available/not used
b	'R' if RGMII@FM1-DTSEC5 or RGMII@FM2-DTSEC6 is supported 'N" if not available/not used
c [What is available in Slot 1] d [What is availabel in Slot 2] e [What is available in Slot 3] f [What is available in Slot 4] g [What is available in Slot 5] h [What is available in Slot 6] i [What is available in Slot 7] j [What is available in Slot 8]	'N' if not available/not used 'P' if PCIe 'X' if XAUI 'R' if SRIO 'S' if SGMII 'H' if SATA 'F' XFI 'A' is AURORA
k	SerDes1 protocol value (decimal notation)
l	SerDes2 protocol value (decimal notation)
m	SerDes3 protocol value (decimal notation)
n	SerDes4 protocol value (decimal notation)

For example,

```
RR_SSFSPRPH_27_55_5_11
```

means:

- RGMII@FM2-DTSEC5
- RGMII@FM2-DTSEC6
- SGMII in Slot 1
- SGMII in Slot 2
- XFI in Slot 3
- SGMII in Slot 4
- PCIe in Slot 5
- SRIO in Slot 6
- PCIe in Slot 7
- SATA [Slot 8 not used]
- SerDes1 Protocol is 27
- SerDes2 Protocol is 55
- SerDes3 Protocol is 5
- SerDes4 Protocol is 11

4.4.19.6 System Memory Map

After system startup, the boot loader maps physical memory space:

Start Physical Address	End Physical Address	Discription	Size
0xF_FFDF_0000	0xF_FFDF_0FFF	IFC - QIXIS	4 KB
0xF_FF80_0000	0xF_FF8F_FFFF	IFC - NAND Flash	1 MB
0xF_F803_0000	0xF_F803_FFFF	PCI Express 4 I/O Space	64 KB
0xF_F802_0000	0xF_F802_FFFF	PCI Express 3 I/O Space	64 KB
0xF_F800_0000	0xF_F800_FFFF	PCI Express 1 I/O Space	64 KB
0xF_F600_0000	0xF_F7FF_FFFF	Qman	32 MB
0xF_F400_0000	0xF_F5FF_FFFF	Bman	32 MB
0xF_E000_0000	0xF_E7FF_FFFF	IFC - NOR Flash	128 MB
0xF_0000_0000	0xF_003F_FFFF	DCSR	4 MB
0xC_6000_0000	0xC_7FFF_FFFF	PCI Express 4 Mem Space	512 MB
0xC_4000_0000	0xC_5FFF_FFFF	PCI Express 3 Mem Space	512 MB
0xC_3000_0000	0xC_3FFF_FFFF	Serial Rapid I/O 2	256 MB
0xC_2000_0000	0xC_2FFF_FFFF	Serial Rapid I/O 1	256 MB
0xC_0000_0000	0xC_1FFF_FFFF	PCI Express 1 Mem Space	512 MB
0x0_0000_0000	0x0_7FFF_FFFF	Memory controller	2 GB

4.4.19.7 Flash Bank Usage

The NOR flash on the board can be seen as eight flash banks. The board DIP switch configuration (for T4240QDS, SW6[1:4]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps

bank 0 with the alternate bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log.

```
CPU0:  T4240E, Version: 2.0, (0x82480020)
Core:  e6500, Version: 2.0, (0x80400120)
Clock Configuration:
       CPU0:1800 MHz, CPU1:1800 MHz, CPU2:1800 MHz, CPU3:1800 MHz,
       CPU4:1800 MHz, CPU5:1800 MHz, CPU6:1800 MHz, CPU7:1800 MHz,
       CPU8:1800 MHz, CPU9:1800 MHz, CPU10:1800 MHz, CPU11:1800 MHz,
       CCB:733.333 MHz,
       DDR:933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:183.333 MHz
       FMAN1: 733.333 MHz
       FMAN2: 733.333 MHz
       QMAN:  366.667 MHz
       PME:   533.333 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 1607001b 18101b16 00000000 00000000
       00000010: 04362858 30548c00 ec020000 f5000000
       00000020: 00000000 ee0000ee 00000000 000307fc
       00000030: 00000000 00000000 00000000 00000028
Board: T4240QDS, Sys ID: 0x1e, Sys Ver: 0x12, vBank: 0
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=>qixis_reset
```

- Switch to alternate bank:

```
=>qixis_reset altbank
```

The table below shows the T4240QDS NOR flash memory map.

Table 105. NOR Flash Memory Map

Range Start	Range End	Definition	Size
0xEC000000	0xEC01FFF	RCW (alternate bank)	128KB
0xEFF40000	0xEFFFFFFF	U-Boot (current bank)	768KB
0xEFF20000	0xEFF3FFFF	U-Boot env (current bank)	128KB
0xEFF00000	0xEFF1FFFF	FMAN microcode (current bank)	128KB
0xEFE00000	0xEFE3FFFF	CORTINA PHY code (current bank)	256KB

Table continues on the next page...

Table 105. NOR Flash Memory Map (continued)

0xED300000	0xEFDFFFFF	rootfs (alternate bank)	44MB
0xED000000	0xED2FFFFF	Guest image #3(alternate bank)	3MB
0xECD00000	0xECFFFFFF	Guest image #2 (alternate bank)	3MB
0xECA00000	0xECCFFFFF	Guest image #1 (alternate bank)	3MB
0xEC900000	0xEC9FFFFF	HV config device tree (alternate bank)	1MB
0xEC800000	0xEC8FFFFF	Hardware device tree (alternate bank)	1MB
0xEC700000	0xEC7FFFFF	HV.uImage (alternate bank)	1MB
0xEC020000	0xEC6FFFFF	Linux.uImage (alternate bank)	6MB + 875KB
0xE8000000	0xE801FFFF	RCW (current bank)	128KB
0xEBF40000	0xEBFFFFFF	U-Boot (alternate bank)	768KB
0xEBF20000	0xEBF3FFFF	U-Boot env (alternate bank)	128KB
0xEBF00000	0xEBF1FFFF	FMAN microcode (alternate bank)	128KB
0xEBE00000	0xEBE3FFFF	CORTINA PHY code (alternate bank)	256KB
0xE9300000	0xEBDFFFFF	rootfs (current bank)	44MB
0xE9000000	0xE92FFFFF	Guest image #3 (current bank)	3MB
0xE8D00000	0xE8FFFFFF	Guest image #2 (current bank)	3MB
0xE8A00000	0xE8CFFFFF	Guest image #1 (current bank)	3MB
0xE8900000	0xE89FFFFF	HV config device tree (current bank)	1MB

Table continues on the next page...

Table 105. NOR Flash Memory Map (continued)

0xE8800000	0xE88FFFFF	Hardware device tree (current bank)	1MB
0xE8700000	0xE87FFFFF	HV.uImage (current bank)	1MB
0xE8020000	0xE86FFFFF	Linux.uImage (current bank)	6MB + 875KB

4.4.19.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command *qixis altbank* only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

By default, an existing U-Boot is run in bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing *qixis*. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
=>qixis altbank
```

The commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections.

Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *qixis*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address <rcw_start_addr>. <rcw_start_addr> is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 1000000 <rcw_file_name>.bin
=>protect off <rcw_start_addr> +$filesize
=>erase <rcw_start_addr> +$filesize
=>cp.b 1000000 <rcw_start_addr> $filesize
=>protect on <rcw_start_addr> +$filesize
=>qixis altbank
```

Programming a New Microcode

To program a new microcode, first switch to bank 0 by performing a hard reset or by typing *qixis*. Next, load the new microcode to RAM by downloading it via TFTP and then copying it to flash at address <fman_ucode_start_addr>.

<fman_ucode_start_addr> is the location of ucode in the alternate bank. Then execute the following commands at the U-Boot prompt to program the ucode to flash and reset to alternate bank.

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
=>qixis altbank
```

4.4.19.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.19.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 1000000 <uImage_name>
   =>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 2000000 <platform_dtb_name>
   =>bootm 1000000 5000000 2000000
   ```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.19.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   ```

```
=>saveenv
```

Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

   The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.19.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
=>bootm 1000000 - c00000
```

Now the board will boot up with NFS filesystem.

4.4.19.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

```
#fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to create the filesystems.

```
# mkfs.vfat /dev/sdb1
# mkfs.ext2 /dev/sdb2
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdb2 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

```
# cp uImage and uImage.dtb to /temp/boot folder
# umount /temp
```

8. Plug in the SD card to the target board and power on.

4.4.19.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

```
# fdisk /dev/sdx
```

3. Use the mkfs command to create the filesystems.

```
# mkfs.ext2 /dev/sdx1
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdx1 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure copy the target kernel image and dtb to sata.

```
# cp uImage and target.dtb to /temp/boot folder
# umount /temp
```

8. Connect the SATA hard drive to the target board and power on.

4.4.19.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

   TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

   ```
   =>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>erase e9300000 +$filesize
   =>cp.b 1000000 e9300000 $filesize
   ```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

   ```
   =>tftp 1000000 hv.uImage
   =>erase e8700000 +$filesize
   =>cp.b 1000000 e8700000 $filesize
   ```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

Target Deployment - for hv-2p mode deployment:

```
=>tftp 1000000 <platform_name>-usdpaa.dtb
=>erase e8800000 +$filesize
=>cp.b 1000000 e8800000 $filesize
```

Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

   Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.20 T4240RDB

4.4.20.1 Overview

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

4.4.20.2 Switch Settings

The RDB has user selectable switches for evaluating different boot options for the T4240 device. The table below lists the default switch settings. For a description of these settings, see T4240RDB User Guide.

Table 106. Default Switch Settings

Switch	1	2	3	4
SW1	OFF [1]	OFF [1]	ON [0]	ON [0]
SW2	OFF	OFF	OFF	OFF
SW3	OFF	OFF	OFF	OFF
SW4	OFF	OFF	OFF	OFF

Note that with the default switch settings, the following are selected:

Table 107. Default Settings

Switch	Default Setting
SW1[1:4] = 0011	SYSCLK output frequency is 66.67 MHz DDRCLK is 133.33 MHz CPU 1800 MHz
SW3[4]= 1	RCW configuration source is I2C EEPROM Flash
SW4[1:2]= 11	Normal power on/off T4240 mode is selected

NOTE

For the T4240RDB, OFF = 1 and ON = 0.

4.4.20.3 U-Boot Environment Variables

4.4.20.3.1 U-Boot Environment Variable “hwconfig”

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set from the U-Boot prompt using the "*setenv*" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified from the U-Boot command prompt using the "*editenv*" command.
- It can be saved in the U-Boot environment via the "*saveenv*" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for DDR, which enables chip select interleaving and cacheline interleaving, as well as the USB1 configuration is as follows:

```
hwconfig=fsl_ddr:ctlr_intlv=3way_4KB,bank_intlv=auto;usb1:dr_mode=host,phy_type=utmi
```

4.4.20.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>
=>setenv serverip <tftp_serverip>
=>setenv gatewayip <your_gatewayip>
=>setenv ethaddr <mac addr0>
=>setenv eth1addr <mac addr1>
=>setenv eth2addr <mac addr2>
=>setenv eth3addr <mac addr3>
=>setenv eth4addr <mac addr4>
=>setenv eth5addr <mac addr5>
=>setenv ethprime <ethx>
=>setenv ethact <ethx>
=>setenv netmask 255.255.x.x
=>saveenv
```

NOTE

(i) <ethx> is the Ethernet port on the board connected to the Linux boot server.

(ii) "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:04:9F:02:00:FD
=>setenv eth1addr 00:04:9F:02:01:FD
=>setenv eth2addr 00:04:9F:02:02:FD
=>setenv eth3addr 00:04:9F:02:03:FD
=>setenv eth4addr 00:04:9F:02:04:FD
```

```
=>setenv eth5addr 00:04:9F:02:05:FD
=>saveenv
```

NOTE

For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

4.4.20.4 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for T4240 version information (e.g., T4240E version 2.0) and also for the version of FMan microcode currently flashed on the T4240RDB (e.g. microcode version 106.4.18). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

- For silicon revision 2.0

```
fsl_fman_ucode_t4240_r2.0_106_4_18.bin (*)
fsl_fman_ucode_t4240_r2.0_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) For instructions on how to flash a new FMan microcode image, see Programming a new U-Boot, RCW, and FMan Microcode.

(iv) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

4.4.20.5 RCW (Reset Configuration Word) and Ethernet Interfaces

QorIQ SDK contains the following RCW binary for use on the T4240RDB:

• SSFFPPH_27_55_1_9/rcw_27_55_1_9_1800MHz.bin (default)

• SSFFPPH_27_55_1_9/rcw_27_55_1_9_1666MHz.bin

The RCW directory name for T4240RDB conform to the following naming convention:

```
abcdefg_h_i_j_k
```

So the default,

```
SSFFPPH_27_55_1_9
```

means:

Table 108. T4240RDB Directory Legend

Slot	Convention

Table continues on the next page...

Table 108. T4240RDB Directory Legend (continued)

a	'S' indicates FM1 @ DTSEC1 DTSEC2 DTSEC3 DTSEC4 ports are supported
b	'S' indicates FM2 @ DTSEC1 DTSEC2 DTSEC3 DTSEC4 ports are supported
c	'F' indicates FM1 @ TGEC1 TGEC2 SFP+ fiber ports are supported
d	'F' indicates FM2 @ TGEC1 TGEC2 SFP+ fiber ports are supported
e	'P' indicates PCIe x8 in the PCI slot
f	'P' indicates PCIe x4 in the PCI slot
g	'H' indicates SATA port is supported
h	'27' indicates SerDes1 protocol 0x1B
i	'55' indicates SerDes2 protocol 0x37
j	'1' indicates SerDes3 protocol 0x1
k	'9' indicates SerDes4 protocol 0x9

Below is a table that maps the different ports available on the T4240RDB and their names in different environments.

Table 109. Port Map

Label on Panel of T4240 RDB	Port in U-Boot	Port in Linux
ETH0	FM1@ DTSEC1	fm1-mac1
ETH1	FM1@ DTSEC2	fm1-mac2
ETH2	FM1@ DTSEC3	fm1-mac3
ETH3	FM1@ DTSEC4	fm1-mac4
ETH4	FM2@ DTSEC1	fm2-mac1
ETH5	FM2@ DTSEC2	fm2-mac2
ETH6	FM2@ DTSEC3	fm2-mac3
ETH7	FM2@ DTSEC4	fm2-mac4
ETH8	FM2@ TGEC1	fm2-mac9
ETH9	FM2@ TGEC2	fm2-mac10
ETH10	FM1@ TGEC2	fm1-mac10
ETH11	FM1@ TGEC1	fm1-mac9

4.4.20.6 System Memory Map

After system startup, the boot loader maps physical memory space as shown below.

Start Physical Address	End Physical Address	Definition	Size
0xff8030000	0xff803ffff	PCI Express 4 I/O Space	64 KB
0xff8020000	0xff802ffff	PCI Express 3 I/O Space	64 KB
0xff8000000	0xff800ffff	PCI Express 1 I/O Space	64 KB
0xff6000000	0xff7ffffff	QMan	32 MB
0xff4000000	0xff5ffffff	BMan	32 MB
0xfe0000000	0xfe7ffffff	IFC - NOR Flash	128 MB
0xf00000000	0xf003fffff	DCSR	4 MB
0xc60000000	0xc7fffffff	PCI Express 4 Mem Space	512 MB
0xc40000000	0xf5fffffff	PCI Express 3 Mem Space	512 MB
0xc00000000	0xc1fffffff	PCI Express 1 Mem Space	512 MB
0x000000000	0x07fffffff	Memory Controller	2 GB

4.4.20.7 Flash Bank Usage

The NOR flash on the board can be seen as eight flash banks. The board DIP switch configuration (for T4240RDB, SW3[1:3]) preselects bank 0 as the hardware default bank.

To protect the default U-Boot in bank 0, it is a convention employed by NXP to deploy work images into the alternate bank, and then switch to the alternate bank for testing. Switching to the alternate bank can be done in software and effectively swaps bank 0 with the alternate bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log.

```
CPU0:  T4240E, Version: 2.0, (0x82480020)
Core:  e6500, Version: 2.0, (0x80400120)
Clock Configuration:
       CPU0:1800 MHz, CPU1:1800 MHz, CPU2:1800 MHz, CPU3:1800 MHz,
       CPU4:1800 MHz, CPU5:1800 MHz, CPU6:1800 MHz, CPU7:1800 MHz,
       CPU8:1800 MHz, CPU9:1800 MHz, CPU10:1800 MHz, CPU11:1800 MHz,
       CCB:733.333 MHz,
```

```
       DDR:933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:183.333 MHz
       FMAN1: 733.333 MHz
       FMAN2: 733.333 MHz
       QMAN:  366.667 MHz
       PME:   533.333 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Reset Configuration Word (RCW):
       00000000: 16070019 18101916 00000000 00000000
       00000010: 6c6e0848 00448c00 6c020000 f5000000
       00000020: 00000000 ee0000ee 00000000 000287fc
       00000030: 00000000 50000000 00000000 00000028
Board: T4240RDB, Board rev: 0x04 CPLD ver: 0x0401, vBank: 0
```

Bank switching can be done in U-Boot using the following statements:

- Switch to bank 0:

```
=> cpld reset
```

- Switch to alternate bank:

```
=> cpld reset altbank
```

The table below shows the T4240RDB 128MB Nor flash memory map. The flash memory used is configured in a 16-bit port size.

Table 110. NOR Flash Memory Map

Range Start	Range End	Definition	Size
0xEC000000	0xEC01FFF	RCW (alternate bank)	128KB
0xEFF40000	0xEFFFFFFF	U-Boot (current bank)	768KB
0xEFF20000	0xEFF3FFFF	U-Boot env (current bank)	128KB
0xEFF00000	0xEFF1FFFF	FMAN microcode (current bank)	128KB
0xEFE00000	0xEFE3FFFF	CORTINA PHY code (current bank)	256KB
0xED300000	0xEFDFFFFF	rootfs (alternate bank)	44MB
0xED000000	0xED2FFFFF	Guest image #3(alternate bank)	3MB
0xECD00000	0xECFFFFFF	Guest image #2 (alternate bank)	3MB
0xECA00000	0xECCFFFFF	Guest image #1 (alternate bank)	3MB

Table continues on the next page...

Table 110. NOR Flash Memory Map (continued)

0xEC900000	0xEC9FFFFF	HV config device tree (alternate bank)	1MB
0xEC800000	0xEC8FFFFF	Hardware device tree (alternate bank)	1MB
0xEC700000	0xEC7FFFFF	HV.uImage (alternate bank)	1MB
0xEC020000	0xEC6FFFFF	Linux.uImage (alternate bank)	6MB + 875KB
0xE8000000	0xE801FFFF	RCW (current bank)	128KB
0xEBF40000	0xEBFFFFFF	U-Boot (alternate bank)	768KB
0xEBF20000	0xEBF3FFFF	U-Boot env (alternate bank)	128KB
0xEBF00000	0xEBF1FFFF	FMAN microcode (alternate bank)	128KB
0xEBE00000	0xEBE3FFFF	CORTINA PHY code (alternate bank)	256KB
0xE9300000	0xEBDFFFFF	rootfs (current bank)	44MB
0xE9000000	0xE92FFFFF	Guest image #3 (current bank)	3MB
0xE8D00000	0xE8FFFFFF	Guest image #2 (current bank)	3MB
0xE8A00000	0xE8CFFFFF	Guest image #1 (current bank)	3MB
0xE8900000	0xE89FFFFF	HV config device tree (current bank)	1MB
0xE8800000	0xE88FFFFF	Hardware device tree (current bank)	1MB
0xE8700000	0xE87FFFFF	HV.uImage (current bank)	1MB
0xE8020000	0xE86FFFFF	Linux.uImage (current bank)	6MB + 875KB

For board revision T4240RDB-PD, the NOR flash can be split into two logical halves by setting the *FBANK_SEL* signal. The *FBANK_SEL* signal is controlled by setting SW3[3]. The table below displays how the addresses are changed when using *FBANK_SEL*.

Table 111. Logical NOR Banks

Setting	NOR bank used
SW3[3] = on	Boot from vbank0
SW3[3] = off	Boot from vbank4

NOTE

T4240RDB-PB does not support *FBANK_SEL* feature.

4.4.20.8 Programming a New U-boot, RCW, FMan Microcode

The following three sections will discuss how to individually update U-Boot, RCW, and FMan Microcode. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash all three at once, reset the board only after U-Boot, RCW and FMan Microcode have all been programmed.

Prior to continuing with the following instructions, please refer to Configuring U-Boot Network Parameters to make sure all necessary U-Boot parameters have been set.

Programming a New U-Boot

Use the following commands to flash a new U-Boot:

```
=>tftp 1000000 <u-boot_file_name>.bin
=>protect off <u-boot_start_addr> +$filesize
=>erase <u-boot_start_addr> +$filesize
=>cp.b 1000000 <u-boot_start_addr> $filesize
=>protect on <u-boot_start_addr> +$filesize
```

After implementing the commands, reset the board to boot it up. Note that the commands above will only program a new U-Boot. Programming a new RCW and a new microcode will be discussed in the next sections. If the user intends on programming a new U-Boot as well as a new RCW and Microcode, wait to reset the board until commands for programming all three sections have been implemented.

Programming a New RCW

At the U-Boot prompt, use the following commands to program a new RCW:

```
=>tftp  1000000  <rcw_file_name>.bin
=>i2c  write  0x1000000  0x50  0  $filesize
```

After implementing the commands above, reset the board to boot it up. Note that the commands above will only program a new RCW.

Programming a New Microcode

Execute the following commands at the U-Boot prompt to program the ucode:

```
=>tftp 1000000 <ucode_file_name>.bin
=>protect off <fman_ucode_start_addr> +$filesize
=>erase <fman_ucode_start_addr> +$filesize
=>cp.b 1000000 <fman_ucode_start_addr> $filesize
=>protect on <fman_ucode_start_addr> +$filesize
```

After implementing the commands above, reset the board to boot it up. Note that the commands above will only program a new microcode.

4.4.20.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

4.4.20.9.1 Ramdisk Deployment from TFTP

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment. Before performing ramdisk deployment, the U-Boot environment variables need to be configured.

   Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for ramdisk deployment from TFTP:

   ```
   =>setenv bootargs 'root=/dev/ram rw console=ttyS0,115200 ramdisk_size=10000000 log_buf_len=128K'
   =>saveenv
   ```

   NOTE

   ramdisk_size needs to be set if the ramdisk uncompress file size is bigger than default setting. It should be more than ramdisk uncompress file size. The file size information is printed in Yocto Project build log.

2. **Booting Up the System**

   Execute the following commands to TFTP the images to the board, then boot into Linux.

   ```
   =>tftp 1000000 <uImage_name>
   =>tftp 5000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>tftp 2000000 <platform_dtb_name>
   =>bootm 1000000 5000000 2000000
   ```

   Now the board will boot into Linux using the images generated by Yocto Project.

4.4.20.9.2 Ramdisk Deployment from Flash

Programming the kernel and ramdisk into flash will allow you to boot up the board afterwards without the need to re-download images.

1. **Setting U-Boot Environment**

   The images generated by Yocto Project allow you to perform ramdisk deployment from flash. Before performing ramdisk deployment from flash, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment from flash:

   ```
   =>setenv ramargs 'setenv bootargs root=/dev/ram rw console=ttyS0,115200'
   =>setenv bootcmd 'run ramargs; bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>'
   =>saveenv
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming Kernel ITB to NOR Flash**

The kernel should be downloaded to the RAM using TFTP then copied to the flash address <kernel_itb_addr>. At the U-Boot prompt, use the following commands to program the kernel to flash:

```
=>tftp 1000000 <uImage name>
=>protect off <kernel_start_addr> +$filesize
=>erase <kernel_start_addr> +$filesize
=>cp.b 1000000 <kernel_start_addr> $filesize
=>protect on <kernel_start_addr> +$filesize
```

3. **Programming Ramdisk to NOR Flash**

   The ramdisk should be downloaded to the RAM then copied to the flash address <ramdisk_start_addr>. At the U-Boot prompt, use the following commands to program the ramdisk to flash:

```
=>tftp 2000000 fsl-image-core-<platform>.ext2.gz.u-boot
=>protect off <ramdisk_start_addr> +$filesize
=>erase <ramdisk_start_addr> +$filesize
=>cp.b 2000000 <ramdisk_start_addr> $filesize
=>protect on <ramdisk_start_addr> +$filesize
```

4. **Programming Device Tree File to NOR Flash**

   The dtb file should be downloaded to the RAM, then copied to the flash address <dtb_start_addr>. At the U-Boot prompt, use the following commands to program the dtb file to flash:

```
=>tftp c00000 <platform dtb name>
=>protect off <dtb_start_addr> +$filesize
=>erase <dtb_start_addr> +$filesize
=>cp.b c00000 <dtb_start_addr> $filesize
=>protect on <dtb_start_addr> +$filesize
```

5. **Booting Up the System**

   The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

   or

```
=> bootm <kernel_start_addr> <ramdisk_start_addr> <dtb_start_addr>
```

4.4.20.9.3 NFS Deployment

1. **Generating File System with Yocto Project**

   Use Yocto Project to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. **Setting Host NFS Server Environment**

   a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
<nfs_root_path> <board_ipaddress>(rw,no_root_squash,async)
```

   b. Restart the NFS service:

```
/etc/init.d/nfs restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto Project allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>
ip=<board_ipaddr>:<tftp_serverip>:
<your_gatewayip>:<your_netmask>:<board_name>:eth0:off console=ttyS0,115200
=>setenv netdev <ethx>
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

   TFTP the kernel image to the board, then boot it up.

```
=>tftp 1000000 <uImage name>
=>tftp c00000 <platform dtb name>
=>bootm 1000000 - c00000
```

   Now the board will boot up with NFS filesystem.

4.4.20.9.4 SD Deployment

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SD card, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootfile uImage
# setenv fdtfile uImage.dtb
# setenv bootcmd 'setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,
$baudrate;mmcinfo;ext2load mmc 0:2 $loadaddr   /boot/$bootfile;ext2load mmc 0:2 $fdtaddr /boot/
$fdtfile;bootm $loadaddr - $fdtaddr'
# save
```

Deploy Filesystem to the SD Card

Once the U-Boot network parameters have been set, follow the steps below to deploy the filesystem to the SD card:

1. Connect the card reader with SD card to the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdb", one MS-DOS partition(sdb1 and this partition usually for u-boot) and one ext2 partition(sdb2) in the SD card.

```
#fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to create the filesystems.

```
# mkfs.vfat /dev/sdb1
# mkfs.ext2 /dev/sdb2
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdb2 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure the kernel image and dtb file are in /temp/boot directory, then umount the /temp.

```
# cp uImage and uImage.dtb to /temp/boot folder
# umount /temp
```

8. Plug in the SD card to the target board and power on.

4.4.20.9.5 Harddisk Deployment

The Linux kernel and filesystem can be put to the SATA hard disk for production deployment (harddisk deployment).

Setting U-Boot Environment

You can place the ext2 filesystem and kernel on the SATA harddisk, then the kernel can boot up automatically after the board is powered on or after reset. Prior to this deployment, make sure U-Boot parameters have been set up:

```
# setenv bootcmd 'setenv bootargs root=/dev/sdx1 rw console=ttyS0,115200;sata init;ext2load sata
0:1 1000000 /boot/uImage;ext2load sata 0:1 c00000 /boot/target.dtb;bootm 1000000 - c00000'
# save
```

Deploying Filesystem to the Harddisk

Once U-Boot network parameters have been set, follow the steps below to start Harddisk deployment. We need one ext2 partition. To make this we need one Linux Host PC.

1. Connect the SATA hard drive with Serial ATA cable and turn on the Linux Host PC.
2. Create the partitions by "fdisk /dev/sdx, one ext2 partition(sdx1) in the Hard disk. The sdx is sda, sdb, sdc, ... depending your host

```
# fdisk /dev/sdx
```

3. Use the mkfs command to create the filesystems.

```
# mkfs.ext2 /dev/sdx1
```

4. Create temp directory in host PC and mount the ext2 partition to the temp.

```
# mkdir /temp
# mount /dev/sdx1 /temp
# cd /temp
```

5. Mount the target board type .iso (eg. QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso) to get the tarball (eg. QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz) on host PC.

```
# mount -o loop QorIQ-SDK-<Version>-PPCE6500-<release date>-yocto.iso /work
```

6. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs.

```
# cp QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz .
# sudo tar -zxvf QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
# rm QorIQ_SDK_<Version>_E6500_<release date>_ROOTFS_Image.tar.gz
```

7. Make sure copy the target kernel image and dtb to sata.

```
# cp uImage and target.dtb to /temp/boot folder
# umount /temp
```

8. Connect the SATA hard drive to the target board and power on.

4.4.20.9.6 Hypervisor Deployment

Introduction

Use the mux_server to connnect to the board to ensure that individual UART streams are decoded correctly on the host side.

```
./mux_server "/dev/ttySx"<port number 1> <port number 2> <port number 3> &
socat -,raw,echo=0 tcp:0:<port number 1>
socat -,raw,echo=0 tcp:0:<port number 2>
socat -,raw,echo=0 tcp:0:<port number3>
```

NOTE

Note: 'ttySx' should be replaced with the actual serial device that is used. The port number 1 relates to the U-Boot console, port number 2 related to Linux partition 1, port number 3 related to Linux partition 2. The total number of port numbers is 10.

For example, we assume mux_server has been run with three port numbers, starting at 15000:

```
socat -,raw,echo=0 tcp:0:15000 socat -,raw,echo=0 tcp:0:15001 socat -,raw,echo=0 tcp:0:15002
```

NOTE

The Linux-based mux_server utility is provided to support multiplexing and demultiplexing capabilities. The mux_server runs on a host system and attaches to the target system via a serial or network communications channel Configuring U-Boot for hv-2p Deployment.

Refer to Configuring U-Boot Network Parameters on page 102 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for hypervisor deployment:

```
=>setenv bootargs config-addr=0xfe8900000 console=ttyS0,115200
=>setenv bootcmd 'bootm 0xfe8700000 - 0xfe8800000'
=>saveenv
```

Now U-Boot is ready for hypervisor deployment.

Hypervisor Deployment

Now, the kernel, hypervisor image, device tree, hypervisor device tree and ramdisk filesystem can be flashed onto the board. These steps should be done assuming the user already has switched to the alternate bank.

1. **Programming Kernel to Flash**

   TFTP the kernel image to RAM, then copy it to the flash address 0xe8020000. Execute the following commands at the U-Boot prompt to program the kernel to flash:

   ```
   =>tftp 1000000 uImage
   =>erase e8020000 +$filesize
   =>cp.b 1000000 e8020000 $filesize
   ```

2. **Programming Ramdisk Filesystem to Flash**

   TFTP the ramdisk file system to RAM, then copy it to the flash at address 0xe9300000. Execute the following commands at U-Boot prompt to program the ramdisk to flash:

   ```
   =>tftp 1000000 fsl-image-core-<platform>.ext2.gz.u-boot
   =>erase e9300000 +$filesize
   =>cp.b 1000000 e9300000 $filesize
   ```

3. **Programming Hypervisor Image to Flash**

   TFTP the hypervisor images to RAM, then copy it to the flash at address 0xe8700000. Execute the following commands at U-Boot prompt to program the hypervisor image to flash:

   ```
   =>tftp 1000000 hv.uImage
   =>erase e8700000 +$filesize
   =>cp.b 1000000 e8700000 $filesize
   ```

4. **Programming Kernel dtb to Flash**

   TFTP the kernel dtb file to ram, then copy it to the flash at address 0xe8800000. Execute the following commands at U-Boot prompt to program the kernel dtb to flash:

   Target Deployment - for hv-2p mode deployment:

   ```
   =>tftp 1000000 <platform_name>-usdpaa.dtb
   =>erase e8800000 +$filesize
   =>cp.b 1000000 e8800000 $filesize
   ```

   Note: Program "hv-2p-lnx-lnx.dtb" to 0xe8800000

5. **Booting Up the System**

Note:The hv-2p configuration needs to run with <platform_name>-usdpaa.dtb. As of now, all the DPAA devices on this platform are given to USDPAA partition. The kernel can boot up automatically after the board is powered on with the correct U-Boot environment. The following command can also be used to boot up the board at U-Boot prompt:

```
=>boot
```

4.4.20.10 Check 'Link Up' for Serial Ethernet Interfaces

If you are experiencing problems with your Ethernet interfaces, this section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors.

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
0 - Vitesse VSC8662 <--> FM1@DTSEC1
1 - Vitesse VSC8662 <--> FM1@DTSEC2
2 - Vitesse VSC8662 <--> FM1@DTSEC3
3 - Vitesse VSC8662 <--> FM1@DTSEC4
4 - Vitesse VSC8662 <--> FM2@DTSEC1
5 - Vitesse VSC8662 <--> FM2@DTSEC2
6 - Vitesse VSC8662 <--> FM2@DTSEC3
7 - Vitesse VSC8662 <--> FM2@DTSEC4
FM_TGEC_MDIO:
16 - Cortina CS4315/CS4340 <--> FM1@TGEC1
17 - Cortina CS4315/CS4340 <--> FM1@TGEC2
18 - Cortina CS4315/CS4340 <--> FM2@TGEC2
19 - Cortina CS4315/CS4340 <--> FM2@TGEC1
```

The results from the above *mdio list* command show that U-Boot was able to see PHYs on each of the 1G and 10G interfaces. If you see “Generic” reported, it is an indication that something is there but the T4240 can’t communicate with the device/port.

Check SGMII Link Status for External PHY

In order to check the status of a SGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read FM1@DTSEC1 1
Reading from bus FSL_MDIO0
PHY at address 0:
1 - 0x79ed
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0."

Check SGMII link Status for Internal PHY

The SGMII PCS on-chip also has link information. In this section we'll check for a valid link for FM1 EMAC1 (listed as FM1@DTSEC1 in U-Boot). For this check, if you want to see a valid link reported, connect the interface to a link partner.

For FM1 EMAC1 (FM1@DTSEC1), the register we want to read is SGMII Status Register (MDIO_SGMII_SR). In particular, we will check MDIO_SGMII_SR[LINK_STAT]. The register MDIO_SGMII_SR is in the SerDes SGMII MDIO register space at offset 0x1.

Note that for SGMII PCS we are using a Clause 22 device.

We will use the following registers in the in the Ethernet MAC controller register space to perform the read:

```
Offset 0x34 Management interface control register (MDIO_CTL)
Offset 0x38 Management interface data register (MDIO_DATA)
```

We will program MDIO_CTL so that a read of MDIO_DATA will return the contents of MDIO_SGMII_SR. The first step is to identify the values needed to write to MDIO_CTL:

Since 0x1 is the Offset of MDIO_SGMII_SR is the SGMII MDIO register space set MDIO_CTL[DEV_ADDR] = 0x1.

The SGMII MDIO register space is selected when the associated SGMIInCR1[MDEV_PORT] matches the Ethernet MAC PHY address (MDIO_CTL[PHY_ADDR]). So we will set:

```
MDIO_CTL[DEV_ADDR] = 0x1
MDIO_CTL[PHY_ADDR] = SGMIInCR1[MDEV_PORT]
```

Note that SGMII Protocol Control Register 1 (SGMIInCR1, n = A, B, ..., H) is in the SerDes register space.

In T4240RM see table 18.1.1.5 "MDIO port mapping" for the mapping of PCD MDIO Slave to MAC MDIO Master. For this example, we're using MAC MDIO master FM1-MAC1 and PCS MDIO slave SGMIIA (i.e., SGMIInCR1 where n=A).

Read SGMIIACR1:

```
=> md fe0ea604
fe0ea604: 000008bf 00000000 00000000 80000000    ................
```

So SGMIIACR1[MDEV_PORT] = 0.

So we will program MDIO_CTL with the following values:

```
MDIO_CTL[DEV_ADDR] = 0x1
MDIO_CTL[PHY_ADDR] = SGMIIACR1[MDEV_PORT] = 0
```

Read of FM1-EMAC1 MDIO_SGMII_SR:

```
=> mm fe4e1034                   <== FM1 (offset 0x400000) EMAC1 (offset 0xe1000) MDIO_CTL (offset
0x34)
fe4e1034: 00000003 ? 8001         <== Set MDIO_CTL[READ] = 1 MDIO_CTL[PHY_ADDR] = 0 and
MDIO_CTL[DEV_ADDR] = 1
fe4e1038: 00001001 ? x
=> md fe4e1030                    <== Read MDIO_DATA (offset 0x38) for contents of MDIO_SGMII_SR
fe4e1030: 40001408 00000001 0000000d 0000000d    @...............
```

The above shows that MDIO_SGMII_SR = 0x0000000d and so MDIO_SGMII_SR[LINK_STAT] = 1 indicating that the link is valid.

Check XFI Link Status for Internal PHY

The XFI PCS on-chip also has link information. In this section we'll checkfor a valid link for FM1 EMAC9 (listed as FM1@TGEC1 in U-Boot). For this check,if you want to see a valid link reported, connect the interface to a link partner.

For FM1 EMAC9 (FM1@TGEC1), the register we want to read is XFI 10GBASE-R PCS Status 1 (MDIO_XFI_PCS_10GR_SR1). In particular, we will check MDIO_XFI_PCS_10GR_SR1[RX_LINK_STAT]. The register MDIO_XFI_PCS_10GR_SR1 is in the SerDes XFI PCS register space at offset 0x20.

Note that for XFI PCS, we are using Clause 45 device.

We will use the following registers in the Ethernet MAC controller register space to perform the read:

```
Offset 0x30 Management interface control register (MDIO_CTRL)
Offset 0x38 Management interface data register (MDIO_DATA)
Offset 0x3C Management interface address register (MDIO_ADDR)
```

We will program MDIO_CTRL and MDIO_ADDR so that a read of MDIO_DATA will return the contents of MDIO_XFI_PCS_10GR_SR1. The first step is to identify the values needed to write to MDIO_ADDR and MDIO_CTL:

Since 0x20 is the Offset of MDIO_XFI_PCS_10GR_SR1 in XFI PCS register space we set MDIO_ADDR[REGADDR] = 0x20.

The XFI PCS register space is selected when the associated XFInCR1[MDEV_PORT] matches the Ethernet MAC port address (MDIO_CTL[PORT_ADDR]) and the MDIO_CTL[DEV_ADDR] is 0x3.

To summarize, we will set:

```
MDIO_ADDR[REGADDR] = 0x20
MDIO_CTL[DEV_ADDR] = 0x3
MDIO_CTL[PORT_ADDR] = XFInCR1[MDEV_PORT]
```

Note that XFI Protocol Control Register1 (XFInCR1, n = A, B, C, D) is in the SerDes register space.

In T4240RM see table 18.1.1.5 "*MDIO port mpaping*" for the mapping of PCD MDIO Slave to MAC MDIO Master. For this example, we're using MAC MDIO master FM1-MAC9 and PCS MDIO slave XFIC (i.e., XFInCR1 where n=C).

Read XFICCR1:

```
=> md fe0ea6e4
fe0ea6e4: 00000000 00000000 00000000 80000000    ................
```

So XFICCR1[MDEV_PORT] = 0.

So we will program MDIO_ADDR and MDIO_CTL with the following values:

```
MDIO_ADDR[REGADDR] = 0x20
MDIO_CTL[DEV_ADDR] = 0x3
MDIO_CTL[PORT_ADDR] = XFICCR1[MDEV_PORT] = 0
```

Read of FM1-EMAC9 MDIO_XFI_PCD_10GR_SR1:

```
=> mm fe4f103c                      <== FM1 (offset 0x400000) EMAC9 (offset 0xf1000) MDIO_ADDR
(offset 0x3c)
fe4f103c: 00000002 ? 20             <== Set MDIO_ADDR[REGADDR] = 0x20
fe4f1040: 00000000 ? x
=> mm fe4f1034                      <== FM1 (offset 0x400000) EMAC9 (offset 0xf1000) MDIO_CTL
```

```
(offset 0x34)
fe4f1034: 00000003 ? 8003           <== Set MDIO_CTL[READ] = 1, MDIO_CTL[PORT_ADDR] = 0,
MDIO_CTL[DEV_ADDR] = 0x3
fe4f1038: 00001001 ? x
=> md fe4f1030                      <== Read MDIO_DATA (offset 0x38) for contents of
MDIO_XFI_PCS_10GR_SR1
fe4f1030: 40001448 00000003 00001001 00001001    @..H............
```

The above shows that MDIO_XFI_PCS_10GR_SR1 = 0x00010001 and so MDIO_XFI_PCS_10GR_SR1[RX_LINK_STAT] = 1 indicating that 10GBase-R PCS receive link is up.

4.4.20.11 Basic Networking Ping Test

In Linux, in order to check that your network driver is set up, follow the commands below:

```
#ip addr show
#ip addr add <ip address of board>/24 brd + dev <port in Linux>
#ip link set <port in Linux> up
#ping <serverip>
```

If your network driver is not working, check your kernel messages. You should not see any errors reported by the FSL FMan MAC API based driver:

```
fsl_mac: fsl_mac: FSL FMan MAC API based driver ()
fsl_mac ffe4e0000.ethernet: FMan MEMAC
fsl_mac ffe4e0000.ethernet: FMan MAC address: 00:e0:0c:00:38:00
fsl_mac ffe4e2000.ethernet: FMan MEMAC
fsl_mac ffe4e2000.ethernet: FMan MAC address: 00:e0:0c:00:38:01
fsl_mac ffe4e4000.ethernet: FMan MEMAC
fsl_mac ffe4e4000.ethernet: FMan MAC address: 00:e0:0c:00:38:02
fsl_mac ffe4e6000.ethernet: FMan MEMAC
fsl_mac ffe4e6000.ethernet: FMan MAC address: 00:e0:0c:00:38:03
fsl_mac ffe4f0000.ethernet: FMan MEMAC
fsl_mac ffe4f0000.ethernet: FMan MAC address: 00:e0:0c:00:38:06
fsl_mac ffe4f2000.ethernet: FMan MEMAC
fsl_mac ffe4f2000.ethernet: FMan MAC address: 00:e0:0c:00:38:07
fsl_mac ffe5e0000.ethernet: FMan MEMAC
fsl_mac ffe5e0000.ethernet: FMan MAC address: 00:e0:0c:00:38:08
fsl_mac ffe5e2000.ethernet: FMan MEMAC
fsl_mac ffe5e2000.ethernet: FMan MAC address: 00:e0:0c:00:38:09
fsl_mac ffe5e4000.ethernet: FMan MEMAC
fsl_mac ffe5e4000.ethernet: FMan MAC address: 00:e0:0c:00:38:0a
fsl_mac ffe5e6000.ethernet: FMan MEMAC
fsl_mac ffe5e6000.ethernet: FMan MAC address: 00:e0:0c:00:38:0b
fsl_mac ffe5f0000.ethernet: FMan MEMAC
fsl_mac ffe5f0000.ethernet: FMan MAC address: 00:e0:0c:00:38:0e
fsl_mac ffe5f2000.ethernet: FMan MEMAC
fsl_mac ffe5f2000.ethernet: FMan MAC address: 00:e0:0c:00:38:0f
```

Chapter 5
System Recovery

NOTE

The following instructions can also be used for NOR flash recovery.

5.1 Environment Setup

5.1.1 Environment Setup (Common)

The section describes the related setup for system recovery

1. Required Materials
   - Target board
   - The related recovery image files
2. Host PC setup

   The host PC should have a serial-terminal program capable of running at 115,200bps, 8-N-1, for communicating with U-Boot running on the target board.
3. Target board setup
   a. Power off the target board system if the power is already on.
   b. If U-Boot runs on this board, and U-Boot commands will be used to reflash the U-Boot images, connect the target board to the network via the eTSEC port on the board.
   c. Connect the target board to the host machine via the serial port with an RS-232 cable and the joined NXP adapter cable, if needed.
   d. Set up the serial terminal in the host machine with 115,200bps, 8-N-1, no flow control.
   e. Verify all the switches and jumpers are set up correctly to default values described in <Hardware configurations> as described in the Switch Settings section of the board's Software Deployment Guide
   f. Connect the JTAG cable for your CodeWarrior TAP or Gigabit TAP to the board if you will be using the CodeWarrior Flash Programmer to recover the board image.
   g. Power on the board.

5.2 Image Recovery

5.2.1 Recover system with already working U-Boot

Target Board Setup

1. Power off the target board system if the power is already on.
2. Connect the target board to the network via the eTSEC port on the board.
3. Connect the target board to the host machine via the serial port with an RS-232 cable and the joined NXP adaptor cable, if needed.

4. Set up the serial terminal in the host machine with 115,200bps, 8-N-1, no flow control.
5. Verify all the switches and jumpers are setup correctly to default value described in the board's "Switch Settings" section in the board's Software Deployment Guide.
6. Power on the board.

Refer to the “Programming a New U-Boot…” section in the Software Deployment Guide for the target board to be recovered.

5.2.2 Recover system using CodeWarrior Flash Programmer

Environment Setup

Required Materials

- Target board.
- CodeWarrior for Power Architecture v10.x (for Windows or Linux)
- CodeWarrior TAP or Gigabit TAP run control device.
- The related recovery image files.

Host PC setup

The host PC is assumed to be running Windows 7, or one of the supported distributions of Linux (refer to the CodeWarrior PA10 Release Notes for the list of supported Linux distributions).

This machine should have latest CodeWarrior PA10 installed and working correctly. If the run control device is a CodeWarrior TAP used over USB, then the USB drivers should be installed automatically when the device is plugged in. If the run control device is a CodeWarrior TAP used over Ethernet, or a Gigabit TAP, then both the host PC and TAP should be connected to the network, and communications between them should be verified.

Target board setup

1. Power off the Target board system if the power is already on.
2. Connect the Target board to the host machine via the serial port with an RS-232 cable and the joined NXP adaptor cable, if needed.
3. Set up the serial terminal in the host machine with 115,200bps, 8-N-1, no flow control.
4. Verify all the switches and jumpers are set up correctly to default values described in the "Switch Settings" section in the board's Software Deployment Guide.
5. Connect the TAP's JTAG cable to the board.
6. Power on the board.

System Recovery

1. Start the CodeWarrior PA10 or ARMv7 IDE.
2. For LS102x targets, see Chapter 3 of *Getting Started for ARMv7 Processors.pdf* in the CodeWarrior ARMv7 installation for steps on creating a BareBoard Core0 project for the LS102x processor on this target board. For other QorIQ targets, see Quick Start for PA 10 Processors.pdf in the CodeWarrior PA10 installation for steps on creating a BareBoard AMP Core0 project for the QorIQ processor on this target board. In the "Debug Target Settings Page", of the procedure for creating a new project, uncheck the 'Download' option, and enable the 'Download SRAM' option, if available.
3. Select your CodeWarrior TAP or Gigabit TAP as your debug connection type. For CodeWarrior TAP, select “USB” or “Ethernet” as the connection medium.
4. Build the project.
5. Bring up the Target Tasks view: go to Window>Show View>Other>Debug>Target Task.

6. Import the Flash Profile:
   a. In the Target Tasks view, click on the Import button. A file-browser window will appear, showing the "Flash_Programmer" folder.
   b. Open the "Flash_Programmer" folder, then the folder associated with the processor family on this target board.
   c. Select the configuration file for the particular target and flash device to be programmed on this target board, and click OK to import it. This file will appear in the Target Tasks view.
7. In the board's Software Deployment Guide, locate the “Flash Bank Usage” section for the target board to be recovered.
   a. Identify the NOR/NAND/SPI flash memory map that applies to the flash to be programmed. For the following steps, if the target flash supports multiple banks, choose the starting addresses for ‘Bank0’ or ‘current bank’, as appropriate.
   b. Identify the starting address for the u-boot image.
   c. Identify the starting address for the RCW image (if applicable).
   d. Identify the starting address for the PPA image (if applicable).
   e. Identify the starting address for the ucode/microcode (if applicable).
   f. Identify the starting address for the dtb image.
   g. Identify the starting address for the RamDisk image.
   h. Identify the starting address for the Linux Kernel image.

      For example:

Images	**Size**	**vbank0**	**vbank4**
RCW + PBI	1MB	0x5_8000_0000	0x5_8400_0000
U-boot boot loader	1MB	0x5_8010_0000	0x5_8410_0000
U-boot Env	1MB	0x5_8020_0000	0x5_8420_0000
MC FW	4MB	0x5_8030_0000	0x5_8430_0000
MC DPL Blob	1MB	0x5_8070_0000	0x5_8470_0000
MC DPC Blob	1MB	0x5_8080_0000	0x5_8480_0000
Unused	4MB	0x5_8090_0000	0x5_8490_0000
Reserved	1MB	0x5_80F0_0000	0x5_84F0_0000
Cortina PHY FW	1MB	0x5_8100_0000	0x5_8500_0000
FIT Images (Linux, device tree, and rfs)	40MB	0x5_8110_0000	0x5_8510_0000
Unused	7MB	0x5_8390_0000	0x5_8790_0000
Primary Protected Application (PPA)	2MB	0x5_80a0_0000	0x5_84a0_0000
Secure Boot Headers	3MB	0x5_80c0_0000	0x5_84c0_0000

      For example:

Table 112. T4240QDS NOR flash

Binaries	Starting Address
U-Boot	0xEFF40000
RCW	0xE8000000
ucode	0xEFF00000
dtb	0xE8800000
RamDisk (rootfs)	0xE9300000
Linux Kernel (uImage)	0xE8020000

8. Configure Flash Programmer.
   a. Double-click on the file name that was imported with the flash profile, to bring up the Flash Programmer Task view.
   b. Click on 'Add Action'>'Program/Verify'.
   c. Set 'File Type' to "Binary".
   d. Click on 'File System' and navigate to the folder containing the u-boot binary image.
   e. Enable "Erase sectors before program".
   f. Enable "Apply address offset", and enter the starting address where this binary recovery image will be flashed (see the tables in the previous step for examples).
   g. (OPTIONAL) Enable 'Verify after program' to verify that the flash programming was successful.
   h. Repeat steps (starting with Click 'Add Action') above for each binary image file to be programmed into flash.
9. Execute Flash Programming.
   a. In the Target Tasks view, right-click on the imported filename and select the green Execute button to launch the programmer.
   b. If Execute is not green, the debugger is not running. The debugger must be running for this flash programmer to work.
   c. When finished flashing, terminate the debugger.
10. This is the end of the process. Now the boot loader, kernel and root file system are programmed to flash.
11. Reset or power-cycle the board and verify that u-boot appears in the board's serial terminal.

Chapter 6
DPAA2-specific Software

6.1 DPAA2 Software Overview

6.1.1 Introduction

This document provides an overview of the software and tools for the DPAA2 networking hardware that is provided on NXP SoCs such as LS2088A. These SoCs are called "DPAA2 SoCs" because they contain the hardware that is required to support the DPAA2 networking architecture. As will be described below, this hardware includes Queue Manager/Buffer Manager (QBMan), the Wire Rate I/O Processor (WRIOP), and the Management Complex (MC).

DPAA2 is an architecture in which some facilities (and thus the hardware that supports them) are optional. For this reason, this document may describe features that are not available on all DPAA2 SoCs.

Intended audience

This document is intended for software developers and architects who want to develop software for DPAA2 SoCs. The document assumes that users will be familiar with Linux-based development, computer networking, and also with the ARM™ architecture

Definitions and acronyms

- AIOP: Advanced I/O Processor hardware
- DCE: Decompression, Compression Engine hardware
- DPAA2: Datapath Acceleration Architecture, second version
- DPAA2 SoC: An SoC that contains the DPAA2 hardware
- GPP: General Purpose Processor
- MC: Management Complex
- NADK: Networking Application Development Kit
- PME: Pattern Matching Engine hardware
- QBMan: Queue Manager and Buffer Manager hardware
- SDK: Software Development Kit
- SEC: Security Engine hardware
- USDPAA: User Space Datapath Acceleration Architecture
- WRIOP: Wire Rate I/O Processor hardware

DPAA2 in the NXP SDK

NXP provides a Linux-based software development kit (SDK) for SoCs. The core of the SDK is an embedded-oriented Linux distribution containing components such as:

- U-Boot boot loader
- Linux kernel with networking support
- GNU tool chain for ARMv8™
- Large set of standard Linux user space packages including shells, initialization scripts, servers, etc.

- Yocto-based package management in an embedded-style source-based Linux distribution

NXP supports and builds upon standard Linux with drivers and additional packages and capabilities including support for the DPAA2 networking hardware such as:

- Management complex firmware for the DPAA2 architecture. DPAA2 is a networking peripheral subsystem architecture and will be discussed at length in later sections.
- Restool: a DPAA2 object management tool
- A DPAA2 Linux Ethernet driver
- Linux kernel support for treating DPAA2 containers as plug-and-play busses with VFIO support
- Integrated kernel-based control of DPAA2 L2 switch objects
- Kernel support for DPAA2 acceleration objects including cryptographic offload
- And more

6.1.2 DPAA2 Hardware

6.1.2.1 Introduction

This section introduces the DPAA2 hardware components and explains their relationship to the DPAA hardware found on previous NXP SoCs. Finally, it shows the DPAA2 hardware blocks in the context of a specific SoC, the LS2088A.

Note that the DPAA2 hardware is configured via DPAA2 objects as will be described below. This section on hardware provides background information to give context to the discussion of the DPAA2 objects. Most developers will deal with the DPAA2 objects and not directly with all aspects of the DPAA2 hardware blocks.

6.1.2.2 DPAA2 hardware

The DPAA2 hardware provides network interfaces, hardware-based queuing, layer 2 switching, more general switching, networking-related accelerators, and also memory dedicated to packet processing.

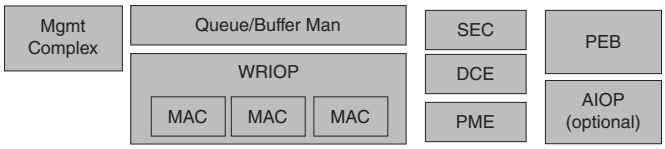


Figure 7. DPAA2 hardware components

See DPAA2 hardware components. The DPAA2 hardware contains the following components:

Management Complex (MC)

The DPAA2 hardware is abstracted by DPAA2 objects with the help of the Management Complex. This means that users need not study the details of the DPAA2 hardware blocks in order to develop drivers for or use DPAA2 capabilities. This software and solution oriented focus is one of the key differences between the first DPAA and DPAA2.

Queue and Buffer Manager (QBMan)

QBMan provides hardware-based buffer and queue management.

WRIOP

WRIOP provides hardware that serves as the basis for network interfaces. It includes Ethernet MACs, packet header key generators, parsers, table look up units, and an interface to the buffer and queue managers.

Accelerators (optional)

Accelerators that interface to QBMan are a key part of DPAA2. They include a cryptographic and security accelerator (SEC), a pattern matching accelerator (PME), a data compression/decompression accelerator (DCE), and a generic DMA engine. The set of accelerators may vary from SoC to SoC and new types of accelerators may be added.

PEB (optional)

PEB is a memory devoted to high-performance packet processing. It can be used to store in-flight packets and other items.

AIOP (optional)

AIOP is a fully programmable multicore engine with tightly coupled hardware accelerators that is specialized for efficient packet processing. It uses techniques somewhat similar to hardware multithreading to provide multiple "tasks" per core. The hardware supports efficient task switching to hide latencies associated with using accelerators and other hardware. The AIOP supports C language programming. It is optional in the DPAA2 architecture and thus is not available on all DPAA2 SoCs.

DPAA2 versus DPAA

DPAA2 is the latest generation of the Datapath Acceleration Architecture (DPAA) hardware. It is an evolution of the DPAA present in previous SoCs.

DPAA2 changes relative to DPAA include:

- DPAA2 contains a hardware block called the Management Complex. It facilitates and simplifies hardware resource allocation and hardware configuration.
- The hardware buffer and queue managers (QMan and BMan) are integrated into a single hardware block called QBMan.
- DPAA2 session context can be maintained per frame, rather than per frame queue, which allows multiple accelerator sessions to share a single frame queue pair. This single frame queue pair then reduces the number of frame queues needed, making session establishment more efficient because frame queues do not need to be initialized per session.
- Software portals are enhanced to make it easier and more efficient for General Purpose Processing (GPP) core software to share them.
- WRIOP in DPAA2 replaces FMan as the hardware block that provides Ethernet interfaces. WRIOP is designed to be more partitionable, in that it allows GPP software to more independently manage separate network interfaces.
- WRIOP and QBMan contain new features that support autonomous L2 switching functionality:
  - WRIOP: L2 address learning and forwarding unit.
  - QBMan: packet replication facility.
- WRIOP does not contain a generic programmable engine like the one present in FMan, and instead DPAA2 has a new hardware block called AIOP that is specifically designed to perform this function.

6.1.2.3 LS2088A block diagram

The LS2088A is an ARMv8-A 64-bit SoC. It contains eight ARM Cortex-A57 cores and numerous peripherals. The LS2088A is an example of a DPAA2 SoC because it contains the required DPAA2 hardware blocks: WRIOP, QBMan, and MC.

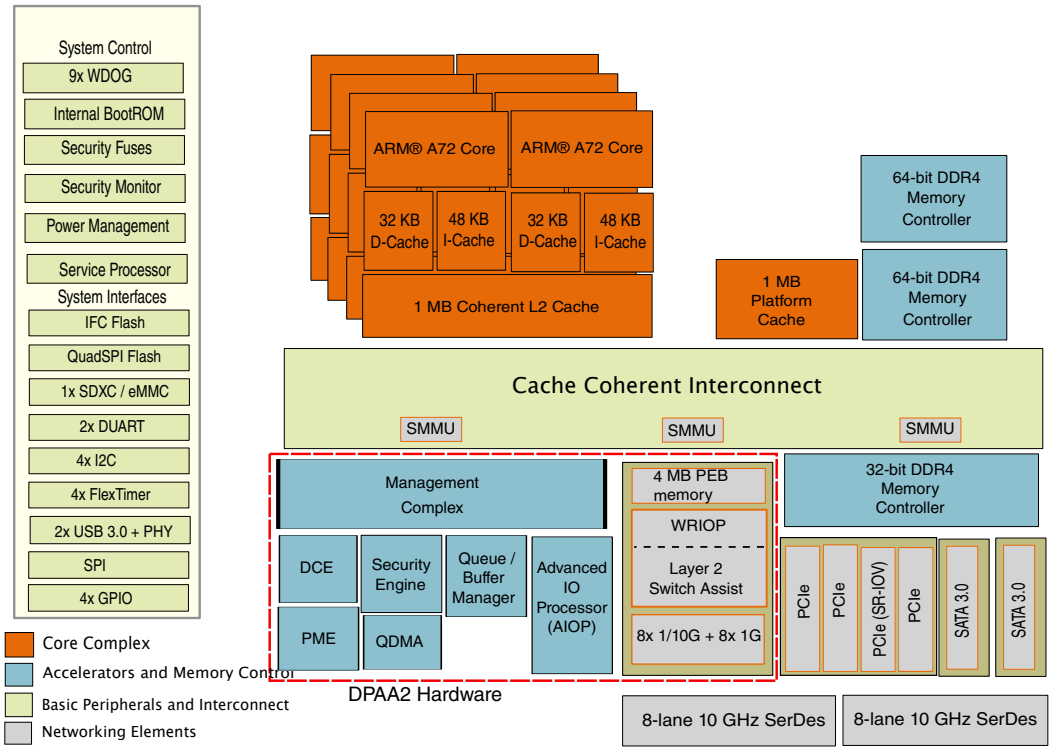


Figure 8. LS2088A SoC

The LS2088A contains standard-ARM components in addition to the cores, such as:

- ARM generic timer
- GIC-500 interrupt controller
- MMU-500 System Memory Management Unit (I/O MMU)

It also contains conventional hardware blocks including:

- DDR controllers
- Flash controller
- SDxC/eMMC controller
- USB controller
- PCIe controller
- SATA controller
- Other blocks visible in the diagram.

Finally, the following DPAA2 components are highlighted in the figure:

- QMan/BMan: hardware queue and buffer management
- WRIOP: Ethernet interfaces
- Management complex: DPAA2 objects and their management
- Accelerators: SEC, PME, and DCE

6.1.3 DPAA2 Linux Software

6.1.3.1 Introduction

This section provides a high-level summary of the most important DPAA2 software associated with the Linux operating system.

6.1.3.2 Linux and DPAA2

This section summarizes major Linux DPAA2 software. See Linux DPAA2 software which shows the software in relation to some standard Linux software.

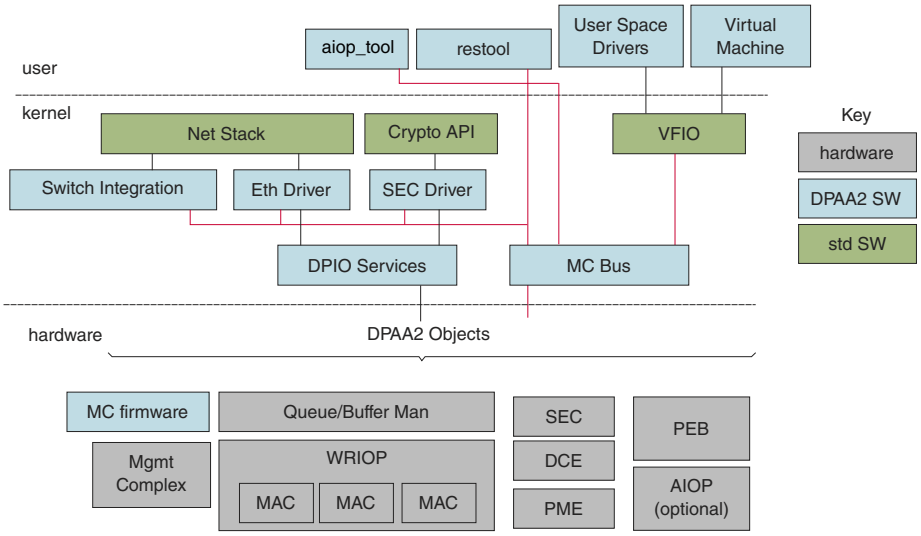


Figure 9. Linux DPAA2 software

Ethernet Driver

DPAA2 software includes a conventional Ethernet driver for use by the Linux network stack. This driver is controlled via standard Linux means such as the "ifconfig" or "ip" commands and also "ethtool". It operates in a manner that will be familiar to Linux users. Drivers in DPAA2 manage DPAA2 "objects" as will be described below. These objects are best regarded as hardware. They are formed from hardware resources.

DPIO Services

DPAA2 drivers such as the Ethernet driver in the Linux kernel use the DPIO services Linux component to do I/O. The DPIO services layer manages the kernel's DPIO objects. DPIO objects contain DPAA2 software portals (which are hardware components). The software portals can be shared by multiple higher-level drivers.

DPAA2 Objects and Management Complex (MC) Firmware

The DPAA2 hardware is presented to software in terms of DPAA2 objects that are realized by means of firmware running on the Management Complex. This will be explained in depth in DPAA2 Networking Subsystem Deeper Dive on page 416 and also immediately below.

MC Bus and Restool

The DPAA2 objects appear as devices on a special software-defined bus called the MC bus. Linux has a driver for this bus (and interactions with VFIO). This software is analogous to PCIe bus software. Like PCIe, the MC bus supports plug and play.

The "restool" utility is a Linux user space command that allows DPAA2 objects to be managed: created, destroyed, queried for status, etc.

SEC Driver

The SEC driver provides the standard Linux kernel cryptographic API but implemented by the SEC hardware by means of a special DPAA2 object. Other accelerators can be handled in the same way, but Linux tends to not provide standard (hardware-independent) kernel-level APIs for them so they are not discussed here.

Switch Integration

Finally, DPAA2 objects exist that perform L2 and more general network switching. These hardware elements can be configured using standard Linux mechanisms such as "bridge". As will be discussed later, there are two types of switch-related DPAA2 objects: DPSW and DPDMUX. There is kernel-based management support for both.

AIOP Tool

AIOP Tool (aiop_tool) is a user space command line utility that allows programs (images) to be loaded onto the AIOP and started. It also supports stopping and resetting the AIOP. Of course, aiop_tool is used only on DPAA2 SoCs that have an AIOP.

It is also possible for user space programs to manage these aspects of AIOP via programmatic means. The aiop_tool utility is most useful during development of AIOP software and for AIOP applications that happen not to be tightly integrated with control software running in user space on the GPP cores.

From the point of view of software running on the GPP cores, AIOP programs may be thought of as firmware that defines the functionality that the AIOP will provide to an overall system.

.

6.1.3.3 DPAA2, Management Complex, and drivers

DPAA2 is the architecture that describes network interfaces and other networking services for an SoC with DPAA2 hardware. It is discussed in depth in DPAA2 Networking Subsystem Deeper Dive on page 416. For now, think of DPAA2 as hardware for networking that is presented in terms of DPAA2 objects. The objects provide specific high-level features or services such as network interfaces or L2 switches.

The objects are managed by means of firmware running on a hardware block called the Management Complex. Software on general purpose cores must load firmware onto the Management Complex before networking can be done using DPAA2 hardware.

Normally, the MC firmware is loaded early in the boot process so that boot loaders can make use of DPAA2 objects and peform networking operations such as network-based booting.

Since the objects represent hardware, they require driver software on general purpose cores. NXP provides drivers for U-Boot and standard Linux and thus both support Ethernet networking out of the box. For example, one can use Linux networking without delving into the details of DPAA2 and its objects just as one can use Linux networking via a PCIe Ethernet card (whose manufacturer provides a driver) without delving into the design of the card.

DPAA2 and its objects are fully documented so it is possible to write drivers for other operating systems, applications, or boot loaders, e.g. ODP, DPDK, UEFI firmware, etc. Many of these drivers exist or are roadmap items.

6.1.3.4 DPAA2 and plug-and-play

There is another analogy between DPAA2 objects and PCIe devices. PCIe devices appear to operating systems as plug-and-play devices on a bus. The operating system can scan the bus to discover and identify the devices on it. It can then use the device identities to associate drivers with devices and bring them into service.

DPAA2 objects work in a similar way. They are placed into datapath containers (DPRC) that can be scanned in an analogous manner. Then objects are associated with drivers and placed into service.

The Linux kernel is provided with a container with its DPAA2 objects. Containers can also be provided to other software including virtual machines and even arbitrary user space processes. This is how the hardware that objects encapsulate can be directly assigned to virtual machines and user space processes. This allows them highly efficient access to hardware but in a secure fashion due to the involvement of the SoC IO-MMU.

This, also, is analogous to PCIe devices in standard Linux; DPAA2 objects can be directly assigned to virtual machines and user space processes using a standard Linux architecture called VFIO which allows devices to be mapped into the address space of user space processes and also enables IO-MMU configuration to constrain the memory to which devices can read and write data via their DMA engines.

Like PCIe devices, DPAA2 objects are also mapped using VFIO. NXP supplies the extensions to VFIO in Linux that makes this possible.

6.1.3.5 Datapath layout files and restool

As mentioned elsewhere, DPAA2 containers are like PCIe busses in that they can be scanned for objects/devices. But containers and PCIe busses are populated very differently. PCIe busses are populated physically, e.g. by plugging a card into a slot.

Objects are encapsulations of DPAA2 hardware resources that must be created via management complex firmware and then assigned to a container. There are several ways to do this:

1. the datapath layout file
2. restool
3. Management Complex commands

Datapath layout (DPL) file

Containers and objects can be defined statically in a file called a datapath layout file (DPL) that is passed to the management complex when it is initially booted. The DPL can specify containers, objects, and connections between objects. When an OS such as Linux boots, it will discover the populated containers.

restool

The utility called “restool” is a NXP-created Linux user space command that allows inspection and dynamic management of containers and objects. With it, one can

- Display the current set of containers and objects
- Create and destroy containers
- Create and destroy objects
- Assign objects to containers
- Create links among objects

One can use a sequence of restool command invocations to create the same container and object state that a DPL might specify. The difference is that restool is dynamic.

Management Complex commands

Finally, objects and containers can be manipulated by software running on general purpose cores by sending commands to the Management Complex. This is, in fact, what restool does. Command line arguments to restool define an operation. The restool utility simply forms a command and passes it to the Management Complex. Other drives can also do this.

6.1.4 DPAA2 Networking Subsystem Deeper Dive

This section provides additional detail on the DPAA2 architecture and the DPAA2 object services paradigm.

This paradigm simplifies using the DPAA2 hardware IP blocks through abstraction and encapsulation. DPAA2 objects are objects in the sense that they:

- Encapsulate specific abstract functionality, e.g. L2 switching.

- Are composed of allocated hardware sub-components of the DPAA2 hardware peripherals, and then mostly abstract their functionality .
- Present functionality in terms of specific attributes and methods, meaning operations on the objects.

NOTE

DPAA2 objects are not associated with object-oriented programming languages, instead they are collections of hardware resources allocated for a specific purpose. General purpose processing (GPP) core software can configure objects by sending them commands expressed in terms of hardware-level descriptors. GPP software can also include C language functions that prepare and interpret the descriptors. No use of object-oriented programming languages is required. For the most part, Linux drivers are written in C as usual

This section:

- Presents the DPAA2 object model at a concept level and describes how objects are created, destroyed, conveyed, configured, and used
- Lists the objects types and their purposes
- Outlines how the Management Complex implements and provides the objects
- Explains what software components use the various DPAA2 object types, and how they use them. The users are often application software running on general purpose processors (cores) or on the optional AIOP.

Driver-level software on GPPs works with the abstracted objects, rather than directly with the hardware. For example, the GPP software deals with L2 switch and network interface objects rather than WRIOPs .

DPAA2 objects express and abstract the DPAA2 hardware into software-managed objects that are:

- Application-oriented in terminology and use, rather than hardware-oriented
- Based on concepts that are generally familiar to programmers and system architects
- Simpler than direct management of the hardware
- Indicate the architectural intent of the hardware blocks

DPAA2 object services are provided by software that runs as firmware on a DPAA2 hardware block called the Management Complex. Users do not need to program the Management Complex in order to use the Network Object Services; they simply use the NXP-supplied firmware. This firmware runs on the Management Complex instead of a general purpose core in order to simplify the integration of the NXP software with customer software. DPAA2 object concept below shows at a concept level how the Management Complex provides objects that perform specific services; the objects have attributes and interfaces that appear as hardware.

Figure 10. DPAA2 object concept

6.1.4.1 DPAA2 hardware abstraction example

This section introduces the DPAA2 objects and the abstractions they provide by means of an example. Example scenario shows a system constructed using the DPAA2 hardware on a DPAA2 SoC such as the LS2088A. The goal is to run two KVM virtual machines (VMs) on the SoC. The two virtual machines each have a hardware network interface that they can directly access (i.e. a dedicated interface) connected to a DPAA2 L2 switch. These VMs can communicate with each other via the L2 switch, and they can communicate externally via the MAC on the L2 switch. So, the L2 switch has three ports, one for an off-SoC connection (connected to a MAC), and two for the VMs.

In addition, there are two network interfaces with MAC addresses for off-SoC communication that are used by the host Linux. The host Linux instance and the virtual machines all run on the Cortex-A72 cores on the LS2088A. In this example, each network interface is associated with an Ethernet driver working with Linux.

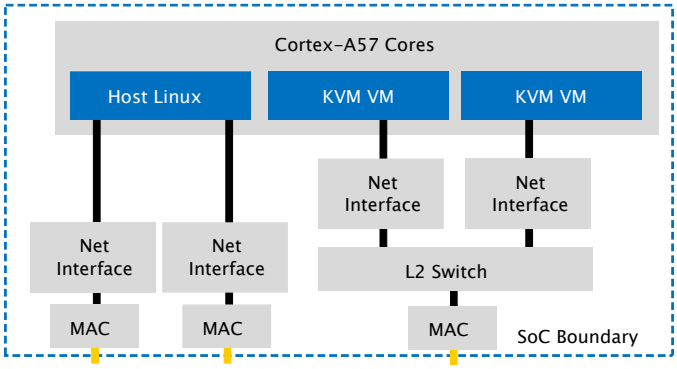


Figure 11. Example scenario

DPAA2 hardware shows the DPAA2 hardware blocks. This figure bears little resemblance to Example scenario. It provides little guidance to how the example scenario could be realized because the hardware blocks are conceptually distant from a natural statement of what is desired in the example. The DPAA2 objects are much closer, as will be seen below.

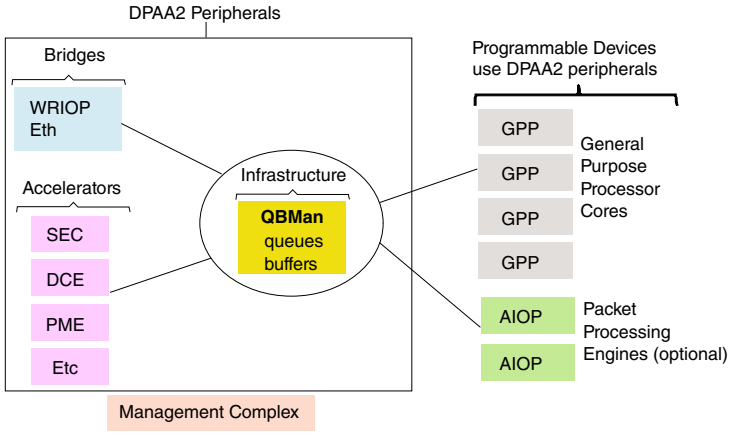


Figure 12. DPAA2 hardware

Example scenario based on DPAA2 objects shows how the example can be realized using the DPAA2 object abstractions of the DPAA2 hardware; this figure is much closer to the goal expressed in Example scenario and its components are described below:

- The host Linux is shown in more detail on the left. The network stack and two instances of the Ethernet drivers appear in the figure above the hardware boundary. Also, the figure shows the stacks and drivers for the two virtual machines.
- The DPAA2 objects appear below the hardware boundary
- The DPNI (Datapath Network Interface) objects correspond directly to the network interfaces in Example scenario. The DPSW (Datapath Switch) object corresponds to the L2 switch.
- The DPMAC (Datapath MAC) objects represent Ethernet MACs within WRIOP. These are hardware components that connect to PHY hardware, and provide Ethernet physical layer termination, i.e. Ethernet connections to the SoC.
- The DPIO (Datapath I/O) objects include QBMan software portals, and they allow GPP core software to read and write packets from the DPNIs. DPIOs are described in more detail later in this document.

See Object summary on page 421 for a summary of the DPAA2 objects.

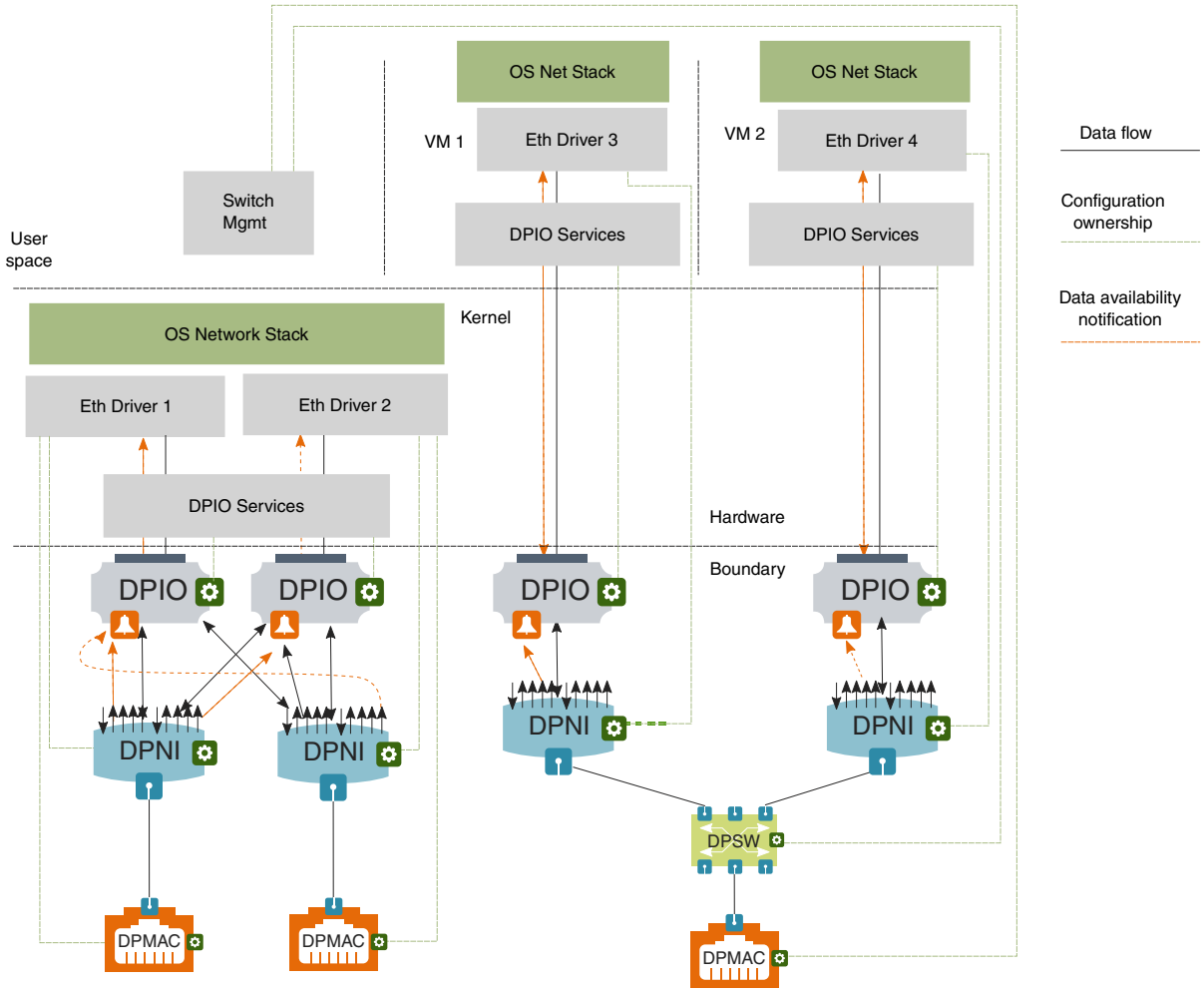


Figure 13. Example scenario based on DPAA2 objects

Objects are partitioned among software owners

Software management of DPAA2 objects is distributed. Software components that use a particular set of objects independently manage the objects in their set. The green boxes on the object icons in Example scenario based on DPAA2 objects represent management interfaces, and the green dashed lines show what software component owns the management of each object. For example, the DPSW is shown as managed by switch management software running on the general purpose processing cores.

Objects can be directly assigned

The virtual machines directly access and manage the objects their software uses, and they do this with minimal host kernel involvement; this enhances efficiency while preserving access isolation. In the figure, the virtual machines have directly assigned hardware-based network interfaces.

DPNI objects provide network interfaces

DPNI objects interact with drivers to allow software to send and receive network frames, usually Ethernet frames. DPNIs are central to DPAA2's concept of network interfaces, but they do not act alone. In general, network drivers manage several

objects as part of managing network interfaces. DPNI ingress shows a high-level outline of DPNI ingress frame processing, and the following steps give insight into how objects work together.

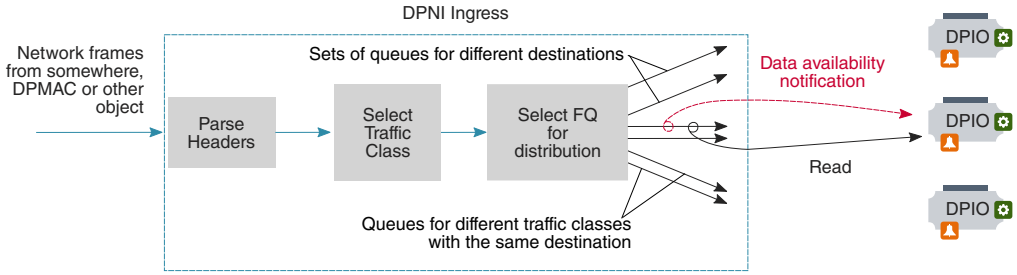


Figure 14. DPNI ingress

1. A frame arrives at DPNI from another object, a MAC (DPMAC), a switch (DPSW) or other object.
2. DPNI parses the packet to locate the header from which lookup keys can be generated.
3. A lookup selects a traffic class (priority) for the frame; this priority causes a specific set of queues (implemented as QMan frame queues) to be selected.
4. DPNI must select a destination for the frame, using either another lookup or an RSS-style hashing operation; this lookup causes a specific queue within the previously selected set to be selected.
5. The frame is enqueued onto the queue, and the queue represents the destination indirectly. At this point, DPIO objects enter the process.
6. Every queue is configured to deliver data availability notifications to a specific DPIO, and these notifications tell the driver software using the DPIO that one or more frames are available to read from a specific queue.
7. Driver software responds by using a DPIO (actually any of its DPIOs) to read a burst of one or more frames from the queue.

Egress is simpler. The driver software uses a DPIO to enqueue a frame to a specific egress queue within DPNI; the queue is selected based on the desired traffic class.

Multiple DPIOs provide parallelism

It is common to assign queues in network interfaces to specific cores, and then to distribute the traffic between them using techniques like RSS or explicit flow steering. DPAA2 supports this process by using multiple DPIOs. See DPIO parallelism for an example involving a single network interface and two cores.

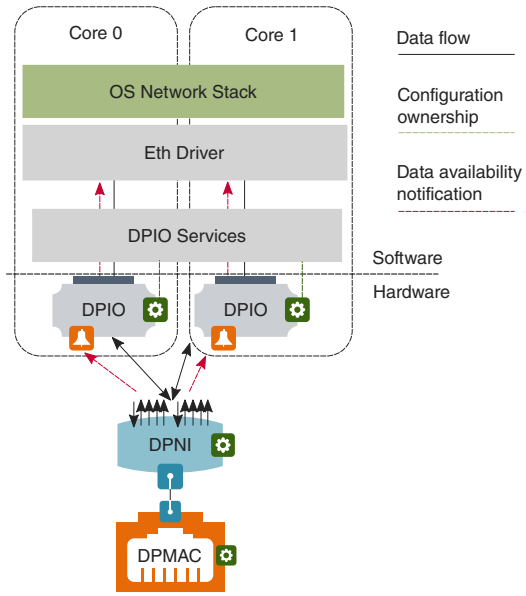


Figure 15. DPIO parallelism

The DPNI is configured so that each of its egress queues send its data availability notifications to one DPIO or another in a balanced way. A core receives an interrupt from its DPIO telling it to read a data availability notification, and then it then uses its DPIO to read a burst of one or more frames. In Linux terms, it starts a NAPI burst.

DPIO services

Notice in Example scenario based on DPAA2 objects that the host operating system on the left has two network interfaces. It has two DPIOs also, but either DPIO can be used for I/O to either of the interfaces. DPIOs are designed to be shared across network interfaces that belong to the same software component, such as the Linux kernel. For this reason, the Linux kernel contains a software layer called DPIO Services that facilitates driver instances performing I/O from a resource that might be shared across a network interface, and also might be shared across cores or software threads. Giving more DPIOs to the DPIO Services layer can increase performance, and using the same DPIO on a core for more than one network interface need not decrease performance because each core is physically able to do only one thing at a time.

6.1.4.1.1 Object summary

This section summaries the DPAA2 objects and shows a standard icon for each used in the illustrations that follow. See DPAA2 object summary and icons.

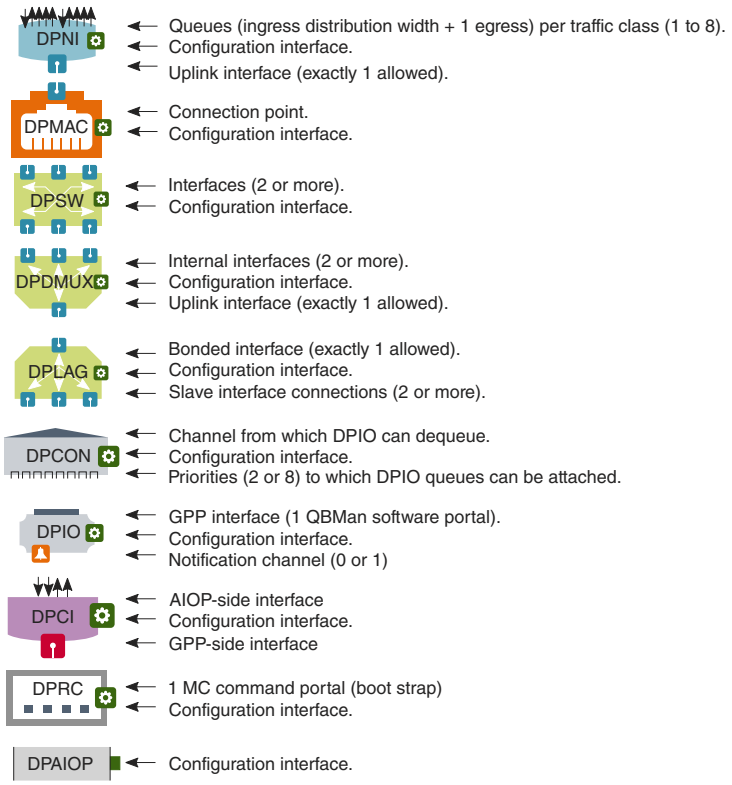


Figure 16. DPAA2 object summary and icons

DPNI

A DPNI object is the key to network interfaces. On ingress, it receives frames from a DPMAC or another object such as a DPSW, parses headers, determines the frame's traffic class, and enqueues the frame onto a frame queue selected based on the traffic class and other header values. This supports both hash-based distribution of frames to multiple cores, and also direct flow steering of frames to specific cores.

DPNI can generate a per-queue data availability notification when a frame is enqueued. On egress, the DPNI dequeues frames from frame queues and transmits them to an external port using a DPMAC, or to another DPAA2 object such as a DPSW.

DPMAC

The DPMAC object represents an Ethernet MAC, a hardware device that connects to a PHY and allows physical transmission and reception of Ethernet frames.

DPSW

The DPSW object provides the functionality of a general layer 2 switch. It receives packets on one port and sends them on another. It can also send packets out on multiple ports for the purposes of broadcast, multi-cast, or mirroring.

DPDMUX

The DPDMUX is another type of switch. It differs from a DPSW in several ways. A DPDMUX may have only a single uplink port. Also, it can be programmed to direct packets based on header values above layer 2.

DPLAG

The DPLAG object provides link aggregation. It combines two or more uplinks into a single downlink.

DPCON

The DPCON object allows multiple DPNIs to be aggregated into a single device that appears to a GPP core or AIOP software as single interface that carries frames from multiple DPNIs; it combines two or more network interfaces into one. It provides a hardware-based scheduling off load because the hardware selects the order based on the priority in which frames from the multiple DPNIs are provided to software on GPP cores or AIOP.

DPCON is also useful for software that polls for input frames; it allows a single interface to be polled instead of multiple interfaces.

DPCON objects are also used by Linux Ethernet drivers for priority-based frame delivery.

DPIO

General purpose processing core software uses a DPIO object to perform hardware queuing operations, such as enqueue and dequeue, and hardware buffer management operations, such as acquire and release. It also allows data availability notifications to be received. DPIOs can generate interrupts. The DPIO object is unusual in that GPP core software is expected to directly access portions of the DPIO's hardware (QBMan software portals) for run time operations, in addition to supporting configuration operations from the management complex.

Note that AIOP software does not rely on DPIO objects; they are used only by software on the general purpose processing cores.

DPBP

The Datapath Buffer Pool object represents a QBMan buffer pool. It is used mainly as a resource by network drivers, but it is an active entity because it can send buffer pool depletion notifications to GPP core software.

DPCI

The Datapath Communication Interface provides general purpose processing core software with a transport mechanism typically for control and configuration command interfaces to AIOP applications. The AIOP service layer implements the AIOP-side of the transport, but the commands are application-specific. Note that AIOP is optional in DPAA2 and is not present on some SoCs.

DPRC

The DPRC object allows the Management Complex to track sets of objects in use by the same software component. The objects in the set are said to be in the same container. It also facilitates the assignment of sets of objects to specific software components, such as a virtual machine or a user space application using user space drivers. The software component can query containers in order to discover objects at run time, and this enables plug-and-play drivers that interface to objects.

Some objects include DMA-capable hardware. All objects in the same DPRC share a common ICID, and a common set of IO-MMU mappings. A number of key features of DPRCs include:

- **Direct access.** All the objects and resources in a container are private to the container, and software components get direct access to the registers (as abstracted by the Management Complex) of the hardware objects.
- **Dynamic discovery.** A software context that is given a DPRC can dynamically discover the objects and resources placed in the container using MC commands.
- **Hot plug/unplug.** Objects can be dynamically plugged and unplugged into DPRCs.
- **Security.** A software context can only see the objects in its DPRC, and cannot affect other containers or the proper operation of other software contexts. DMA transactions from MC objects are isolated using the system IOM-MU.

DPMCP

The DPMCP object represents a Management Complex command portal and is used by drivers to send commands to manage objects.

DPAIOP

DPAIOP is a configuration object that aids in loading programs onto the AIOP, running the AIOP, resetting the AIOP, and receiving status, error, and log information from AIOP programs. Note that the AIOP is optional in DPAA2 and is not present on some SoCs.

Objects for accelerators

There are also objects associated with accelerators such as SEC, PME, and DCE. These objects provide software with interfaces to the accelerator hardware. For this reason, the accelerator interface objects end in "I".

- DPSECI - SEC (security/cryptographic coprocessor) interface.
- DPDCEI - DCE (data compression engine) interface.
- DPDMAI - DMA engine interface.

Software uses queues associated with an object to send a buffer to an accelerator for processing and to receive the result.

New types of objects

NXP will create new types of objects over time to address new needs and use cases as they arise.

6.1.4.2 Management Complex: How DPAA2 objects are created and managed

This section outlines how the Management Complex creates and manages DPAA2 objects.

The best way to think of DPAA2 hardware, in particular WRIOP and QBMan, is that it provides many low-level resources ranging from Ethernet MACs to look up tables to frame queues and so on. Software's mission is to assemble the right set of these low-level resources, and configure them collectively to achieve a goal.

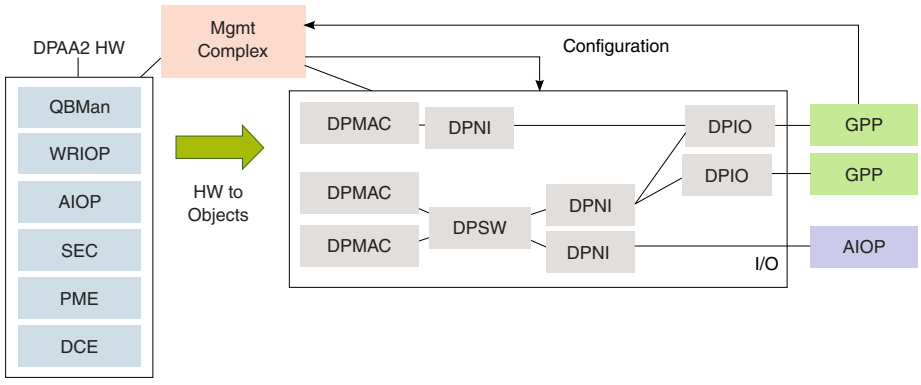


Figure 17. Management Complex creates objects from hardware sub-components

Think of the low-level resources as “atom resources” because they are always allocated as a unit. DPAA2 objects are then “composite resources,” or collections of atom resources that are then configured to achieve a common goal, like being an L2 switch as shown in Realizing an L2 switch.

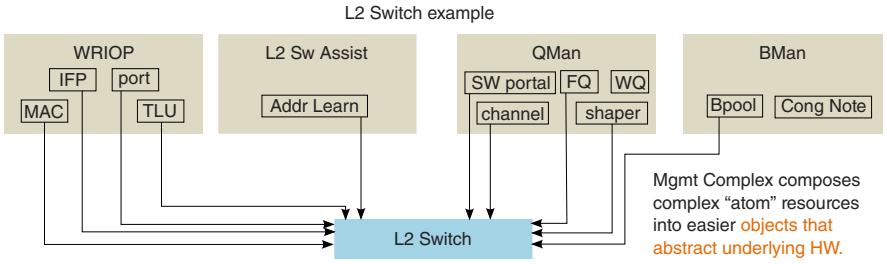


Figure 18. Realizing an L2 switch

The creation method for a DPAA2 object involves allocating the necessary atom resources and configuring them enough to place the object in an initial idle state. Object methods and other interfaces then allow it to be further configured and used. For example, forming an L2 switch from DPAA2 atom resources is quite complex. The NXP firmware running on the Management Complex implements the methods necessary, and hides this complexity from GPP (and AIOP) developers.

Continuing the example, an L2 switch object can also be shutdown and disassembled by its methods. Its atom-resources are then placed back into the pools of atom resources that the Management Complex firmware manages.

Hardware directly visible to software

Clearly, DPAA2 provides abstractions. The objects are best thought of as being hardware, and most actually are collections or encapsulations of hardware resources that are allocated and configured to achieve a higher-level and more abstract purpose than would be clear from a direct view of the hardware resources. An example of an abstract purpose is “be an L2 switch” (DPSW).

It can be helpful to focus on exactly what is visible to driver-level software running on the general purpose cores and AIOP, especially since what is visible is a mixture of direct access to hardware and indirect access to hardware via abstractions. This discussion will be biased towards the view of objects from drivers running on general purpose processing cores (such as in U-Boot and Linux).

Also the discussion will avoid details of individual objects since this is an overview with the purpose of clarifying objects in general.

DPAA2 visibility boundary describes in one diagram what is directly visible to the driver layer software.

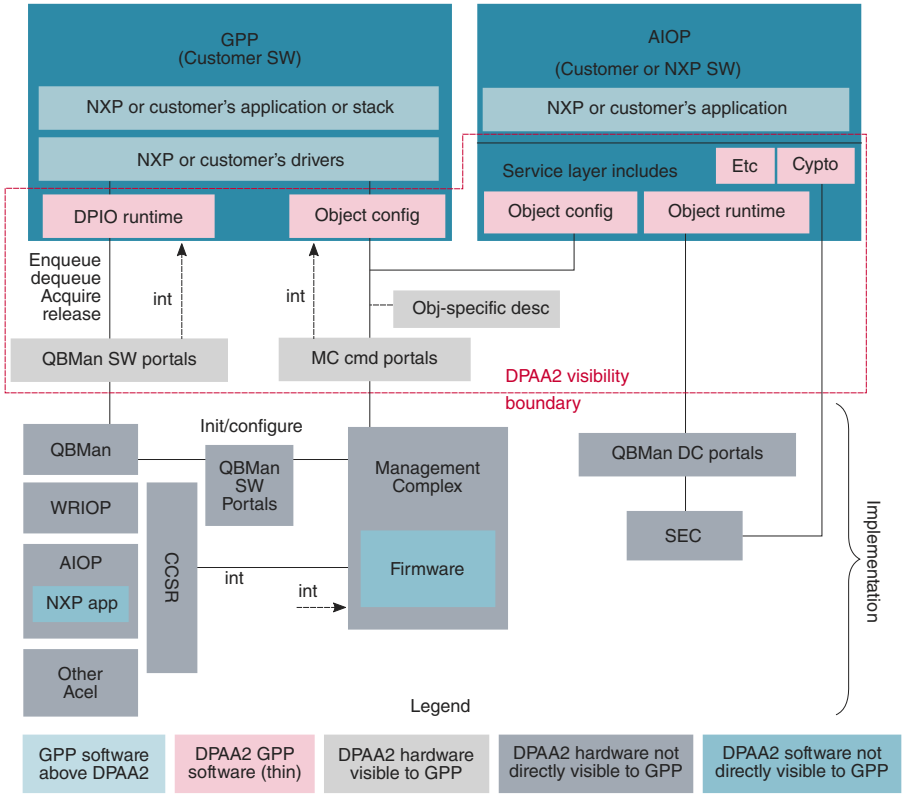


Figure 19. DPAA2 visibility boundary (AIOP not present on all DPAA2 SoCs)

There is quite a lot in the figure above, so it is best to break it down. What driver level software can see and do is dictated by its function.

This begins with the Management Complex (MC) itself. The discussion below will focus on the services that the MC provides to other software in the system. There will be no discussion of MC firmware's internal design.

See Management complex visibility in DPAA2. The first step is that general purpose processing core software (usually a boot loader) must load the opaque firmware image onto the Management Complex and then start it running. This involves direct access to portions of the Management Complex hardware: registers defining the location of the Management Complex's portion of DDR, image location, address translation, and run state control.

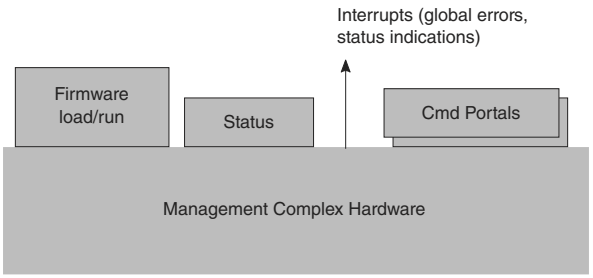


Figure 20. Management Complex visibility in DPAA2

The driver software also requires visibility to global status, particularly to status for global errors. Changes in the state of this status can be signaled by interrupts to the general purpose processing cores so the Management Complex can produce these interrupts.

Finally, the Management Complex exists to serve its masters, the general purpose processor core (and AIOP when present) software that “owns” objects, i.e. has been allocated access rights to them via container ownership and hierarchy. The service is provided by responding to commands so driver software needs a way to deliver commands to the Management Complex. In addition, this process must be secure in that the Management Complex must know, in a way that cannot be spoofed, an ID of the software sending the command. This is to allow the Management Complex to enforce object access rights.

Driver software delivers commands to the Management Complex via hardware called Management Complex command portals. SoC hardware provides significant numbers (at least 10s) of these portals because:

- They can be directly assigned to multiple different drivers, all of which independently use the Management Complex’s services. If they each have their own command portal, they do not have to coordinate with each other.
- Each independent driver instance has its own ID (ICID) that is securely associated with the command portal to prevent spoofing. This prevents a driver from being able to access for configuration an object that it does not "own".

To send a command to the Management Complex, driver software creates a descriptor and enqueues a pointer to it to the command portal.

Next consider objects. See DPAA2 objects.

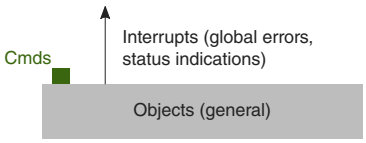


Figure 21. DPAA2 objects

Objects are created either via the DPL file or driver software sending a command to the Management Complex instructing it to create an object (as in restool). The Management Complex supplies a globally unique ID for the new object.

Object command interfaces are abstractions. There is no hardware that directly represents object command portals. Objects are usually hardware, but in most cases that hardware does not directly expose a hardware-level programming model to driver software. Instead, driver software configures objects via an indirect mechanism; it sends a command to the Management Complex. The command is a descriptor that includes the ID of the object as well as the definition of the operation to be performed.

The Management Complex automatically gets the ID of the requestor when it reads the command. The command portal securely adds it. The Management Complex then checks that the requestor is authorized to configure the object and, if so, performs the configuration on behalf of the requestor.

So, object configuration is a visible part of DPAA2, but the configuration of the individual hardware subcomponents that make up an object is not.

The fundamental programming model for object configuration is the commands that can be sent to the Management Complex to configure the object. Each object type has a different purpose so each object type’s configuration programming model is defined by the descriptor set that describes the commands to configure the particular type of object.

NXP also provides C callable APIs that basically allocate and populate descriptors and pass them as commands to the Management Complex. The APIs bear a close relationship to the more fundamental descriptors.

Many object types have nothing but a configuration space, but this is not always true. Some objects also provide I/O interfaces. The DPIO object is a prime example. See DPIO object and I/O interfaces.

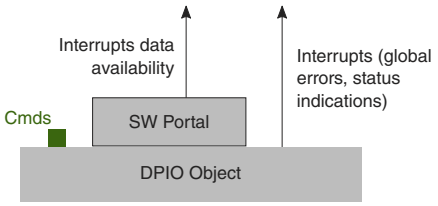


Figure 22. DPIO object and I/O interfaces

As has been stated before, DPAA2 objects are usually opaque bundles of hardware sub-resources allocated and configured to achieve a more abstract purpose. A DPIO object includes a hardware sub-resource called a QBMan software portal but this hardware is not opaque to the driver software running on the general purpose processing cores. The reason is performance. Software portals are the hardware mechanism for actually doing I/O with DPAA2 peripherals so driver software must directly access them. There are also data availability interrupts associated with DPIOs. These indicate availability of data to read using the software portals.

NOTE

Software portals actually support more than I/O (enqueue onto queues and dequeue from them). They also support commands. The simplest examples are buffer acquires and releases. Without going into full detail, software portals actually support commands that require privilege (example: initialize a frame queue) and commands that do not (example: acquire a buffer). Driver software on general purpose processing cores (and AIOP) uses only the unprivileged commands. The privileged commands are not part of the visible architecture. They are used only by Management Complex firmware.

In summary, the visible architecture includes both hardware and abstractions as follows:

- Management Complex hardware associated with loading and running images
- Management Complex hardware associated with accessing global status
- Management Complex global interrupt
- Management Complex hardware command portals
- Objects themselves (abstraction):
  - Object configuration interface and command set as defined by descriptors (abstraction)
  - Object error interrupts
  - Some objects (like DPIO) also have additional interfaces that are hardware directly accessed by driver software. DPIO's QBMan software portals are an example. They can produce interrupts.

6.1.4.2.1 Object creation, the datapath layout file, and restool

DPAA2 objects can be created in multiple ways. First, they can be specified in a Datapath Layout (DPL) file that the Management Complex reads and applies before Linux boots. This file contains the specific list of objects that are to be automatically created as the system initializes.

DPAA2 objects also can be created and destroyed dynamically by sending commands to the Management Complex through its command portals via a kernel driver. For Linux, a user space command line tool called "restool" uses this interface to allow interactive and dynamic creation of objects. It also allows destruction and some additional configurations to be done.

Restool also shows information about objects and what they are connected to.

6.1.4.2.2 DPRC objects, plug and play, and the fsl-mc Linux "bus"

As mentioned previously, it is common for a GPP software component to manage multiple objects. The DPIO parallelism diagram shows a simple example of the Linux kernel managing a set of objects to provide a pair of network interfaces. The

DPRC (Datapath Resource Container) is a special object that serves to organize other objects, and also the hardware sub-components from which objects can be dynamically created; the hardware sub-components include frame queues, channels, buffer pools, etc. Containers can be created and filled with objects and resources and then passed to the software component, such as a virtual machine, that will use them.

The software that was assigned a DPRC can enumerate the objects inside it; this is a form of dynamic hardware discovery that relates to plug-and-play. For example, an operating system can scan a DPRC and associate all DPNI objects found within with an Ethernet driver that will use them to form network interfaces. The Ethernet driver then uses a dynamic allocator within the kernel to aquire other objects such as DPBPs that it needs to operate.

The device discovery analogy is strong enough that Linux exposes DPRCs assigned to it as a bus in sysfs-- much like physical buses like PCI. The same sysfs mechanism that allow a physical PCI device to be assigned (bound) to virtual machines are also used to assign containers to virtual machines. Objects can even be dynamically added and removed from DPRCs. This is analogous to hot plug and unplug on a bus.

Many DPAA2 objects are DMA-capable so that they can autonomously read and write memory. SoCs like the LS2088A contain an IO-MMU, so objects must express an identifier (that they cannot control) when they perform DMA operations. This identifier is called an ICID in DPAA2, and it serves as a key for the IO-MMU to associate I/O virtual addresses with I/O physical addresses. In DPAA2, ICIDs are attributes of DPRCs, and all objects in a DPRC express the same ICID value.

A GPP software context (a virtual machine or application) will typically be assigned a single DPRC that contains all the fsl-mc resources that the software context can access or use. As mentioned elsewhere, there are two general types of resources that can be in a container:

- **Resources:** Resources are primitive resources that can't be further decomposed, and are uninitialized and unpurposed. Some examples are MC portals, QBman portals, frame queues, buffer pools, etc. Generally primitives are “fungible,” in that there is nothing distinctive among the same kind of primitives. However, some primitives may be non-fungible, such as an external port or MAC.
- **Objects:** Objects are created and configured with a purpose, typically constructed of multiple resources. Some examples of objects are network interfaces, an L2 switch, or a crypto instance. A DPRC is itself an fsl-mc object.

NOTE

See documentation of the Linux restool facility for more information related to this topic.

Management Complex (MC) initialization and boot

The MC is normally enabled and initialized by system boot firmware such as U-Boot. The boot firmware is responsible for reserving a region of memory (DDR) for the fsl-mc, and then loading the MC firmware into memory, loading a datapath layout file (see below for DPL overview info), and writing a bit to enable/start the MC. See Management Complex initialization and boot.

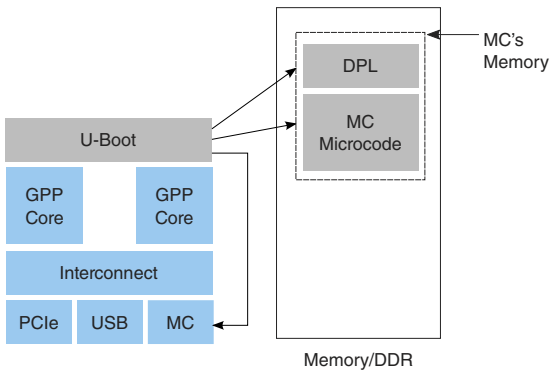


Figure 23. Management Complex initialization and boot

Management Complex datapath layout file (DPL)

As mentioned above, a datapath layout file (DPL) must be supplied to the Management Complex when it is booted. The DPL contains the definitions of initial objects and containers/DPRCs to create.

A DPL is defined in a text file in device tree syntax (DTS) format and then compiled into a standardized DTB binary format (used by ePAPR compliant device trees).

See the *DPAA2 User Manual* for more information and examples on the datapath layout file.

Boot loader use of the MC

In typical usage, the boot loader loads the MC firmware image and starts the MC running. At this time, it supplies a data path control (DPC) file that supplies the MC image with basic configuration information that allows it to operate.

The boot loader can now use the services of the MC in order to access network devices. It is a good approach to have the boot loader dynamically create the objects it needs and destroy them (releasing resources) before starting the operating system. This way, the operating system is not forced to operate with the constraints of objects and DPRCs established by the boot loader. The OS can see a "green field".

Optionally, the boot loader can apply the data path layout (DPL) file mentioned above just before starting this OS. This approach allows the DPL to be written only to serve the operating system's needs and not the boot loader's, which tend to be much simpler.

DPRCs are hierarchical

The MC manages DPRCs in a hierarchical relationship. There is a single root DPRC at the root of the hierarchy. That DPRC can have child DPRCs, children can have grandchildren, and so on. The root DPRC belongs to the root software context of the system, usually an OS or hypervisor. The root DPRC can further allocate its resources to its child DPRCs and assign them to other entities such as user space applications or virtual machines.

In this example there are 3 DPRCs/containers managed by the Management Complex: a root container “root” with 2 children “foo” and “bar”. The DPRCs all contain 3 objects, a DPNI, DPBP, and DPIO. There are 3 software contexts: the host Linux, a user space application, and Linux in a KVM virtual machine. Each software context is assigned a DPRC that it can use and manage; see DPRC hierarchy for a figure that illustrates this example.

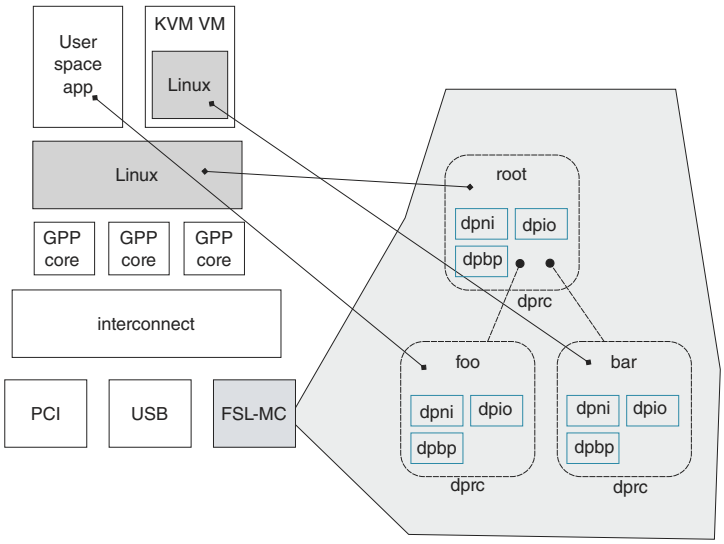


Figure 24. DPRC hierarchy

The container hierarchy allows the parent to manage the resources of the children. If the OS in the KVM VM crashes, the parent (Linux) can reset and clean up the VM's DPRC. If the user space application terminates, the parent (Linux) has the option of destroying the container.

6.1.4.3 Objects and topology

As mentioned elsewhere, objects have a topological relationship with each other. See Object topology example for an example.

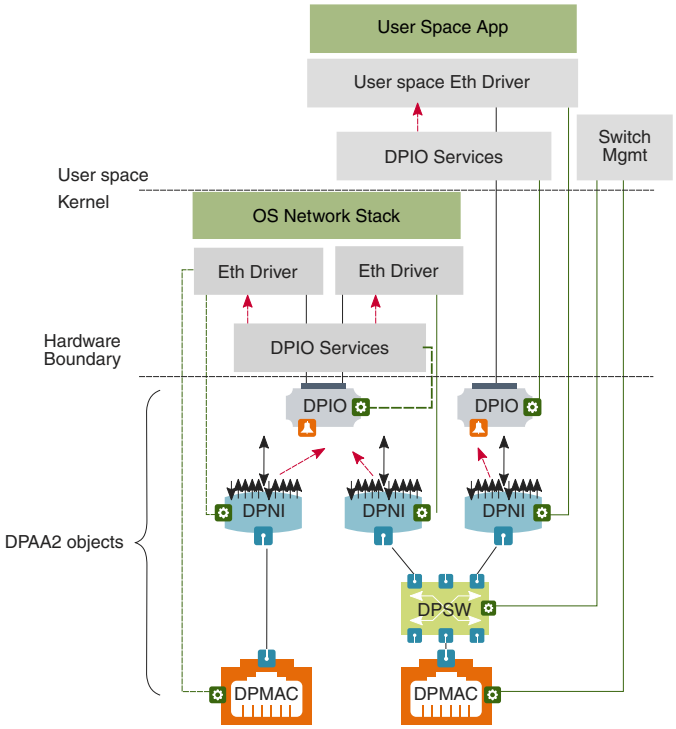


Figure 25. Object topology example

- There are three network interfaces managed by the DPAA2 Linux Ethernet driver. Each of the network interfaces uses a DPNI object.
- All of the Ethernet drivers happen to have a distribution width of one (an example), so they cannot load balance to multiple cores or threads; this was done to simplify the diagram and discussion. If a network interface has a distribution width greater than one, then many times it is connected to more than one DPIO but this is not required.
- Two of the network interfaces are connected to a switch; two DPNIs are connected to a DPSW. This allows both network interfaces to communicate outside of the SoC using the DPMAC that is also connected to the DPSW, and they can also communicate with each other using the DPSW.
- One of the network interfaces is directly assigned to a user space process, and has a user space Ethernet driver. This network interface could also be directly assigned to a KVM virtual machine under Linux.
- Two of the DPNIs have Linux network stack drivers; they interface to the Linux network stack. One of them has its own DPMAC, and a traditional type of controller represented by its DPNI being directly connected to a DPMAC.
- The two DPNIs connected to the Linux network stack share a single DPIO; this is possible when they can cooperatively use a layer of GPP software that provides DPIO services. The hardware that makes up a DPIO is a QBMan software portal and, optionally, a QMan channel for data availability notifications. QBMan software portals are a relatively scarce hardware resource, so they are designed to be sharable, in particular for NAPI-compliant Linux Ethernet drivers.
- It is a key assumption of DPAA2 that objects are managed (or “owned”) by a single software entity. Independent software entities can independently manage the objects they own, and this allows software to be decoupled from other entities.
- The management relationship between objects and software entities is not defined or imposed by DPAA2; DPAA2 defines the objects and what they do, and not what software uses them. Customer GPP core software is allowed to determine the management relationship; a single monolithic software entity that manages all of the objects can be created.

- The Linux DPAA2 Ethernet driver design defines the set of objects needed to provide a network interface. The green lines show the management relationships for Linux network interfaces and switches. Note that switches are managed independently from the network interfaces that connect to it.

The *DPAA2 User Manual* provides a complete description of the rules that govern object topology.

6.1.4.4 AIOP in DPAA2

This section describes AIOP in the DPAA2 architecture and how it uses DPAA2 objects. The figure below shows an example in which the Linux kernel network stack has a single Ethernet interface, a user space application has a single directly assigned Ethernet interface, and AIOP has two Ethernet interfaces.

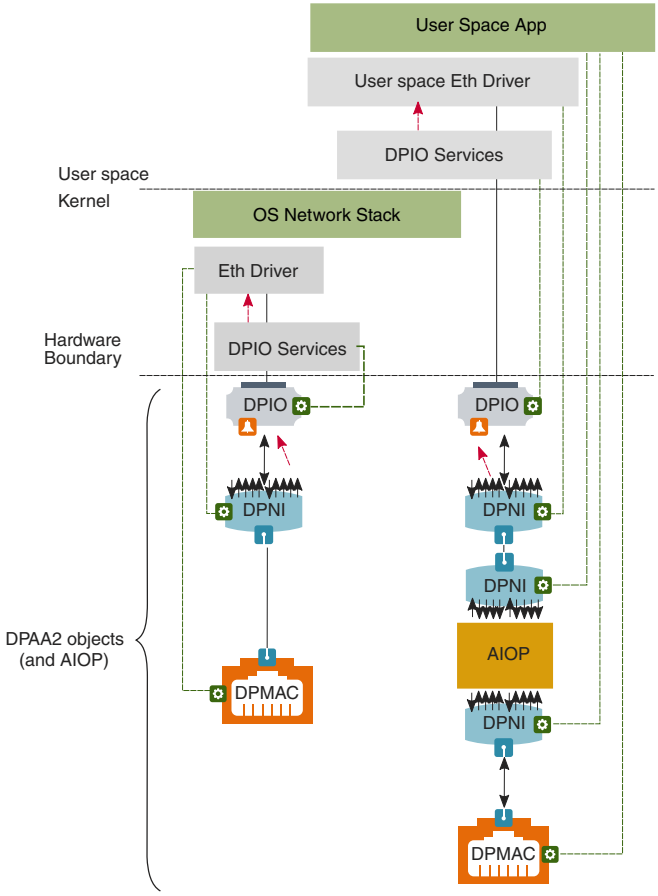


Figure 26. AIOP in DPAA2

AIOP runs software and uses objects in a manner that is similar to general purpose processing cores. In this example configuration, the AIOP's software has access to two Ethernet interfaces. One is connected to a MAC for an external connection to outside the SoC. The other is connected point-to-point to the general purpose processing core user space application's Ethernet interface.

So, there are two Ethernet interfaces involved on the path between the AIOP and the user space application. This is logical because there is software running in both places. Thus, each software component should see and control its own Ethernet Interface. Both software components do Ethernet I/O without being coupled to what their Ethernet interface is connected to.

One difference is that AIOP software is focused on packet processing. It does not actively manage or configure its own Ethernet interfaces. Usually, a control application on the general purpose processing cores takes that role. Also, the AIOP does not use DPIO objects.

Note that AIOP itself is not a DPAA2 object. It is an active entitiy that uses other DPAA2 objects. However, there is a DPAIOP object that general purpose processing core software can use to manage the AIOP, e.g. start it, stop it, load images onto it, get error status from it, etc.

In addition, there are DPAA2 objects (not shown) that facilitate passing commands (rather than packets) between general purpose processing core software and AIOP software.

AIOP Service Layer

NXP provides an AIOP software library called the "AIOP Service Layer". This library's main purpose is to provide very lightweight drivers for hardware components that AIOP software must access. These include components such as Ethernet intefaces, the QBMan buffer manager, timers, table lookup units, etc.

6.2 Introduction

This section provides instructions to enable users to execute usage examples provided in the SDK release. The section also provides information about key release files that users need to be familiar with to run the release examples.

6.2.1 Acronyms and definitions

AIOP	Advanced I/O Processor
ACL	Access Control List Permissions granting access to and allowing operations on a given object; a table type
CAAM	Cryptographic Acceleration and Assurance Module
DCE	Data Compression/Decompression Engine
DPAA	Data Path Acceleration Architecture
DPAA2	Data Path Acceleration Architecture, second generation
DPC	Data Path Configuration file
DPDMUX	Datapath De-multiplexer DPAA2 object modelling an EVB
DPL	Data Path Layout file
DPNI	DPAA2 object modelling a network interface
EVB	Edge Virtual Bridge
FDB	Forwarding Database A table used by an L2 switch to store MAC addresses learned and the respective ports on which they were learned
GPP	General Purpose Processor
MC	Management Complex
NAT	Network Address Translation
ODP	Open Data Plane
QBMan	Queue and Buffer Manager
RCW	Reset Configuration Word Sets up clocks and ratios, IP pin muxing on boards; resides in non-volatile memory (e.g. NOR) on boards
SEC	Security Engine

STP	Spanning Tree Protocol A network protocol preventing loop formation when switches or bridges are interconnected via multiple paths
VEB	Virtual Ethernet Bridge
VEPA	Virtual Ethernet Port Aggregator (defined in IEEE 802.1Qbg)
WRIOP	Wire Rate I/O Processor

6.3 DPAA2 Software

For information about the DPAA2 software, see the DPAA2 Software Overview that is part of the release documentation. For customers that are unfamiliar with DPAA2 concepts, it is recommended to read that content before proceeding.

6.4 Data Path Resource Containers

Many sections refer to Data Path Resource Containers (DPRC), so a brief introduction to the concept may be helpful. DPRCs are part of the DPAA2 object architecture that is described in the DPAA2 Software Overview that is part of the Linux SDK documentation set.

DPRCs are communicated to software entities as a part of their start up process; this is true for software entities such as:

- The host Linux kernel (that may provide KVM services to virtual machines)
- Linux kernel instances that run in virtual machines
- ODP applications
- AIOP applications

DPRCs contain DPAA2 objects that are used by the software entity that owns the DPRC. For example, DPNI objects are used as network interfaces.

As an example, see RDB DPL on page 438. The DPRC called “dprc@1” is supplied to the host Linux kernel. It contains one DPNI object. The DPNI object binds to the DPAA2 Linux kernel Ethernet driver, and causes two standard Linux Ethernet interfaces to exist and be visible using “ifconfig”. See later sections in this document for additional details and explanations on the use of objects by various types of software entities.

As mentioned previously, DPRCs must be created and populated with the initial set of DPAA2 objects prior to the startup of the software entity that will use the DPRC.

6.4.1 Creating DPRCs

There are two ways to create and populate DPRCs:

1. Statically: by means of a control file called a datapath layout (DPL) file. Key Release Files: RCW, DPC and DPL on page 436 describes the DPL files that are supplied as examples with the Linux SDK.
2. Dynamically: by means of the Linux command line utility called restool. See DPRCs and restool section, and also the document titled *Standard Linux Documentation.*

A software entity behaves exactly the same way on startup regardless of whether its DPRC was created statically using a DPL or dynamically using restool. The DPL method is convenient for situations when the desired DPRCs are known in advance. DPRCs defined within the DPL are created and populated automatically with no need for a subsequent use of restool.

6.4.2 DPRCs and Hot Plug

A DPRC must be supplied to a software entity when the software entity is started; this implies prior creation of the DPRC. However, it is also possible to dynamically alter the contents of a DPRC *after* the software entity that is using it is already running; this is a form of hot plug.

For example, restool can be used to dynamically create a DPNI object and then assign it to a DPRC. If that DPRC is being used by a Linux kernel instance, this will cause that kernel to dynamically detect a new network interface and bind it to the Linux kernel Ethernet driver; the “ifconfig” command will now show a newly created network interface.

It is also possible to dynamically destroy or unassign objects within an in-use DPRC; this is a form of hot unplug. Hot plug and unplug are advanced topics, and not covered further here. The key take-away is that dynamically creating and populating a DPRC *before* supplying it to software entity when it is started is a very different use case than hot plug/unplug. The latter use case involves changing the contents of a DPRC *while it is in use* by a software entity.

6.5 Key Release Files: RCW, DPC and DPL

This section describes the key binaries that are available on the RDB. These include the reset configuration word (RCW), the data path configuration (DPC) and the data path layout (DPL) files.

6.5.1 RCW (LS2088A)

The reset configuration word (RCW) resides in non-volatile memories (e.g. NOR, QSPI, SDHC). It gives flexibility to accommodate a large number of configuration parameters to support a high degree of configurability of the SoC. Configuration parameters generally include:

- Frequencies of various blocks including cores/DDR/interconnect.
- IP pin-muxing configurations
- Other SoC configurations

The following binaries are provided:

- `PBL_1600_600_1600_1600_0x2a_0x41.bin`
  - Core: 1600 MHz
  - Platform: 600 MHz
  - DDR: 1600 MHz
- `PBL_1800_700_1866_1600_0x2a_0x41.bin`
  - Core: 1800 MHz
  - Platform: 700 MHz
  - DDR: 1866MHz

The RCWs provided with the release enable the following features:

- The figure below shows the SERDES configuration supported for LS2088A. Note that LS2088A supports upto 5 ports out of the 8 available on the RDB at a time.

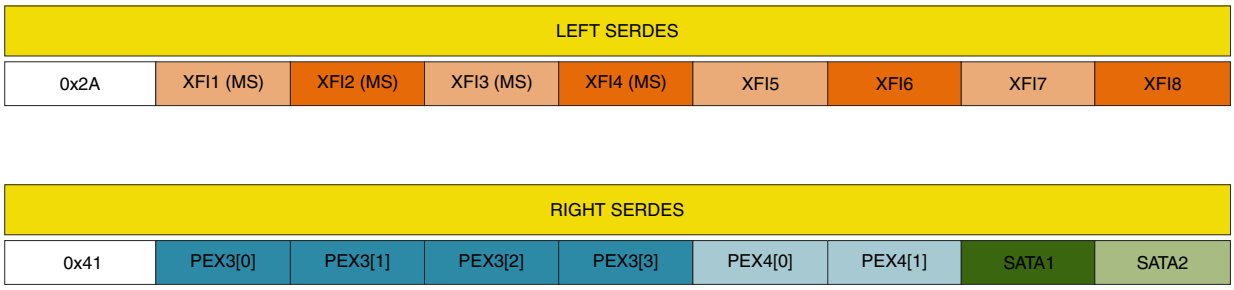


Figure 27. SERDES

- Boot location as NOR flash
- Enables 4 UART without flow control
- Enables I2C1, I2C2, I2C3, I2C4, SDHC, IFC, PCIe, SATA

6.5.2 Data path configuration file (DPC)

The data path configuration (DPC) contains board-specific and system-specific information that overrides the default DPAA hardware configuration.

The SDK release provides one example data path configuration (DPC) file. The file is named dpc-0x2a41.dts. This file specifies the following information:

- default logging mode for the MC
- default LS2085ARDB/LS2088ARDB MACs
- default number of DPAA channels with 2 and 8 work-queues

The DPC is based on a text source file (similar to a device tree source file (DTS)) and compiled with the DTC utility to form a binary structure (blob, similar to DTB). The DPC file should be compiled to a binary blob using standard DTC tool.

The DPC binary (dpc-0x2a41.dtb) is located in the IMAGE iso under *dpl-examples/ls2088a/RDB/*.

6.5.3 Data path layout file (DPL)

The data path layout file (DPL) defines the containers created during MC initialization. A prerequisite for the DPL to define the containers created during MC initialization is that the device tree compiler (dtc) tool is installed on the host system, since the dtc command is used to compile the DPL.

As described in Creating DPRCs on page 435, the example DPL source code is provided in the *dpl-examples* package.

This release provides one example data path layout (DPL) files to be used with the RDB:

- dpl-eth.0x2A_0x41.dts – used for simple Ethernet scenarios

The DPL file specifies the basic resources needed for a simple usecase; other resources are created and managed dynamically using restool capabilities. For each of the use cases included in this document, there is a diagram that depicts the objects that are necessary for that usecase.

The DPL is based on a text source file (similar to a device tree source file (DTS)) and compiled with the DTC utility to form a binary structure (blob, similar to DTB). The DPL file should be compiled to a binary blob using standard DTC tool.

Using a static DPL is not a requirement since restool can be used to dynamically create/manage objects and resources.

6.5.3.1 RDB DPL

The source for the RDB DPL is in the *dpl-examples* package:

- dpl-examples/LS2088a/RDB/dpl-eth.0x2A_0x41.dts

The figure below shows a graphical view of the container configuration:

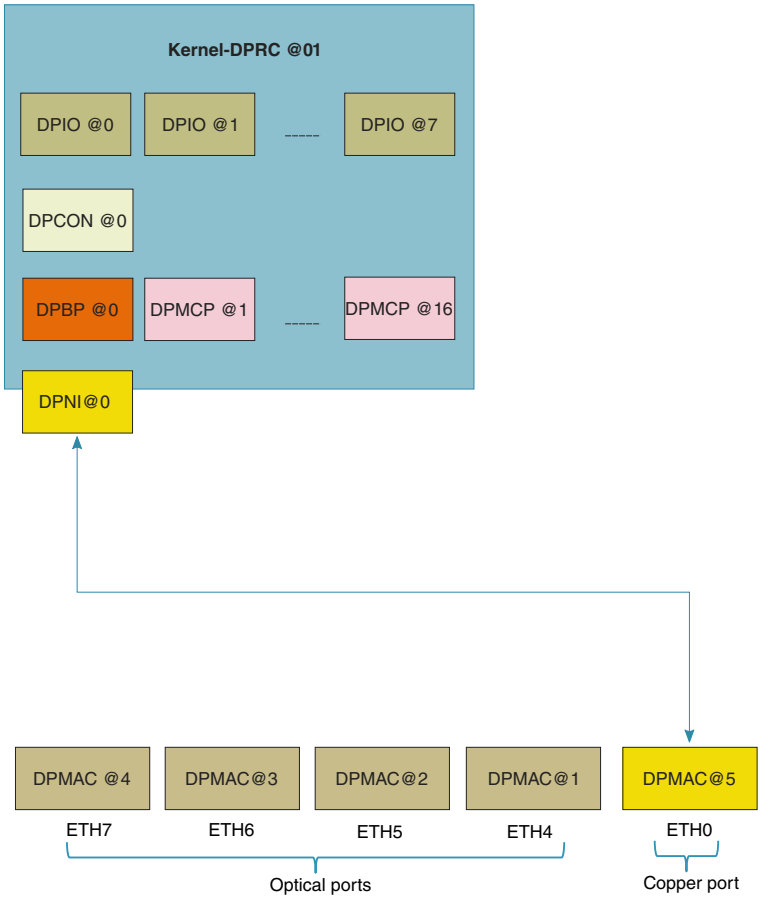


Figure 28. RDB DPL container configuration

6.5.3.2 DPRCs and restool

The release provides a Linux command-line tool called restool that can be used for examining the resource containers used for managing DPAA2 objects and resources. See the DPAA2 Overview for an overview of DPAA2 and the data path resource containers (DPRCs). Also see the Stardard Linux Documentation for details about functionality and use of restool.

Given below is an example of using restool:

List dprc:

```
$ restool dprc list
dprc.1
```

List all objects in container dprc.1:

```
$ restool dprc show dprc.1
dprc.1 contains 33 objects:
object          label           plugged-state
dpio.7                          plugged
dpio.6                          plugged
dpio.5                          plugged
dpio.4                          plugged
dpio.3                          plugged
dpio.2                          plugged
dpio.1                          plugged
dpio.0                          plugged
dpni.0                          plugged
dpbp.0                          plugged
dpmac.5                         plugged
dpmac.4                         plugged
dpmac.3                         plugged
dpmac.2                         plugged
dpmac.1                         plugged
dpcon.0                         plugged
dpmcp.0                         plugged
dpmcp.16                        plugged
dpmcp.15                        plugged
dpmcp.14                        plugged
dpmcp.13                        plugged
dpmcp.12                        plugged
dpmcp.11                        plugged
dpmcp.10                        plugged
dpmcp.9                         plugged
dpmcp.8                         plugged
dpmcp.7                         plugged
dpmcp.6                         plugged
dpmcp.5                         plugged
dpmcp.4                         plugged
dpmcp.3                         plugged
dpmcp.2                         plugged
dpmcp.1                         plugged
```

Get information about dpni.0:

```
dpni version: 7.1
dpni id: 0
plugged state: unplugged
endpoint state: -1
endpoint: No object associated
link status: 0 - down
mac address: 00:00:00:00:00:00
dpni_attr.options value is: 0
num_queues: 1
num_tcs: 1
mac_entries: 16
vlan_entries: 0
qos_entries: 0
fs_entries: 64
qos_key_size: 0
```

```
fs_key_size: 56
ingress_all_frames: 0
ingress_all_bytes: 0
ingress_multicast_frames: 0
ingress_multicast_bytes: 0
ingress_broadcast_frames: 0
ingress_broadcast_bytes: 0
egress_all_frames: 0
egress_all_bytes: 0
egress_multicast_frames: 0
egress_multicast_bytes: 0
egress_broadcast_frames: 0
egress_broadcast_bytes: 0
ingress_filtered_frames: 0
ingress_discarded_frames: 0
ingress_nobuffer_discards: 0
egress_discarded_frames: 0
egress_confirmed_frames: 0
```

6.6 Linux Ethernet

This chapter provides guidelines on exercising creation, functionality and statistics of Linux DPAA2 Ethernet interfaces.

6.6.1 Features overview

The following is an overview of the functionality of the Linux DPAA2 Ethernet driver that will be described in this chapter:

- Primary MAC address change
- Scatter-gather support
- Checksum offload
- MAC filtering
- Large frame support
- GRO – generic receive offload
- Egress traffic shaping
- Rx hashing
- Interface statistics

6.6.2 Compiling and selecting Kconfig options

The DPAA2 Ethernet driver is by default selected by the kernel configuration shipped with the SDK, with a set of sensible compile-time defaults. The driver path in the kernel config file is: `" Device Drivers -> Staging drivers -> Freescale DPAA2 devices -> Freescale DPAA2 Ethernet" (CONFIG_FSL_DPAA2_ETH)`.

The following Kconfig selects are also available, but not checked by default:

- "Enable Rx error queue" `(CONFIG_FSL_DPAA2_ETH_USE_ERR_QUEUE)` - This configures a separate queue for presenting Rx Error frames to the Ethernet driver running on the GPP. By default this option is disabled and Rx Error frames are dropped in hardware and counted for statistics inspection. Rx Error processing increases the GPP load and so it is only necessary in debugging situations when inspection of actual frames is required.
- "Enable debugfs support" `(CONFIG_FSL_DPAA2_ETH_DEBUGFS)`. This option depends on two others: "NXP QBMAN Debug API" `(CONFIG_FSL_QBMAN_DEBUG)` and "Debug Filesystem" `(CONFIG_DEBUG_FS)`. If selected with its

dependencies, this option enables a number of Ethernet driver statistics under DebugFS nodes `(/sys/kernel/debug/dpaa2-eth/*)` that are not normally visible to the user.

6.6.3 Creating a DPAA2 network interface

This section documents the resource utilization and the necessary steps for creating a DPAA2 network interface (DPNI) in Linux.

A DPNI can be created either statically through the DPL file or dynamically using the 'restool' utility.

6.6.3.1 DPAA2 objects dependencies

This section documents the steps to create a DPNI and related objects in order to have a fully functional network interface. It describes the dependencies a DPNI has on other DPAA2 objects.

This is of interest to anyone who changes a static DPL file or uses *restool* commands to dynamically createdynamically create DPNI objects and their dependencies.

The DPAA2 object definition allows for flexibile software architectures. The Linux drivers, in particular the Ethernet driver, are additionally bound by requirements from the kernel architecture. This enforces a certain usage model of the DPAA2 objects that the DPNI interacts with; in particular, it affects the number of various other DPAA2 objects that a DPNI needs.

Generally, the Linux Ethernet driver requires private DPAA2 resources (e.g. Frame Queues) and objects (e.g. DPCON objects), distinct from other DPNIs. There are exceptions, such as the DPIO or DPMAC, which are not owned by the DPNI. To create a DPNI, either statically in DPL or dynamically using 'restool', the following types of objects may need to be instantiated in the current container (i.e., made available if they are not already):

- DPBP
- DPMCP
- DPCON
- DPIO
- DPMAC

The significance and number of these objects per DPNI are detailed in the following table:

Object	Private to DPNI?	Cardinality	Comments
DPBP	Yes	1 per DPNI	Each network interface (NI) has private buffer pools, not shared with other NIs.
DPMCP	Yes	1 per DPNI 1 per DPMAC	MC command portals (MCPs) are used to send commands to, and receive responses from, the MC firmware. One such example is configuring DPNI functionality like hashing or checksumming, but DPNI statistics are also queried via the MCPs. Like the DPNI, each DPMAC also requires one private DPMCP.

Table continues on the next page...

Table continued from the previous page...

Object	Private to DPNI?	Cardinality	Comments
DPCON	Yes	Rx hash size/ number of transmitter queues (“num_queues”) of the DPNI.	DPCONs are used to distribute Rx or Tx Confirmation traffic to different GPPs, via affine DPIO objects. The implication is that one DPCON must be available for each GPP we want to distribute Rx or Tx Confirmation traffic to. Rx and Tx Confirmation share the same DPCONs if they are available. (If for example GPP #0 processes both types of traffic, one DPCON is enough. If in addition GPP #1 processes only Tx Confirmation traffic, then a second DPCON is necessary.) Since we must be able to distinguish between traffic from different NIs arriving on the same GPP, the DPCONs must be private to the DPNIs. These design constraints may cause a relatively large consumption of DPCONs by DPNIs with large Rx distribution width. The DPNI's Rx distribution width is implemented by the "num_queues" property (see Rx hashing on page 449 for extra information). Notes: DPCONs' main hardware resource are the Work Queues (WQs). The DPCONs come in 2 flavours: 2-WQ and 8-WQ DPCONs, depending on the number of traffic class priorities the object is going to support. (Note: the Ethernet driver only supports one traffic class at the moment, so using 2-WQ DPCONs is safe and enough.) The MC firmware can convert any number of 8-WQ DPCONs to four times as many 2-WQ DPCONs, depending on the static configuration provided at boot. Since WQs are a limited hardware resource, DPCONs tend to be limited, too, especially the 8-WQ flavor. The DPNIs being one of the major consumers of DPCONs, the current SDK ships with a default configuration where a number of the 8-WQ DPCONs are converted to 2-WQ DPCONs, thereby increasing their availability. Note that DPIO objects themselves transparently consume DPCONs (one per DPIO object), which therefore must be substracted from the total number available to the DPNIs (they need not be explicitly declared in the DPL, but they are simply not available to the rest of the system). The system can provide up to 64 8-WQ DPCONs (and up to 256 2-WQ DPCONs and combinations thereof). So for a container with 8 DPIOs, only up to 56 8-WQ DPCONs will be in fact available for DPNI configuration.
DPIO	No	One per running GPP	DPIOs are used to provide data availability notifications to the GPPs. For each GPP that we want to distribute traffic to, there must be an affine DPIO. While DPIOs are the source of data availability interrupts, the DPCONs are used (among other things) to identify the NI that has produced ingress data to that GPP. Due to a known limitation, the number of DPIOs in a container must not be less than the number of running GPPs. The static DPL in this release defensively provides 8 DPIOs at boot-time, one for each running GPP.

Table continues on the next page...

Table continued from the previous page...

Object	Private to DPNI?	Cardinality	Comments
DPMAC	No	User-defined.	DPMACs are proxy objects which link DPNIs to external PHYs on the board. DPMACs effectively decouple DPNIs from the PHYs they are linked to (if they are indeed linked to an external PHY, which is in fact transparent to the DPNI). As such, the DPMACs are not "owned" by a DPNI, which is unaware of their presence, but they can be "connected" to the DPNI, via the DPL file or 'restool'. DPMACs can be connected to other types of objects, too, such as the EVB. Having DPMACs connected to external PHYs depends on the board wiring and is strictly confined to the SerDes configuration. See also: DPMAC configuration on page 444.

6.6.3.2 Static DPNI definition

The default DPL provides a simple DPNI object definition, under the dpni@0 node as follows:

```
dpni@0 {
   options = "";
   num_queues = <1>;
   num_tcs = <1>;
};
```

The DPNI object is linked to a DPMAC object, also created in the DPL, via the “connections” node as follows:

```
connections {
   connection@5{
       endpoint1 = "dpni@0";
       endpoint2 = "dpmac@5";
   };
};
```

The DPNI object can be more complex, as in the following enhanced example of a DPNI node:

```
dpni@1 {
     options = "DPNI_OPT_HAS_KEY_MASKING";
     num_tcs = <1>;
     num_queues = <8>;
     mac_filter_entries = <64>;
};
```

In this example, dpni@1 has more options declared than dpni@0 in the previous example. In addition, it can distribute traffic to more GPPs than dpni@0, as declared by the “num_queues” attribute.

The following section describes the DPNI bindings in the DPL file.

6.6.3.2.1 DPNI bindings

.

- The num_queues attribute indicates the number of queues to be used for transmission as well as the number of Rx queues (hash distribution size). This also implicitly defines the number of queues used for Tx Confirmation, since each "sender" uses a dedicated queue for confirmations. This may impact the number of necessary DPCON objects - see "DPAA2 objects dependencies" chapter for details on resourcing the DPNI.

- num_tcs represents the number of traffic classes; it must have a value of 1, since the driver does not support priority classes yet.
- Other possible attributes are listed below. Unless otherwise stated, attributes with value <0> receive a default, non-trivial, value from the MC firmware and can be skipped from the DPL altogether.
  - fs_entries
  - vlan_filter_entries
  - mac_filter_entries

Note: See MC documentation for all available options and supported values.

6.6.3.3 DPMAC configuration

This section is a brief introduction to DPMAC objects and their relation to the DPNIs.

DPMACs are essentially proxies to external PHYs, which are board-level components and therefore not managed by the MC firmware.

DPMACs can be connected to other DPAA2 objects, such as DPNI, DPDMUX and DPSW. For example, to statically connect a DPMAC to a DPNI in the DPL file, the “connections” node is used:

```
connection@5{
     endpoint1 = "dpni@0";
     endpoint2 = "dpmac@5";
};
```

In this DPL example, DPNI0 is connected to DPMAC5, which the MC thereon connects to a lane depending on the current SerDes. Unlike most other DPAA2 objects, the id of the DPMAC (in this example, “5”) is relevant as the MC uses it to identify a physical MAC (at the moment, there is no other property of the DPMAC object to do that).

6.6.3.4 Dynamically creating a DPNI

This chapter documents the steps to create a DPNI using the *restool* utility and the *restool* wrapper scripts.

6.6.3.4.1 Using restool to create a DPNI

DPNIs can be dynamically created and plugged into the Linux container using the restool utility. Before creating a DPNI, one must create a number of DPAA2 objects (dependencies), for which multiple *restool* commands are needed. This section provides simple examples of commands that should be used to create a working DPNI (Linux network interface) and its dependencies. Usage of the restool wrapper shell scripts bundled with the SDK is encouraged, because of their better ease-of-use.

In order to create an object, the “restool create” command must be executed and then the new object can be assigned to a container. For example, to create a DPBP object:

```
$ restool dpbp create
dpbp.1 is created under dprc.1
$ restool dprc assign dprc.1 --object=dpbp.1 --plugged=1
```

For automation purposes, the “--script" flag can be used, reducing the verbosity of the command output. Object properties can be specified at creation time as follows:

```
$ restool --script dpio create --channel-mode="DPIO_LOCAL_CHANNEL" --num-priorities=8
dpio.8
```

To create a DPNI, a number of DPMCP, DPBP and other dependencies are required, if they do not exist already in the container - refer to the DPNI Resource Utilization chapter for details on the types and number of DPNI dependencies. The

static DPL from the current release already defines 8 DPIO objects, one for each running GPP, so adding new DPIOs is not normally required. Also, the maximum number of DPMACs supported on LS2088A are already created in the static (default) DPLs, so adding new ones is not necessary in the default configuration.

The general steps to create and configure a DPNI using restool are:

1. Create DPAA2 object dependencies (DPBPs, DPCONs, DPMCPs, etc);
2. Create and parametrize the DPNI;
3. Connect the DPNI to another object (typically but not necessarily a DPMAC).

The restool wrapper shell scripts automatically take care of the DPNI resourcing and parametrization, therefore we encourage their use instead of bare *restool* for complex objects like the DPNI.

6.6.3.4.2 Restool Wrapper Scripts

User-friendly scripts are provided in the release rootfs to assist dynamic creation of DPNIs and associated dependencies. They also implement parameter restrictions and workarounds related to known limitations of the DPAA2 objects in the current SDK release.

The following scripts are available to interact with DPNI and DPMAC objects, respectively Linux network interfaces: ls-addni, ls-listni, ls-listmac.

1. **ls-addni**

   This script creates a new DPNI object, required dependencies (potentially DPBP, DPMCP, DPCON, DPMAC, depending on the options being passed to the script) and an associated Linux network interface. The script can be used to connect the newly-created DPNI to another DPNI, DPMAC or DPDMUX, which must be already created and not currently connected.

   The script supports a multitude of parameters to fine-tune configuration of the DPNI; in fact, it is intended to support every parameter as *restool* itself for creating DPNIs. An empty list of options will choose sensible defaults for maximal performance of the new DPNI, such as Rx hashing to the maximum number of cores.

   Adding a new DPNI has the effect of discovering the new object on the Linux mc-bus and probing it as a new Ethernet device. This results in a new Linux network interface becoming available. The new interface has the name ni<X>, with the same id as the just-created dpni.<X>. The actual value of the ni id <X> cannot be controlled, i.e. the user cannot request a specific ni<X>.

   Utilization examples:

   - ```
     # ls-addni dpmac.6
     Will allocate 8 DPCON objects for this hash size
     [   33.782201] fsl_dpaa2_eth dpni.1: Probed interface ni1
     Created interface: ni1 (object:dpni.1, endpoint: dpmac.6)
     ```

     This is probably the most typical usage example. It creates a network interface (ni1) and the underlying dpni (dpni.1) and connects it to an external MAC (dpmac.6).

     Connecting DPNIs to DPMACs is not the only option, though:

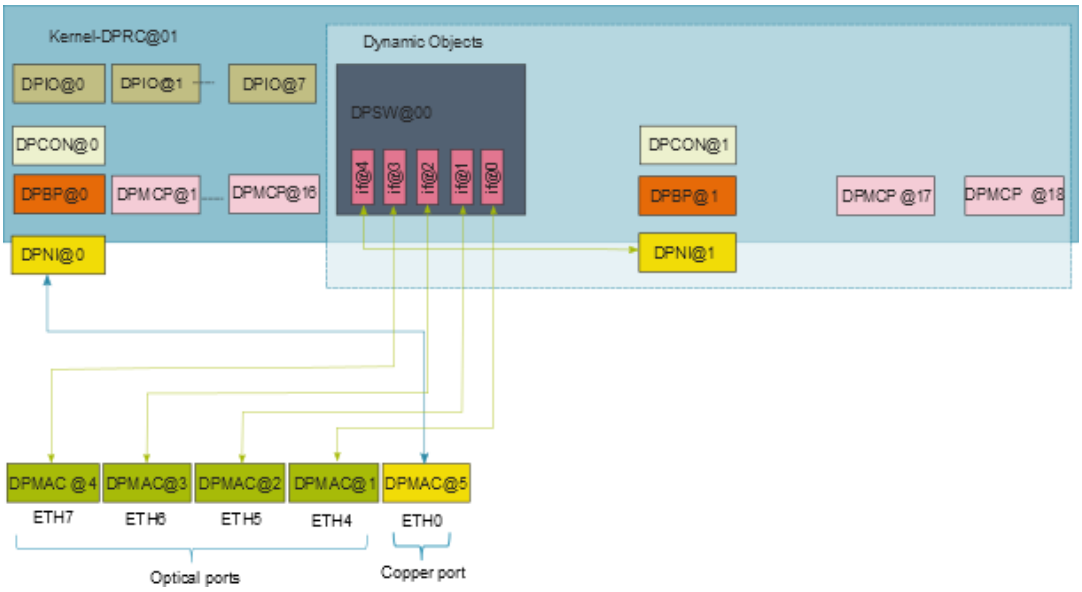
   - ```
     # ls-addni -n
     Will allocate 8 DPCON objects for this hash size
     [  159.132755] fsl_dpaa2_eth dpni.2: Probed interface ni2
     Created interface: ni2 (object:dpni.2, endpoint: none)
     ```

     This command creates the unconnected object dpni.2 and the respective Linux interface ni2. Not being connected to anything, there is little practical use for this interface; therefore, a command such as the following would be used:

   - ```
     # ls-addni dpni.2
     Will allocate 8 DPCON objects for this hash size
     ```
     ```

```
[  301.162763] fsl_dpaa2_eth dpni.3: Probed interface ni3
Created interface: ni3 (object:dpni.3, endpoint: dpni.2)
```

This command creates another network interface ni3 (and the underlying dpni.3 object) and connects it with the previously created ni2 (dpni.2) interface.

Notes:

ls-addni --help list all supported options.

Although it is technically possible to connect a DPNI to itself, the wrapper scripts do not support this;

2. **ls-listni**

   This script lists all the dpni objects available in the root and child containers, the associated network interface name, the end point and the label.

   Output after running the above examples (dpni.0 had been statically defined in the DPL):

```
# ls-listni
dprc.1/dpni.3 (interface: ni3, end point: dpni.2)
dprc.1/dpni.2 (interface: ni2, end point: dpni.3)
dprc.1/dpni.1 (interface: ni1, end point: dpmac.6)
dprc.1/dpni.0 (interface: ni0, end point: dpmac.5)
```

3. **ls-listmac**

   This script lists all the dpmac objects available in the root and child containers, the associated network interface name, the end point and the label.

   Output after running the above examples (dpni.0 had been connected to dpmac.5 in the static DPL):

```
# ls-listmac
dprc.1/dpmac.8
dprc.1/dpmac.7
dprc.1/dpmac.6 (end point: dpni.1)
dprc.1/dpmac.5 (end point: dpni.0)
dprc.1/dpmac.4
dprc.1/dpmac.3
dprc.1/dpmac.2
dprc.1/dpmac.1
```

6.6.4 DPAA2 Ethernet features

This section presents the individual functions of the Linux DPAA2 Ethernet driver.

6.6.4.1 Bring up the bootstrap DPNI interface

From Linux, interfaces are visible through the '`ifconfig`' command. The DPAA2 interfaces are named as "`ni<X>`", where X is the number of the underlying DPNI device in the DPL file or as dynamically created with 'restool'.

The default DPL file shipped with the current BSP release contains one statically-defined DPNI object (DPNI.0), visible in Linux as interface "ni0".

DPNI.0 is configured with a minimal set of resources – e.g. it can only receive traffic on GPP0 – and its intended uses are network boot and low-bandwidth traffic. For fully-featured DPNI objects, dynamic configuration is recommended (see Dynamically creating a DPNI on page 444).

For IP connectivity between ni0 and an external host, first assign a valid IP address to ni0 as in the following example:

```
$ ifconfig ni0 192.168.1.2
```

Assuming the remote peer has address 192.168.1.1, ping to test as shown:

```
$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=87.0 ms
```

Notes:

- A SSH server is running by default and can be connected to, via ssh or scp, for example:

```
$ scp root@192.168.1.2:/bin/sh . # copy remote /bin/sh over
```

- A tftpd server is also installed; with the current configuration it must be explicitly started first. The steps to start tftpd, bound to local address 192.168.1.2, on port 69 (note the "." at the end, indicating the current directory, from where it will serve files to tftp clients):

```
$ udpsvd -vE 192.168.1.2 69 tftpd .
```

- The steps to connect an external tftp client to the RDB board depend on the actual tftp client. For example, to use the tftp client from the RDB board itself and get <some_filename> from a remote server, use the following command:

```
$ busybox tftp -g -r <some_filename> 192.168.1.1
# Note the explicit invocation of the Busybox tftp
```

6.6.4.2 Primary MAC address change

Changing the primary MAC address of a Linux Ethernet interface is supported without the need to bring down the interface.

For example:

```
$ ifconfig ni0 hw ether 02:00:C0:01:02:0a
```

```
$ ip link set dev ni0 address 02:00:C0:01:02:0b
```

6.6.4.3 Scatter/gather configuration

The Ethernet driver supports scatter/gather (S/G) on both the transmit and receive side. The S/G option can be configured through ethtool on Tx; on Rx, S/G support is always on. For example, in order to see the current state of the device features and hardware offloads for device ni0 :

```
$ ethtool -k ni0
Features for ni0:
[…]
scatter-gather: on
        tx-scatter-gather: on
[…]
```

In order to change the S/G status of the Linux Ethernet interface ni0:

```
$ ethtool -K ni0 sg off
Actual changes:
scatter-gather: off
        tx-scatter-gather: off
generic-segmentation-offload: off [requested on]

$ ethtool -K ni0 sg on
Actual changes:
```

```
scatter-gather: on
        tx-scatter-gather: on
generic-segmentation-offload: on
```

Notes:

- S/G support on the *egress* path, together with High DMA, which is also supported, allows for efficiently transmitting TCP segments from user-space, without copying them to kernel-space first ("Tx zero-copy").
- Egress S/G is necessary for other kernel features such as generic segmentation offload (GSO, implicitly turned on).
- The Ethernet driver support for S/G frames on the *ingress* path is transparent to the user.

6.6.4.4 Checksum offload configuration

The Ethernet driver supports hardware offloading of both Rx checksum validation and Tx checksum generation for TCP and UDP over IPv4/IPv6. The hardware checksum offload can be configured through ethtool.

For viewing the current state of the feature use the "-k" flag:

```
$ ethtool –k ni0
Features for ni0:
[…]
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: on
        tx-checksum-ipv6: on
[…]
```

The checksum offload can be controlled separately on Rx and Tx paths as follows:

```
$ ethtool –K ni0 rx on|off
$ ethtool –K ni0 tx on|off

$ ethtool -k ni0 | grep tx-checksumming
tx-checksumming: off
```

6.6.4.5 MAC filtering

The DPAA2 hardware supports unicast and multicast MAC filters on the ingress path. In Linux, MAC unicast filtering can be accomplished with the help of MACVLAN interfaces. Kernel configuration and DPNI configuration are required to enable this feature, as follows:

- To enable support in the kernel, CONFIG_MACVLAN must be selected at compile-time, from the kernel menuconfig:
  `"Device Drivers -> Network device support -> Network core driver support -> MAC-VLAN support" .`

The Linux Ethernet driver allows adding and deleting of MAC filters via the standard "ip" command. An example of adding/deleting a MAC unicast address is the following:

```
$ ip link add link ni0 address <macvlan_mac_addr> ni0.1 type macvlan
$ ifconfig ni0.1 up
[…]
$ ip link delete ni0.1 type macvlan
```

Adding a multicast address is also possible using the "ip" command as follows:

```
$ ip maddr add 01:00:00:00:00:01 dev ni0
```

6.6.4.6 Large frame support

The DPAA2 hardware supports large frames. The Ethernet driver correlates between the Layer-2 maximum frame length (MFL) and Layer-3 MTUs. The maximum MTU that a Linux user can request on a DPAA2 Ethernet interface is 10222 bytes.

Notes:

- Outgoing packets larger than the current MTU are going to be fragmented by the kernel stack.
- All Ethernet devices on the same LAN must have the same MTU

6.6.4.7 Generic receive offload

The DPAA2 Ethernet driver is integrated with the kernel's generic receive offload (GRO) support. GRO is enabled by default and is configurable via "ethtool":

```
$ ethtool -k ni0 | grep generic-receive-offload
 generic-receive-offload: on
$ ethtool -K ni0 gro off
$ ethtool -k ni0 | grep generic-receive-offload
 generic-receive-offload: off
```

NOTE

For better performance, GRO should be disabled on the receiving interfaces in certain scenarios such as IP Forwarding.

6.6.4.8 Egress traffic shaping

The DPAA2 Ethernet interface supports traffic shaping in the egress path. Egress shaping can be set or cleared via a per-interface entry in SysFS:

```
$ echo M N > /sys/class/net/ni0/tx_shaping
```

where:

M is the maximum throughput, expressed in Mbps.

N is the maximum burst size, expressed in bytes, at most 64000.

To remove shaping, use M=0, N=0.

6.6.4.9 Rx hashing

The DPAA2 Ethernet driver supports hash distribution of ingress flows, based on some L2/L3/L4 fields. Rx hashing is enabled bydefault if num_queues has a value larger than 1. The distribution key is not user configurable, but can be viewed through ethtool:

```
$ ethtool -n ni1 rx-flow-hash <proto_type>
```

The hashing configuration is the same for all protocols.

NOTE

The ni0 interface present by default (via the static DPL) has num_queues = <1>, so Rx hashing is not available on ni0 unless it is dynamically destroyed and re-created.

For full details and examples on dynamic DPNI creation using *restool*, refer to this chapter. The interfaces created using those examples do support Rx hashing.

6.6.4.9.1 Rx hashing size

The hashing size (number of target GPPs to which the ingress traffic is distributed) is configured through the " `num_queues` " option in the DPNI node.

Also worth noting is that enough DPCON objects must be available when a DPNI is probed, in order to sustain the required hashing size. For details on the resource utilization of the DPNI, refer to DPAA2 objects dependencies on page 441.

6.6.4.10 Rx flow steering

The DPAA2 Ethernet driver supports steering of ingress traffic, directing flows to specific GPPs based on exact-match operations on some of the common L2/L3/L4 fields. The advantage versus Rx hashing on page 449 is cache locality of ingress data: the user-space applications that actually process the traffic make better use of the local GPP's cache than if the traffic were processed on another GPP. The disadvantage stems from the static configuration of flow affinity and from the fact that flow characteristics (e.g. L4 ports) must be known in advance, which is not always possible in real scenarios.

Configuration is done via standard "ethtool" support as follows:

```
$ ethtool -N ni1 flow-type <proto_type> <header_field> <value> [m <mask>] action <cpu_id>
```

Steering is supported on a subset of fields: SMAC, DMAC, VLAN tag, IP SRC, IP DST, IP next proto, L4 ports. Masking of header fields is also supported.

For example, in order to set up flow steering based on destination IP:

```
$ ethtool -N ni1 flow-type ip4 dst-ip 192.168.1.0 action 0
```

or subnet:

```
$ ethtool -N ni1 flow-type ip4 dst-ip 192.168.2.0 m 0.0.0.255 action 1
```

NOTE

Currently, flow steering can be enabled only if hashing is also enabled. Flow steering is only available for L2/L3/L4 fields that are also part of the Rx hash key. Additionally, the MC firmware and Linux Ethernet driver will only configure flow-steering if `DPNI_OPT_HAS_KEY_MASKING` is set in the "options" list of the DPNI object either via the DPL node or via restool - e.g.:

```
dpni@1 {
[…]
    options = "DPNI_OPT_HAS_KEY_MASKING";
[…]
};
```

respectively:

```
restool dpni create [...] --options=DPNI_OPT_HAS_KEY_MASKING
```

The examples in this paragraph are using dpni@1/n1 as reference because at Linux boot, the default ni0 interface enabled via the static DPL does not declare Rx Flow Steering in its capabilities list, so Rx Flow Steering would not be available on ni0 unless it is dynamically destroyed and re-created. For full details and examples on dynamic DPNI creation using *restool*, refer to this chapterDynamically creating a DPNI on page 444. The interfaces created using those examples do support Rx Flow Steering.

6.6.4.11 Flow Control Pause Frames

The DPAA2 Ethernet interfaces support sending and responding to pause frames, as part of the Ethernet flow control mechanism. The behavior of the pause frames is described in the IEEE 802.3x standard. In a nutshell, in a scenario involving a full duplex link, if the sender is sending at a higher rate than the receiver can process frames, the receiver can choose to send a special kind of frame, called pause frame, which asks the sender to halt the transmission of traffic for a specified period of time.

Pause frame control is integrated into ethtool. By default it's enabled for both Tx and Rx.

```
~# ethtool -a ni0
Pause parameters for ni0:
Autonegotiate:  on
RX:             on
TX:             on

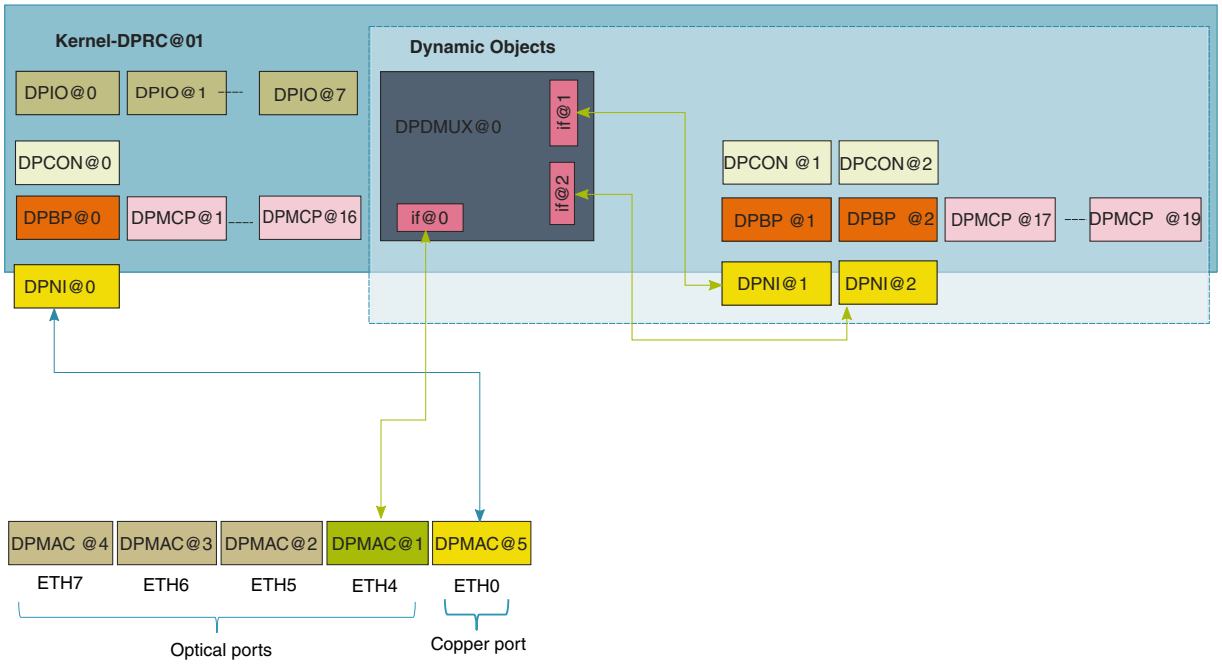
~# ethtool -A ni0 rx off
~# ethtool -a ni0
Pause parameters for ni0:
Autonegotiate:  on
RX:             off
TX:             on
```

Currently, there's no support in configuring pause frame autonegotiation independently from link general autonegotiation. Hence, if link autonegotiation (e.g. rate and duplex) is on, it is on pause frames as well, and viceversa.

Pause frames are interpreted at the MAC level. Therefore, the counters for pause frames are visible when configuring netdevice objects for DPMAC objects using the `CONFIG_FSL_DPAA2_MAC_NETDEVS=y` kernel option. This option will bring up a set of additional Linux network interfaces named `macX`, one for each probed DPMAC object in the system.

```
~# ethtool -S mac1 | grep pause
     rx pause: 106731474
     tx b-pause: 15287253
```

If the driver is configured with Tx pause frames on, the hardware will start sending pause frames when the interface enters a congestion state on the Rx side.

If the driver is configured with Rx pause frames on, it will respond to any pause frames received on the line by reducing the send rate.

6.6.4.12 Busy Poll

Busy polling is a mechanism to reduce the latency on the network receive path. When enabled, it allows the socket layer code to directly poll the receive queue of a network device. During this time, network interrupts are disabled, thus removing delays caused by interrupt handling and context switches. The side effect is an increased CPU utilization.

Busy polling is disabled by default. There are 2 requirements to enable it:

- Add the `SO_BUSY_POLL` option to the socket. This can be done in the userspace application, per socket. In order to enable it globally, set `sysctl.net.core.busy_read` to a value greater than 0. This parameter controls the number of microseconds to wait for packets on the device queue for socket reads. It also sets `SO_BUSY_POLL` to all created sockets by default.

  ```
  ~# sysctl -w net.core.busy_read=50
  ```

- Set `sysctl.net.core.busy_poll` to a value greater than 0. This parameter controls the number of microseconds to wait for packets on the device queue socket poll and select.

  ```
  ~# sysctl -w net.core.busy_poll=50
  ```

Checking whether the driver supports busy polling is done via ethtool:

```
~# ethtool -k ni0 | grep busy-poll
busy-poll: on [fixed]
```

6.6.4.13 Interface statistics

DPAA2 Ethernet interface counters can be read via either of two standard tools, but there is a subtle difference:

- " `ifconfig ni0` " counters reflect packets received by the Ethernet driver – i.e. those frames that have passed through the Rx filters (if any are active) and have been effectively processed by the Ethernet driver on the GPP, and possibly by the kernel stack. These are software counters, maintained by the Ethernet driver and the networking stack.
- " `ethtool -S ni0` " counters reflect more detailed counters, from two categories:
  1. Statistics maintained by the DPAA2 hardware. These largely correspond in meaning to the standard "ifconfig" counters, but the values may be different from the "ifconfig" counters - i.e. they may reflect frames that have not been received on the GPP, such as those dropped by the ingress policer or due to MAC filtering. Also noteworthy is that retrieving these counters requires a series of calls into the MC firmware, which could make the operation potentially slower.
  2. Advanced counters, specific to the DPAA2 Ethernet driver. These are software-maintained driver-specific counters which do not fit into the standard " `ifconfig` " set.

     The following detailed counters are presented by the '`ethtool -S`' command:

     a. Hardware-maintained counters:
        - `rx frames`: number of valid frames presented to the DPNI hardware and not dropped (see "rx frames dropped" below);
        - `rx bytes`: number of bytes comprised within the "rx frames" counter;
        - `rx frames dropped`: number of valid frames but dropped because e.g. of MAC filtering;
        - `rx err frames`: number of frames with various physical errors; frames dropped because of insufficient buffers to DMA them into are counted here;
        - `rx mcast frames`: number of valid multicast frames;
        - `rx mcast bytes`: number of bytes included in "rx mcast frames";
        - `rx bcast frames`: number of valid broadcast frames;
        - `rx bcast bytes`: number of bytes included in "rx bcast frames";
        - `tx frames`: number of valid frames presented for transmission;
        - `tx bytes`: number of bytes included in "tx frames";
        - `tx err frames`: number of frames with Tx errors;

     b. Software-maintained, driver-specific counters:
        - `tx conf frames`: number of frames presented back to the Ethernet driver in the Tx confirmation queues. In an idle system, this counter should be equal to "tx frames";
        - `tx conf bytes`: number of bytes comprised by the "tx conf frames" counter;
        - `tx sg frames`: number of egress frames in scatter-gather format; these are a subset of "tx frames", the difference being contiguous frames;
        - `tx sg bytes` : number of bytes comprised in "tx sg frames";
        - `rx sg frames`: number of frames received in scatter-gather format; typically this reflects frames larger than the largest buffer that can be used at the time of reception;
        - `rx sg bytes`: number of bytes comprised in "rx sg frames";

- `tx portal busy`: number of times the Ethernet driver had to retry the frame enqueue command (on the egress path) due to QBMan portal being busy;
- `portal busy`: number of times the Ethernet driver had to retry the frame dequeue command (on the ingress path) due to QBMan portal being busy;
- `cdan` : number of Channel Dequeue Available Notifications (CDANs) received by the Ethernet driver (Rx and Tx Conf paths). Each CDAN corresponds to one DPIO interrupt and triggers a NAPI processing cycle which can process Rx or Tx Conf frames (or both).

6.7 Setting up Ethernet Switch Capability

6.7.1 Ethernet Switch overview

The following switch features are supported:

- Dynamic learning
- Adding/deleting static FDB entries
- Adding/deleting static multicast entries
- Configuring multicast groups
- Flooding of broadcast and multicast traffic
- Forwarding of unicast traffic that is both VLAN tagged and untagged
- Setting STP state of ports

Important notes:

- Learning is supported and enabled by default.
- Learned FDB entries cannot be displayed.
- There is no support to flush the FDB.

Acronyms and abbreviations:

- DPSW - DPAA2 object modelling an L2 Switch
- DPNI - DPAA2 object modelling a network interface

6.7.2 Switch object creation

A switch object can be created:

- Dynamically using the restool as described in Using restool for dynamic object creation on page 453 or
- Statically in a DPL file as described in Using the data path layout file (DPL) on page 455

6.7.2.1 Using restool for dynamic object creation

A switch can be created at run-time, using restool. Before creating the switch, a number of DPAA2 objects (dependencies) have to be added, for which multiple restool commands are needed. Switch requires at least a DPMCP object, which is created like:

```
$ restool dpmcp create
```

The following section describes the main commands to create a switch starting from the **dpl-eth.0x2A_0x41.dtb** DPL file.

Figure 29. Dynamic DPSW demo



NOTE

When a new object is created via restool, an object with the id of the first available resource will be returned.

NOTE

Depending on the board type, DPMAC availability varies. For more details please refer to **Limitations and Known Issues**.

6.7.2.1.1 Creating a DPSW

The switch object is created by this command:

```
$ restool dpsw create --num-ifs=5 --max-vlans=16 --max-fdbs=1 --options="DPSW_OPT_CTRL_IF_DIS"
```

The command specifies some configuration options, including the number of ports in the switch, the maximum number of VLANs that can be used on the switch – including VLAN1 used implicitly, plus the number of FDBs.

For all the configuration options and parameters see the help output:

```
$ restool dpsw create -h
```

Currently VLAN private FDBs are not supported; a single shared FDB is used. Also control traffic is not supported, so option DPSW_OPT_CTRL_IF_DIS has to be specified.

6.7.2.1.2 Connecting the switch

Linking the switch ports to other objects is done with:

```
restool dprc connect "$RC" --endpoint1="$SW".0 --endpoint2=dpmac.1
restool dprc connect "$RC" --endpoint1="$SW".1 --endpoint2=dpmac.2
restool dprc connect "$RC" --endpoint1="$SW".2 --endpoint2=dpmac.3
restool dprc connect "$RC" --endpoint1="$SW".3 --endpoint2=dpmac.4
restool dprc connect "$RC" --endpoint1="$SW".4 --endpoint2="$NI"
```

In the context of the configuration script, these commands create the following layout:

- sw0p0 (dpsw@1/if@0) <-> dpmac@1 (SFP+ port, labeled ETH4 on RDB front panel)
- sw0p1 (dpsw@1/if@1) <-> dpmac@2 (SFP+ port, labeled ETH5 on RDB front panel)
- sw0p2 (dpsw@1/if@2) <-> dpmac@3 (SFP+ port, labeled ETH6 on RDB front panel)
- sw0p3 (dpsw@1/if@3) <-> dpmac@4 (SFP+ port, labeled ETH7 on RDB front panel)
- sw0p4 (dpsw@1/if@4) <-> dpni@1

For SerDes **0x2A_0x41**, DPMACs 1-4 are mapped to the optical PHYs while DPMACs 5-8 are mapped to the copper PHYs. User can choose to connect any of the optical or copper ports and any NIs to the switch.

6.7.2.1.3 Enabling the switch

This command plugs the switch object on the bus, in the Linux resource container. The switch driver probes the switch and presents the associated network interfaces in Linux.

```
$ restool dprc assign "$RC" --object="$SW" --plugged=1
```

After enabling the switch it can be configured from Linux using the commands specified in Commands supported on page 457.

6.7.2.1.4 Restool wrapper scripts

For user convenience the **ls-addsw** script is provided to assist creation of a new DPSW object.

To replicate the setup described in sectionConnecting the switch on page 454 the following commands are required:

```
$ ls-addni -n
$ ls-addsw -i=5 dpmac.1 dpmac.2 dpmac.3 dpmac.4 dpni.1
```

The first command creates a new DPNI object and the second the DPSW object. DPMACs are already defined in the DPL file. To display the DPNIs and DPMACs available one can use *ls-listni* or *ls-listmac* commands. The DPIO, DPBP, DPCON and DPMCP objects that are dependencies for the new objects are created by the script, without user intervention.

For all the script options and parameters see the help:

```
$ ls-addsw -h
```

The endpoints are connected in the specified order to switch ports. If there are less endpoints than the number of interfaces, the user can later add the rest using *ls-addni* or *restool* commands.

6.7.2.2 Using the data path layout file (DPL)

A switch object may be defined statically in the DPL, allowing it to be created automatically during platform initialization. Below is an example of switch definition in the DPL:

```
dpsw@0 {
      compatible = "fsl,dpsw";
      options = "DPSW_OPT_CTRL_IF_DIS";
      max_vlans = <0x10>;
      max_fdbs = <0x1>;
      num_fdb_entries = <0x400>;
      fdb_aging_time = <0x12c>;
      num_ifs = <0x5>;
      max_fdb_mc_groups = <32>;
};
```

This example is for a 5-port switch that includes support for up to 16 VLAN IDs, including VLAN 1 (that is internally used by the switch), up to 1024 FDB entries, and up to 32 multicast groups.

Links are defined in the DPL 'connections' section as follows:

```
connections {
       connection@1 {
             endpoint1 = "dpsw@0/if@0";
             endpoint2 = "dpmac@1";
        };
        connection@2 {
             endpoint1 = "dpsw@0/if@1";
             endpoint2 = "dpmac@2";
        };
        connection@3 {
             endpoint1 = "dpsw@0/if@2";
             endpoint2 = "dpmac@3";
        };
        connection@4 {
            endpoint1 = "dpsw@0/if@3";
            endpoint2 = "dpmac@4";
        };
        connection@5 {
             endpoint1 = "dpsw@0/if@4";
             endpoint2 = "dpni@1";
       };
};
```

The generated layout is the one described in Connecting the switch on page 454.

6.7.3 Setting up the driver

To compile the driver, you enable the **FSL_DPAA2_ETHSW** option in the kernel's config. It can be found in *menuconfig* under the following items:

```
| -> Device Drivers
|   -> Network device support
|     -> Ethernet driver support
|       -> Freescale devices
|         -> Freescale DPAA Ethernet
|           -> Layerscape DPAA Ethernet Switch
```

The driver is enabled in the default kernel configuration file. The driver registration is signaled in the kernel log by this message:

```
MC object device driver dpaa2_ethsw registered
```

After deploying the driver and instantiating a DPSW, the system should present following Linux interfaces (actual number of interfaces depends on the specified number of switch ports):

```
sw0         Link encap:Ethernet HWaddr 00:00:00:00:00:00
            inet6 addr: fe80::200:ff:fe00:0/64 Scope:Link
            UP BROADCAST RUNNING MASTER MULTICAST MTU:1500 Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

```
sw0p0      Link encap:Ethernet HWaddr 00:00:00:00:00:00
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

sw0p1      Link encap:Ethernet HWaddr 00:00:00:00:00:00
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

sw0p2      Link encap:Ethernet HWaddr 00:00:00:00:00:00
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

sw0p3      Link encap:Ethernet HWaddr 00:00:00:00:00:00
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

sw0p4      Link encap:Ethernet HWaddr 00:00:00:00:00:00
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

There is one ***sw0pX*** interface for each switch port, plus one ***sw0*** master interface that provides control over the whole switch. All these interfaces are used to manage the switch and its ports; they cannot be used to perform I/O. Any I/O through the switch must be performed using the interfaces linked to the switch.

6.7.4 Commands supported

The switch management commands supported are:

- ifup/ifdown (using ifconfig or similar)
- setting large frame size limit (using ifconfig or similar)
- retrieving statistics (using ifconfig or similar)
- configuring FDB (using bridge fdb)
- configuring multicast groups (using bridge fdb)
- configuring VLANs (using bridge vlan)
- configuring learning (using bridge link set)

6.7.4.1 Interface control

Any of the switch ports, or the switch as a whole, can be enabled/disabled using any of the following commands:

```
$ ifconfig { sw0pX | sw0 } { up | down }
$ ip link set { sw0pX | sw0 } { up | down }
```

6.7.4.2 Maximum frame size configuration

The DPAA2 hardware supports large frames. Ethernet switch driver correlates between the Layer-2 maximum frame length (MFL) and Layer-3 MTUs. The maximum MTU that a Linux user can request on a DPAA2 Ethernet switch interface is 10218 bytes and has to be set on each port individually.

```
$ ifconfig <sw0pX> mtu <NN>
$ ip link set { sw0pX | dev sw0pX } mtu <NN>
```

Notes:

- Frames larger than the configured MTU will be dropped, so connected Ethernet devices need to have the same setting.
- All Ethernet devices on the same LAN must have the same MTU to avoid traffic loss.

6.7.4.3 Learning control

The switch is set by default to enable learning, and the learning can be controlled using this command:

```
$ bridge link set dev sw0p0 learning { on | off }
```

NOTE

The command is executed on a switch port, although it does affect the learning function at switch level. Turning off learning does not remove the learned entries.

To establish a static topology, learning should be disabled before injecting any traffic.

6.7.4.4 FDB static entries

The default switch configuration does not include any static entries; these can be added using the *bridge fdb add* command, as shown below:

```
$ bridge fdb add 00:00:05:00:00:11 dev sw0p0
```

This command adds the MAC address of ni0 as a static entry in the FDB on the first switch port, linked to ni0.

6.7.4.5 Multicast entries

Multicast groups can be configured via bridge `fdb add|append|delete` commands:

```
$ bridge fdb add 01:00:05:00:00:13 dev sw0p1
$ bridge fdb append 01:00:05:00:00:13 dev sw0p2
$ bridge fdb append 01:00:05:00:00:13 dev sw0p3
$ bridge fdb append 01:00:05:00:00:13 dev sw0p4

$ bridge fdb del 01:00:05:00:00:13 dev sw0p4
```

The commands add all external ports, one by one, to the `01:00:05:00:00:13` multicast group. The last command removes the last port from the group.

6.7.4.6 VLAN configuration

The switch starts up with VLAN 1 configured as the default VLAN. All untagged traffic received on any switch port is classified to VLAN 1, and all frames classified in VLAN 1 are sent out untagged on all ports. Additional VLANs can be added on the switch using these commands:

```
$ bridge vlan add vid 2 dev sw0p0
$ bridge vlan add vid 2 dev sw0p1
$ bridge vlan add vid 2 dev sw0p2
$ bridge vlan add vid 2 dev sw0p3
$ bridge vlan add vid 2 dev sw0p4
```

This example includes all five ports in VLAN 2. After running these commands the switch should allow frames tagged with vid 2 to pass through the switch; not all ports have to be included in the VLAN.

To remove a port from a given VLAN use the following command:

```
$ bridge vlan del vid 2 dev sw0p0
```

To remove a VLAN completely from the switch run the following command:

```
$ bridge vlan del vid 2 dev sw0 self
```

6.7.4.7 Port statistics

Switch port statistics are available through `ip` or similar tools:

```
$ ip -s link
[...]
6: sw0: <NO-CARRIER,NOARP,MASTER,UP> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen
1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes packets errors dropped overrun mcast
    0         0       0      0       0       0
    TX: bytes packets errors dropped carrier collsns
    0         0       0      0       0       0
7: sw0p0: <NO-CARRIER,NOARP,SLAVE,UP> mtu 1500 qdisc noop master ethsw state DOWN mode DEFAULT group
default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes packets errors dropped overrun mcast
    500       6       0      0       0       0
    TX: bytes packets errors dropped carrier collsns
    2894      28      0      1       0       0
8: sw0p1: <NO-CARRIER,NOARP,SLAVE,UP> mtu 1500 qdisc noop master ethsw state DOWN mode DEFAULT group
default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes packets errors dropped overrun mcast
    0         0       0      0       0       0
    TX: bytes packets errors dropped carrier collsns
    0         0       0      0       0       0
9: sw0p2: <NO-CARRIER,NOARP,SLAVE,UP> mtu 1500 qdisc noop master ethsw state DOWN mode DEFAULT group
default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes packets errors dropped overrun mcast
    9504      63      0      0       0       0
    TX: bytes packets errors dropped carrier collsns
    476       6       0      0       0       0
10: sw0p3: <NO-CARRIER,NOARP,SLAVE,UP> mtu 1500 qdisc noop master ethsw state DOWN mode DEFAULT
group default qlen 1000
```

```
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes packets errors dropped overrun mcast
    0         0       0      0       0       0
    TX: bytes packets errors dropped carrier collsns
    5910      51      0      8       0       0
[...]
```

6.8 Setting Up Edge Virtual Bridge Capability

6.8.1 EVB overview

An edge virtual bridge allows the sharing of a physical connection between multiple entities (virtual hosts). It can act as a VEB or as a VEPA.

In VEB mode, traffic is forwarded between connected virtual hosts or between virtual hosts and uplink.

In VEPA mode, all traffic from virtual hosts is forwarded to uplink, bridging functions (including 'hairpin' forwarding) being performed by an external device.

Features supported:

- VEB/VEPA mode
- Traffic steering according to MAC, VLAN (in VEPA mode only) or MAC+VLAN
- Static FDB entries management (add/delete/show)
- Static multicast FDB entries management (add/delete/show)
- Flooding of broadcast and multicast traffic

6.8.2 EVB object creation

EVB objects can be created as follows:

- Dynamically using the restool as described in Using restool for dynamic object creation on page 460
- Statically in a DPL file as described in Using the data path layout file (DPL) on page 462

6.8.2.1 Using restool for dynamic object creation

A DPDMUX can be instantiated at run-time, using restool.

The following section describes the main commands to create an EVB and its dependencies starting from the **dpl-eth.0x2A_0x41.dtb** DPL file.



Figure 30. Dynamic DPDMUX demo

NOTE

When a new object is created using restool, an object with the ID of the first available resource is returned.

NOTE

Depending on the board type, DPMAC availability varies. For more details please refer to **Limitations and Known Issues**.

6.8.2.1.1 Creating a DPDMUX

The EVB is created by this command:

```
$ restool dpdmux create --num-ifs=2 --control-if=0                        \
          --options=DPDMUX_OPT_BRIDGE_EN --method=DPDMUX_METHOD_MAC       \
          --max-dmat-entries=8 --max-mc-groups=8 --manip=DPDMUX_MANIP_NONE
```

The command must specify the number of downlinks and the ID of the uplink (ranges for 0 to [number of downlinks -1]). The other parameters are optional. For more information about the available options see the output of the command:

```
$ restool dpdmux create -h
```

6.8.2.1.2 Connecting the EVB

Linking the EVB ports to other objects is done with:

```
$ restool dprc connect "$RC" --endpoint1="$MUX".0 --endpoint2="$MAC"
$ restool dprc connect "$RC" --endpoint1="$MUX".1 --endpoint2="$NI1"
$ restool dprc connect "$RC" --endpoint1="$MUX".2 --endpoint2="$NI2"
```

$RC represents the container for the objects, $MUX is the object identified for the EVB. The uplink is the endpoint with the id specified by *control-if* parameter at creation time.

6.8.2.1.3 Enabling the EVB

This command plugs the EVB object on the bus, in the Linux resource container. The EVB driver probes the switch and presents the associated network interfaces in Linux.

```
$ restool dprc assign "$RC" --object="$MUX" --plugged=1
```

6.8.2.1.4 Restool wrapper scripts

For user convenience the **ls-addmux** script is provided to assist creation of a new DPDMUX object.

Example to replicate setup in section Connecting the EVB on page 462 :

```
# ls-addmux -d=2 -u=0 dpmac.1
[ 4298.023745] dpaa2_evb dpdmux.0: probed evb device with 2 ports
Created EVB: evb0 (object: dpdmux.0, uplink: dpmac.1)
```

This command creates EVB evb0 (and the corresponding dpdmux.0 object) with two downlinks and the uplink connected to dpmac.1.

After creating the DPDMUX, its downlinks can be connected to DPNIs using **ls-addni** script:

```
# ls-addni dpdmux.0.1
Will allocate 8 DPCON objects for this hash size
[ 5118.645253] fsl_dpaa2_eth dpni.1: Probed interface ni1
Created interface: ni1 (object:dpni.1, endpoint: dpdmux.0.1)
# ls-addni dpdmux.0.2
Will allocate 8 DPCON objects for this hash size
[ 5122.169030] fsl_dpaa2_eth dpni.2: Probed interface ni2
Created interface: ni2 (object:dpni.2, endpoint: dpdmux.0.2)
```

6.8.2.2 Using the data path layout file (DPL)

A DPDMUX instance can statically be defined in the DPL file:

```
dpdmux@0 {
       compatible = "fsl,dpdmux";
       options = "DPDMUX_OPT_BRIDGE_EN";
       method = "DPDMUX_METHOD_MAC";
       manip = "DPDMUX_MANIP_NONE";
       control_if = <0>;
       num_ifs = <2>;
       max_dmat_entries = <8>;
       max_mc_groups = <8>;
};
```

Links are defined in the DPL 'connections' section:

```
connection@1{
        endpoint1 = "dpdmux@0/if@0";
        endpoint2 = "dpmac@1";
};
connection@2{
        endpoint1 = "dpni@1";
        endpoint2 = "dpdmux@0/if@1";
};
connection@3{
        endpoint1 = "dpni@2";
        endpoint2 = "dpdmux@0/if@2";
};
```

Based on the above configuration the DPDMUX ports are linked to:

- evb0 (dpdmux@1/if@0) <-> dpmac@1
- evb0p0 (dpdmux@1/if@1) <-> dpni@1
- evb0p1 (dpdmux@1/if@2) <-> dpni@2

NOTE

DPDMUX ports connected to a DPMAC must be configured before the others (e.g. connected to DPNIs).

6.8.3 Setting up the EVB driver

Driver compilation is enabled by default and is controlled by the FSL_DPAA2_EVB option in the kernel's config. This can be found in *menuconfig* under the following items:

```
| -> Device Drivers
|     -> Staging drivers
|         -> Freescale Management Complex (MC) bus driver
|             -> Freescale DPAA2 devices
|                 -> DPAA2 Edge Virtual Bridge
```

The kernel log will display a message when an EVB is probed as follows:

```
dpaa2_evb dpdmux.0: probed evb device with 2 ports
```

After deploying the driver and configuring an EVB (via DLP or restool), the system should present the following Linux interfaces after typing the 'ifconfig command':

```
evb0     Link encap:Ethernet HWaddr 00:00:00:00:00:00
         UP BROADCAST RUNNING MASTER MULTICAST MTU:1500 Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

evb0p0   Link encap:Ethernet HWaddr 00:00:00:00:00:00
         UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

```
evb0p1   Link encap:Ethernet HWaddr 00:00:00:00:00:00
         UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

Interface ***evb0*** represents the uplink and is also the handler for the EVB. Each other EVB port has its own interface. They are used for management and cannot be used for I/O. Any I/O through the EVB must be performed using the connected interfaces.

An EVB only forwards traffic to links that are enabled (peer interface is up) and only if the filtering rules on the peer interface do not lead to the frame being discarded. One way to ensure that all traffic subject to forwarding rules is actually forwarded by the EVB is to set the peer interface in promiscuous mode, as follows:

```
$ ip link set ni0 up promisc on
```

6.8.4 EVB commands supported

EVB management can be performed using the following generic Linux networking tools:

- interface up/down (using ifconfig or similar)
- configuring FDB (using bridge fdb)
- configuring VLANs (using bridge vlan)
- configuring multicast groups (using bridge fdb)
- port statistics retrieval (ethtool or similar)

6.8.4.1 EVB interface control

Any of the EVB ports, or the EVB as a whole, can be enabled/disabled using any of the following commands:

```
$ ifconfig { evb0pX | evb0 } { up | down }
$ ip link set { evb0pX | evb0 } { up | down }
```

6.8.4.2 EVB FDB entries

The EVB method DPDMUX_METHOD_MAC allows configuration of FDB entries via a bridge utility as follows:

```
$ bridge fdb add 02:00:c0:a8:50:01 dev evb0p0
$ bridge fdb show
02:00:c0:a8:50:01 self permanent
01:00:5e:00:00:01 self permanent
```

The EVB method DPDMUX_METHOD_C_VLAN_MAC also allows configuration of FDB entries via a bridge utility as follows:

```
$ bridge fdb add 02:00:c0:a8:50:02 vlan 10 dev evb0p0 vlan 10
$ bridge fdb show dev evb0p0
02:00:c0:a8:50:02 self permanent
01:00:5e:00:00:01 self permanent
```

6.8.4.3 EVB VLAN assignment

The EVB method DPDMUX_METHOD_C_VLAN allows port VLAN assignment via a bridge utility as follows:

```
$ bridge vlan add vid 10 dev evb0p2
$ bridge vlan show dev evb0p2
port vlan ids
evb0p2 10
$ bridge vlan del vid 10 dev evb0p2
```

NOTE

This method is allowed only for VEPA mode.

6.8.4.4 EVB port statistics

EVB port statistics are available through *ip* or similar tools as follows:

```
$ ip -s link
[...]
9: evb0: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode
DEFAULT group default qlen 1000
       link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
       RX: bytes packets errors dropped overrun mcast
       0         0       0      0       0       0
       TX: bytes packets errors dropped carrier collsns
       384       6       0      0       0       0
10: evb0p0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master evb0 state
UNKNOWN mode DEFAULT group default qlen 1000
       link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
       RX: bytes packets errors dropped overrun mcast
       252       6       0      0       0       0
       TX: bytes packets errors dropped carrier collsns
       0         0       0      0       0       0
[...]
```

6.8.5 Forwarding methods overview

A DPAA2 DPDMUX instance can forward traffic using information from various fields in the frame headers:

- Forwarding by destination MAC address
- Forwarding by VLAN tag
- Forwarding by VLAN tag and destination MAC adress

6.8.5.1 Forwarding by destination MAC address

This method forwards frames according to the destination MAC address and the static rules added into the EVB forwarding database.

It is configured specifying *--method="DPDMUX_METHOD_MAC"* when the DPDMUX is created. It is the default value for the *ls-addmux* script.

Entries are configured in the FDB using *bridge fdb* command. See EVB FDB entries on page 464 section for more information.

Configuration example:

```
# Create a MUX with 2 downlinks and uplink connected to dpmac.1;
# forwarding method is by default DPDMUX_METHOD_MAC
$ ls-addmux -b -d=2 -u=0 dpmac.1
```

```
# Create a ni (dpni.1) and links it to evb0p0
$ ls-addni dpdmux.0.1

# Create a ni (dpni.2) and links it to evb0p1
$ ls-addni dpdmux.0.2

# Check MUX configuration
# $ restool dpdmux info dpdmux.0

# Configure ni1
$ ip netns add ns1
$ ip link set ni1 netns ns1
$ ip netns exec ns1 ifconfig ni1 192.168.10.10/24 up
$ ip netns exec ns1 ip link set ni1 promisc on

# Configure ni2
$ ip netns add ns2
$ ip link set ni2 netns ns2
$ ip netns exec ns2 ifconfig ni2 192.168.10.12/24 up
$ ip netns exec ns2 ip link set ni2 promisc on

# Connectivity checks [downlink - uplink ]
$ ip netns exec ns1 ping 192.168.10.13 -c 1
[..]
--- 192.168.10.13 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms

# Check EVB port statistics
$ ip -s link
[...]
4: evb0: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode
DEFAULT group default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    436        6        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    460        6        0       0       0       0
5: evb0p0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master evb0 state UP
mode DEFAULT group default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    364        6        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    376        5        0       0       0       0
6: evb0p1: <NO-CARRIER,BROADCAST,MULTICAST,SLAVE,UP> mtu 1500 qdisc pfifo_fast master evb0 state
DOWN mode DEFAULT group default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    0          0        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    0          0        0       0       0       0
```

6.8.5.2 Forwarding by VLAN tag

This method forwards frames according to the VLAN tag of the frame, as set into the customer tag of the double-tagged frames.

It is configured specifying *--method="DPDMUX_METHOD_C_VLAN"* when EVB is in VEPA mode (*--options="DPDMUX_OPT_BRIDGE_EN"* is not set).

EVB port VLAN assignment is done with "*bridge vlan* command. See EVB VLAN assignment on page 465 section for more information.

Configuration example:

```
# Create a MUX with DPDMUX_METHOD_C_VLAN forwarding method,
# configured as a VEPA and with 2 downlinks and uplink connected
# to dpmac.1
$ ls-addmux -m=DPDMUX_METHOD_C_VLAN -d=2 -u=0 dpmac.1

# Create a ni (dpni.1) and link it to evb0p0
$ ls-addni dpdmux.0.1

# Create a ni (dpni.2) and link it to evb0p1
$ ls-addni dpdmux.0.2

# Configure ni1
$ ip netns add ns1
$ ip link set ni1 netns ns1
$ ip netns exec ns1 ip link add link ni1 name ni1.6 type vlan id 6
$ ip netns exec ns1 ifconfig ni1.6 192.168.6.10
$ ip netns exec ns1 ip link set ni1 up
$ ip netns exec ns1 ip link set ni1 promisc on

# Configure ni2
$ ip netns add ns2
$ ip link set ni2 netns ns2
$ ip netns exec ns2 ip link add link ni2 name ni2.7 type vlan id 7
$ ip netns exec ns2 ifconfig ni2.7 192.168.7.12
$ ip netns exec ns2 ip link set ni2 up
$ ip netns exec ns2 ip link set ni2 promisc on

# For the downlinks interfaces also add the VLAN ids
$ bridge vlan add vid 6 dev evb0p0
$ bridge vlan add vid 7 dev evb0p1

# Connectivity checkings [example for downlink - uplink ]
$ ip netns exec ns1 ping -I ni1.6 192.168.6.13 -c 1

# Check VLAN assignment
$ bridge vlan show
```

6.8.5.3 Forwarding by VLAN tag and destination MAC adress

This method forwards frames according to the VLAN tag and the destination MAC address of the frame .

It is configured specifying *--method="DPDMUX_METHOD_C_VLAN_MAC"* when the DPDMUX is created.

Entries are configured in the FDB using *bridge fdb* command. See EVB FDB entries on page 464 section for more information.

Configuration example:

```
# Create a MUX with DPDMUX_METHOD_C_VLAN_MAC forwarding method,
# configured as a VEB and with 2 downlinks and uplink connected
# to dpmac.1
$ ls-addmux -b -m=DPDMUX_METHOD_C_VLAN_MAC -d=2 -u=0 dpmac.1

# Create a ni (dpni.1) and link it to evb0p0
$ ls-addni dpdmux.0.1

# Create a ni (dpni.2) and link it to evb0p1
$ ls-addni dpdmux.0.2

# Configure ni1
$ ip netns add ns1
$ ip link set ni1 netns ns1
```

```
$ ip netns exec ns1 ip link add link ni1 name ni1.6 type vlan id 6
$ ip netns exec ns1 ifconfig ni1.6 192.168.6.10
$ ip netns exec ns1 ip link set ni1 up
$ ip netns exec ns1 ip link set ni1 promisc on

# Configure ni2
$ ip netns add ns2
$ ip link set ni2 netns ns2
$ ip netns exec ns2 ip link add link ni2 name ni2.7 type vlan id 7
$ ip netns exec ns2 ifconfig ni2.7 192.168.7.12
$ ip netns exec ns2 ip link set ni2 up
$ ip netns exec ns2 ip link set ni2 promisc on

# For the downlinks interfaces, you would also need to add
# the downlinks MACs to fdb table
$ bridge fdb add 4a:64:0a:af:14:a2 dev evb0p0 vlan 6
$ bridge fdb add 62:9c:86:0f:f7:cf dev evb0p1 vlan 7

# Connectivity checkings [example for downlink - uplink ]
$ ip netns exec ns1 ping -I ni1.6 192.168.6.13 -c 1

# Check EVB FDB entries
$ bridge fdb show
```

6.9 Security Engine (SEC)

This paragraph describes the software for the SEC hardware block that is part of the DPAA2 family of SoCs.

6.9.1 DPAA2 CAAM driver specifics

Module Loading

The dpaa2-caam driver is built as a module that is registered when a `dpseci` object is created dynamically with `restool`. Without any parameter, the DPSECI object being created has 2 pairs of (rx,tx) queues.

```
# restool dpseci create && restool dprc assign dprc.1 --object=dpseci.0 --plugged=1
```

To create 8 (maximum) number of queues:

```
restool dpseci create --num-queues=8 --priorities=1,2,3,4,5,6,7,8 && restool dprc assign dprc.1 --
object=dpseci.0 --plugged=1
```

You can check more options:

```
restool dpseci create --help
```

You can check the list of algorithms registered by the dpaa2-caam driver as below:

```
# grep dpaa2-caam /proc/crypto
```

Enabling congestion management

Congestion management can be enabled when working with a MC that has a DPSECI object version greater or equal to 5.1. The first MC that will feature congestion management will have version 10.3. Enabling congestion management is done while creating the DPSECI object:

```
restool dpseci create --num-queues=8 --priorities=1,2,3,4,5,6,7,8 --options="DPSECI_OPT_HAS_CG" &&
restool dprc assign dprc.1 --object=dpseci.0 --plugged=1
```

Please note that this release features a MC with the version 10.2 and therefore it does not support congestion management activation. To use congestion management please use a newer version for both MC and restool. Using the current release restool with the above command will result in error.

Kernel configuration

Kernel Configure Tree View Options	Description
-- Cryptographic API [*] Hardware crypto devices -->> <*> Freescale DPAA2 CAAM ---	**Enables dpaa2 caam device driver**

Source files

The driver source files are maintained in the Linux kernel source tree: `drivers/crypto/dpaa2-caam`.

Features overview

The following is an overview of the functionalities of the Linux DPAA2 CAAM SEC driver:

Authenticated encryption with associated data (AEAD) algorithms

These algorithms are used in applications where the data to be encrypted overlaps, or partially overlaps, the data to be authenticated, as is the case with the IPSec protocol. These algorithms are implemented in the driver such that the hardware makes a single pass over the input data, and both encryption and authentication data are written out simultaneously. Note that, on decryption, integrity verification is performed in h/w, when available. The AEAD algorithms are mainly for use with IPSec ESP. The CAAM driver currently supports offloading the following AEAD algorithms:

authenc(hmac(sha512), rfc3686(ctr(aes)))
authenc(hmac(sha384), rfc3686(ctr(aes)))
authenc(hmac(sha256), rfc3686(ctr(aes)))
authenc(hmac(sha224), rfc3686(ctr(aes)))
authenc(hmac(sha1), rfc3686(ctr(aes)))
authenc(hmac(md5), rfc3686(ctr(aes)))
authenc(hmac(sha512), cbc(des)))
authenc(hmac(sha384), cbc(des)))
authenc(hmac(sha256), cbc(des)))
authenc(hmac(sha224), cbc(des)))
authenc(hmac(sha1), cbc(des)))
authenc(hmac(md5), cbc(des)))
authenc(hmac(sha512), cbc(des3_ede)))
authenc(hmac(sha384), cbc(des3_ede)))
authenc(hmac(sha256), cbc(des3_ede)))
authenc(hmac(sha224), cbc(des3_ede)))
authenc(hmac(sha1), cbc(des3_ede)))
authenc(hmac(md5), cbc(des3_ede)))
authenc(hmac(sha512), cbc(aes)))
authenc(hmac(sha384), cbc(aes)))
authenc(hmac(sha256), cbc(aes)))
authenc(hmac(sha224), cbc(aes)))
authenc(hmac(sha1), cbc(aes)))
authenc(hmac(md5), cbc(aes)))
gcm(aes)

i.e., all combinations of AES-CBC, (3)DES-EDE, RFC3686-CTR-AES with MD-5, SHA-1,-224,-256,-384, and -512.

Cipher encryption algorithms

The CAAM driver currently supports offloading the following encryption algorithms:

rfc3686(ctr(aes)
ctr(aes)
cbc(des)
cbc(3des)
cbc(aes)

Authentication algorithms

CAAM driver support both keyed and unkeyed hash algoritms, as follows:

hmac(md5)
hmac(sha1)
hmac(sha512)
hmac(sha384)
hmac(sha256)
hmac(sha224)
sha1
sha384
sha256
sha224
sha512
md5

How to test the driver

To test the driver, in the kernel configuration menu, under `"Cryptographic API -> Cryptographic algorithm manager"`, ensure that run-time self-tests are not disabled, i.e. the "`Disable run-time self tests`" (`CONFIG_CRYPTO_MANAGER_DISABLE_TESTS`) entry is not set. This will run standard test vectors against the driver after the driver registers its supported algorithms with the kernel crypto API. To verify if the 'selftest' fields have 'passed', the /proc/crypto entries should be checked. An entry such as this:

```
name : md5
driver : md5-dpaa2-caam
module : kernel
priority : 3000
refcnt : 1
selftest : passed
internal : no
type : ahash
async : yes
blocksize : 64
digestsize : 16
```

means the driver has successfully registered support for the algorithm with the kernel crypto API. Note that although a test vector may not exist for a particular algorithm supported by the driver, the kernel will emit messages saying which algorithms weren't tested, and mark them as 'passed' anyway. The drivers capablilits can also be tested with tcrypt testing framework available in linux kernel by selecting "`Cryptographic API --> Testing module`" (also `Disable run-time self tests` should be unchecked). A kernel module will be generated: crypto/tcrypt.ko This has to be copied on the target. Then on target, after a dpseci object is registered:

```
insmod tcrypt.ko mode=10
```

Other values for tcrypt mode parameter:

functional:
mode=155-157, 181-190;
speed ("sec" - seconds - param. is optional):
mode=400 [sec=1] - xxx(md5,sha*) ahash_speed
mode=500 [sec=1] - xxx(aes) acipher_speed
mode=501 [sec=1] - xxx(3des) acipher_speed
mode=502 [sec=1] - xxx(des) acipher_speedetc.

`unavailable`

There is no need to rmmod, tcrypt does not stay "resident", it exits after running the tests. That's why you'll see:

```
#insmod: ERROR: could not insert module tcrypt.ko: Resource temporarily
```

For algorithms not supported, you'll see errors as follows:

```
[ 2650.067737] failed to load transform for rmd128: -2
[ 2650.076480] failed to load transform for rmd160: -2
[ 2650.085099] failed to load transform for rmd256: -2
[ 2650.093739] failed to load transform for rmd320: -2
```

These are expected. Algorithm names registered by DPSECI driver (DPAA2-CAAM) are ending in `"-dpaa2-caam"`.

To verify the operation and correctness of the driver, other than noting the performance advantages due to the crypto offload, one can also ensure the h/w is doing the crypto by looking for driver messages in `dmesg`. The driver emits console messages at initialization time:

```
# dmesg | grep fsl_dpaa2_caam
[    4.988889] MC object device driver fsl_dpaa2_caam registered
[ 1172.598591] fsl_dpaa2_caam dpseci.0: Opened dpseci object successfully
[ 1172.619979] fsl_dpaa2_caam dpseci.0: prio 0: rx queue 135, tx queue 119
[ 1172.626633] fsl_dpaa2_caam dpseci.0: prio 1: rx queue 136, tx queue 128
[ 1172.633278] fsl_dpaa2_caam dpseci.0: prio 2: rx queue 137, tx queue 129
[ 1172.639915] fsl_dpaa2_caam dpseci.0: prio 3: rx queue 138, tx queue 130
[ 1172.646555] fsl_dpaa2_caam dpseci.0: prio 4: rx queue 139, tx queue 131
[ 1172.653195] fsl_dpaa2_caam dpseci.0: prio 5: rx queue 140, tx queue 132
[ 1172.659831] fsl_dpaa2_caam dpseci.0: prio 6: rx queue 141, tx queue 133
[ 1172.666470] fsl_dpaa2_caam dpseci.0: prio 7: rx queue 142, tx queue 134
[ 1172.694319] fsl_dpaa2_caam dpseci.0: DPSECI version 3.0
[ 1172.700617] fsl_dpaa2_caam dpseci.0: algorithms registered in /proc/crypto
[ 1172.707971] fsl_dpaa2_caam dpseci.0: hash algorithms registered in /proc/crypto
```

Given a time period when crypto requests are being made, the SEC h/w will fire completion notification interrupts:

```
# cat /proc/interrupts | grep DPIO
```

If the number of interrupts fired increment, then the h/w is being used to do the crypto. If the numbers do not increment, then check if the algorithm being exercised is supported by the driver.

6.9.2 DPAA2 security engine use cases

Running OpenSSL

OpenSSL 1.0.2h and cryptodev module version 1.8 are available in SDK.

Testing OpenSSL with offload on SEC engine is done through cryptodev API. Cryptodev uses ioctl's to map user-space crypto requests to Linux kernel which executes them internally and then returns the results back to user-space. OpenSSL recognizes the cryptodev module automatically if it is loaded in the kernel so no special configuration is required for OpenSSL.

First, register dpseci driver:

```
# restool dpseci create --num-queues=8
      --priorities=1,2,3,4,5,6,7,8 && restool dprc assign dprc.1 --object=dpseci.0
      --plugged=1
```

Check that algorithms have been registered:

```
# cat /proc/crypto | grep dpaa2-caam
```

Load cryptodev module:

```
# modprobe cryptodev
```

Check that openssl has support for cryptodev engine:

```
# openssl engine
```

Check interrupts before running openssl tests:

```
# cat /proc/interrupts | grep DPIO
```

Run openssl tests:

```
# openssl speed -evp sha1 -engine cryptodev –elapsed
# openssl speed -evp md5 -engine cryptodev -elapsed
# openssl speed -evp aes-128-cbc -engine cryptodev -elapsed
# openssl speed -evp aes-256-cbc -engine cryptodev -elapsed
# openssl speed -evp aes-192-cbc -engine cryptodev –elapsed
# openssl speed -evp des-cbc -engine cryptodev –elapsed
```

Check interrupts after running openssl tests. Increased interrupts numbers confirm that execution was actually done with the SEC engine:

```
# cat /proc/interrupts
```

Running IPSec

DPNI configuration

Before running any configuration script on the boards you must ensure that there are 2 fully-feature configured DPNIs. The provided static DPL file configures only a single reduced-feature DPNI. To bypass this problem please remove any DPNIs from the DPL and configure the DPNIs dynamically. To do that write these commands after Linux boot at the command prompt:

```
$ ls-addni dpmac.2
$ ls-addni dpmac.1
```

Hardware setup

2 LS2080 boards;

Network simulator: Spirent TestCenter Performance Analysis System

Test Configuration

Linux Ipv4 SEC Forward Configuration

128 tunnels (bi-directional) configuration:

Spirent TestCenter configuration:

Port 1 to Port 2: Src: 192.85.1.2 – 192.85.1.9 Dst: 192.86.1.2 – 192.86.1.9

Port 2 to Port 1: Src: 192.86.1.2 – 192.86.1.9 - Src: 192.85.1.2 – 192.85.1.9

Linux configuration: The Linux configuration file is listed below.

On left gateway(Encrypt) board, run: bash iproute_128tunnels.sh left

On right gateway(Decrypt) board run: iproute_128tunnels.sh right

Linux Configuration File

```
#!/bin/bash
eth0=ni0
eth1=ni1

make_esp_tunnel() {

# ESP SAs do$dir2g encryption us$dir2g 192 bit long keys (168 + 24 parity)
# and hmac-sha1 authentication us$dir2g 160 bit long keys

echo "add $1 $2 esp 0x$3 -m tunnel
    -E $4  0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831
    -A hmac-sha1 0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;" | setkey -c

echo "add $2 $1 esp 0x`expr $3 + 100` -m tunnel
    -E $4  0xf6ddb555acfd9d77b03ea3843f2653255afe8eb5573965df
    -A hmac-sha1 0xea6856479330dc9c17b8f6c37e2a895363d83f21;" | setkey -c

}

make_esp_policy() {

# Security policies1
if [ $1 == left ]
then
    dir1=out
    dir2=in
    echo "spdadd $2 $3 any -P $dir1 ipsec
           esp/tunnel/$4-$5/require;" | setkey -c
    echo "spdadd $3 $2 any -P $dir2 ipsec
           esp/tunnel/$5-$4/require;" | setkey -c
else
    dir1=in
    dir2=out
    echo "spdadd $2 $3 any -P $dir1 ipsec
           esp/tunnel/$4-$5/require;" | setkey -c
    echo "spdadd $3 $2 any -P $dir2 ipsec
           esp/tunnel/$5-$4/require;" | setkey -c
fi

}

# Flush the SAD and SPD
setkey -F
setkey -FP

# set ip address
left_addr_ip=192.85.1.1
```

```
right_addr_ip=192.86.1.1
left_src_mac=00:10:94:00:00:01
right_src_mac=00:10:94:00:00:02
proto="aes-cbc"
base1=200
base2=200
echo 1 > /proc/sys/net/ipv4/ip_forward

case $1 in
        left)
                ifconfig $eth0 $left_addr_ip
                i=2
                for((j=2;j<10;j++))
                do
                    arp -s 192.85.1.$j $left_src_mac -i $eth0
                    for((k=2;k<10;k++))
                    do
                    if [ $base2 == 256 ]
                    then
                    base2=`expr $base2 - 256`
                        base1=`expr $base1 + 1`
                    fi
                    ip addr add 200.$base1.$base2.10/24 dev $eth1
                        make_esp_policy $1 192.85.1.$j 192.86.1.$k 200.$base1.$base2.10
200.$base1.$base2.20
                        make_esp_tunnel 200.$base1.$base2.10 200.$base1.$base2.20 `expr 200 + $i`
$proto
                        ((base2++))
                        ((i++))
                    done
                done
ifconfig $eth1 up
route add default dev $eth1

            ;;
        right)
                ifconfig $eth0 $right_addr_ip
                i=2
                for((j=2;j<10;j++))
                do
                    arp -s 192.86.1.$j $left_src_mac -i $eth0
                    for((k=2;k<10;k++))
                    do
                    if [ $base2 == 256 ]
                    then
                    base2=`expr $base2 - 256`
                        base1=`expr $base1 + 1`
                    fi
                    ip addr add 200.$base1.$base2.20/24 dev $eth1
                        make_esp_policy $1 192.85.1.$j 192.86.1.$k 200.$base1.$base2.10
200.$base1.$base2.20
                        make_esp_tunnel 200.$base1.$base2.10 200.$base1.$base2.20 `expr 200 + $i`
$proto
                        ((base2++))
                        ((i++))
                    done
                done
        ifconfig $eth1 up
        route add default dev $eth1
```

```
            ;;
esac
```

Running the Test

After Spirent Testcenter application is configured and the testing Ethernet interfaces are connected to the traffic generator, start to generate traffic to measure IPv4 SEC & Forward throughput.

6.10 Decompression Compression Engine (DCE)

Introduction

This section describes the software interface to DCE (Decompression Compression Engine) accelerator available on the LS2088A SoC.The interface is designed to simplify interaction with DCE as much as possible without loss of flexibility and acceleration offered by DCE hardware.

Hardware Overview

This section gives an overview of the operation of the DCE hardware to provide basic fundamentals for software developers using the driver API. More detailed information is available in the hardware reference manual.

The DCE is a hardware accelerator that is part of the Datapath Acceleration Architecture version 2 (DPAA2). The DCE is one of the DPAA2 accelerators that include others like security and pattern matching engines. These accelerators are connected using a queue manager (QMan) and buffer manager (BMan) that allows data to be exchanged between software and accelerators.

Software enqueues data to a TX frame queue leading to DCE. DCE receives the data and processes it. It then enqueues a response on an RX frame queue leading back to software. The software then provides a DCE object that abstracts the details of frame queue configuration and usage, as well as other hardware details. This object is called DPDCEI.

Software Elements

The DCE driver APIs are documented in the release API Reference Manual. Instructions to run example kernel module, as well as expected output, is documented in later sections of this document.

SDK 2.0 release supports only kernel-based DCE software. User space driver and example applications are planned for future releases.

6.10.1 The DCE application

The `fsl-dce-api-time-trial` application is included in the release as an example application of the DCE. It performs basic compression and decompression and prints performance metrics.

The application sends as many work units to DCE as possible in a given number of seconds. The work units are sent asynchronously to the DCE and are configured for DCE to perform compression. Processed work units from the DCE invoke an applicaiton call back routine. The applicaiton keeps a count of outstanding work units and throttles back if the count exceeds a pre-determined threshhold. Once all work units have been sent the test waits for all processed work units from DCE to be received. The test then calculates and prints compression related performance data and checks its integrity. Performance numbers are then printed. The test runs then takes the compressed data from DCE and sends it back through the DCE - the work units configured to be decompressed this time. Decompression performance results are calculated and printed once all work units have gone through the decompression step. The test also compares the decompressed data against the original data for validity.

To keep the test logic simple, the data sent to DCE for compression is the same for all work units. This is important to know as the data used for compression or decompression is one of the factors that impacts DCE performance. Other factors like the work unit size and compression level also affect DCE performance. Users can specifiy the latter via paramters to the DCE applicaiton.

The test can be packaged as a loadable module and it takes optional parameters that specify time length for test in seconds, the size of each work unit, whether the test should run in verbose mode, and the compression level from 0 to 3. 3 being the best possible compression and 0 being no compression.

6.10.2 Running the DCE test application

Create DCE compression and decompression objects. Assign to appropriate resource container (dprc)

```
$ restool dpdcei create --engine=DPDCEI_ENGINE_COMPRESSION --priority=1
/* dpdcei.0 is created under dprc.1 */

$ restool dpdcei create --engine=DPDCEI_ENGINE_DECOMPRESSION --priority=1
/* dpdcei.1 is created under dprc.1 */

$ restool dprc assign dprc.1 --object=dpdcei.0 --plugged=1
$ restool dprc assign dprc.1 --object=dpdcei.1 --plugged=1
```

In this example the DCE API was built as a module so it must be loaded before we can load the test which depends on it

```
$ insmod fsl-dce-api.ko
fsl_dce_api: module is from the staging directory, the quality is unknown, you have
been warned.
fsl_dce_api dpdcei.0: dpdcei probe
fsl_dce_api dpdcei.0: DPDCEI: id=0, engine=COMPRESSION
fsl_dce_api dpdcei.0: dpdcei: probed object 0
fsl_dce_api dpdcei.1: dpdcei probe
fsl_dce_api dpdcei.1: DPDCEI: id=1, engine=DECOMPRESSION
fsl_dce_api dpdcei.1: dpdcei: probed object 1 MC object device driver
fsl_dce_api registered
```

Insert the test module to run the test

```
$ insmod fsl-dce-api-time-trial.ko
fsl_dce_api_time_trial: module is from the staging directory, the quality is unknown,
you have been warned.
```

Compression results are output by the test application upon completion

```
Running compression test for 30 seconds ...
Number of work units 2320000, work unit size 8192, compression ratio (out/in) 31%
DCE Driver API & DCE performance = 5039559827 bit/s and time is 30170 ms
```

Decompression results from the test are shown below

```
Running decompression test for 30 seconds ...
Number of work units 2720000, work unit size 2612, decompression ratio (out/in) 313%
DCE Driver API & DCE performance = 5908449453 bit/s and time is 30170 ms
```

Remove the test application and clean-up resources using the following commands

```
$ rmmod fsl-dce-api-time-trial.ko
dce_api_time_trial_exit
$ echo dpdcei.0 > /sys/bus/fsl-mc/drivers/fsl_dce_api/unbind
$ echo dpdcei.1 > /sys/bus/fsl-mc/drivers/fsl_dce_api/unbind
$ restool dprc assign dprc.1 --object=dpdcei.1 --plugged=0
$ restool dprc assign dprc.1 --object=dpdcei.0 --plugged=0
$ rmmod fsl-dce-api
$ restool dpdcei destroy dpdcei.0
dpdcei.0 is destroyed
$ restool dpdcei destroy dpdcei.1
dpdcei.1 is destroyed
$
```

6.11 DPAA2 Standard Linux Documentation

Following is a summary of relevant documentation from standard Linux sources and formats. It provides links to these documents, provides a snapshot of the document, or both.

6.11.1 Kernel Documentation Directory

The Linux kernel source code contains a documentation directory, and there is some information there that is relevant to DPAA2. It is possible to see the upstream versions of these documents by going to http://lkernel.org and browsing the Linux source code trees.

- Kernel Management Complex (MC) bus driver: This document is in-flight to kernel.org so a copy is provided below rather than a link to kernel.org.

```
Copyright (C) 2016 Freescale Semiconductor Inc.

DPAA2 (Data Path Acceleration Architecture Gen2)
-----------------------------------------------

This document provides an overview of the Freescale DPAA2 architecture
and how it is integrated into the Linux kernel.

Contents summary
   -DPAA2 overview
   -Overview of DPAA2 objects
   -DPAA2 Linux driver architecture overview
        -bus driver
        -dprc driver
        -allocator
        -dpio driver
        -Ethernet
        -mac

DPAA2 Overview
--------------

DPAA2 is a hardware architecture designed for high-speeed network
packet processing.  DPAA2 consists of sophisticated mechanisms for
```

```
processing Ethernet packets, queue management, buffer management,
autonomous L2 switching, virtual Ethernet bridging, and accelerator
(e.g. crypto) sharing.


A DPAA2 hardware component called the Management Complex (or MC) manages the
DPAA2 hardware resources.  The MC provides an object-based abstraction for
software drivers to use the DPAA2 hardware.

The MC uses DPAA2 hardware resources such as queues, buffer pools, and
network ports to create functional objects/devices such as network
interfaces, an L2 switch, or accelerator instances.

The MC provides memory-mapped I/O command interfaces (MC portals)
which DPAA2 software drivers use to operate on DPAA2 objects:

         +--------------------------------------+
         |                  OS                  |
         |                        DPAA2 drivers |
         |                             |        |
         +-----------------------------|--------+
                                       |
                                       | (create,discover,connect
                                       |  config,use,destroy)
                                       |
                        DPAA2          |
         +-------------------------| mc portal |-+
         |                         |        |
         |   +- - - - - - - - - - -V- - -+  |
         |   |                           |  |
         |   |   Management Complex (MC) |  |
         |   |                           |  |
         |   +- - - - - - - - - - - - - -+  |
         |                                  |
         | Hardware                Hardware |
         | Resources               Objects  |
         | ---------               -------  |
         | -queues                 -DPRC    |
         | -buffer pools           -DPMCP   |
         | -Eth MACs/ports         -DPIO    |
         | -network interface      -DPNI    |
         |  profiles               -DPMAC   |
         | -queue portals          -DPBP    |
         | -MC portals              ...     |
         |  ...                             |
         |                                  |
         +----------------------------------+

The MC mediates operations such as create, discover,
connect, configuration, and destroy.  Fast-path operations
on data, such as packet transmit/receive, are not mediated by
the MC and are done directly using memory mapped regions in
DPIO objects.

Overview of DPAA2 Objects
-------------------------
The section provides a brief overview of some key objects
in the DPAA2 hardware.  A simple scenario is described illustrating
the objects involved in creating a network interfaces.
```

```
-DPRC (Datapath Resource Container)

    A DPRC is an container object that holds all the other
    types of DPAA2 objects.  In the example diagram below there
    are 8 objects of 5 types (DPMCP, DPIO, DPBP, DPNI, and DPMAC)
    in the container.

    +---------------------------------------------------------+
    | DPRC                                                    |
    |                                                         |
    |  +-------+  +-------+  +-------+  +-------+  +-------+  |
    |  | DPMCP |  | DPIO  |  | DPBP  |  | DPNI  |  | DPMAC |  |
    |  +-------+  +-------+  +-------+  +---+---+  +---+---+  |
    |  | DPMCP |  | DPIO  |                                   |
    |  +-------+  +-------+                                   |
    |  | DPMCP |                                             |
    |  +-------+                                             |
    |                                                         |
    +---------------------------------------------------------+

    From the point of view of an OS, a DPRC is bus-like.  Like
    a plug-and-play bus, such as PCI, DPRC commands can be used to
    enumerate the contents of the DPRC, discover the hardware
    objects present (including mappable regions and interrupts).

     dprc.1 (bus)
       |
       +--+--------+-------+-------+-------+
          |        |       |       |       |
        dpmcp.1  dpio.1  dpbp.1  dpni.1  dpmac.1
        dpmcp.2  dpio.2
        dpmcp.3

    Hardware objects can be created and destroyed dynamically, providing
    the ability to hot plug/unplug objects in and out of the DPRC.

    A DPRC has a mappable mmio region (an MC portal) that can be used
    to send MC commands.  It has an interrupt for status events (like
    hotplug).

    All objects in a container share the same hardware "isolation context".
    This means that with respect to an IOMMU the isolation granularity
    is at the DPRC (container) level, not at the individual object
    level.

    DPRCs can be defined statically and populated with objects
    via a config file passed to the MC when firmware starts
    it.  There is also a Linux user space tool called "restool"
    that can be used to create/destroy containers and objects
    dynamically.

-DPAA2 Objects for an Ethernet Network Interface

    A typical Ethernet NIC is monolithic-- the NIC device contains TX/RX
    queuing mechanisms, configuration mechanisms, buffer management,
    physical ports, and interrupts.  DPAA2 uses a more granular approach
    utilizing multiple hardware objects.  Each object has specialized
    functions, and are used together by software to provide Ethernet network
    interface functionality.  This approach provides efficient use of finite
    hardware resources, flexibility, and performance advantages.
```

```
The diagram below shows the objects needed for a simple
network interface configuration on a system with 2 CPUs.



          +---+---+ +---+---+
             CPU0     CPU1
          +---+---+ +---+---+
              |         |
          +---+---+ +---+---+
             DPIO     DPIO
          +---+---+ +---+---+
                \     /
                 \   /
                  \ /
               +---+---+
                  DPNI  --- DPBP,DPMCP
               +---+---+
                   |
                   |
               +---+---+
                 DPMAC
               +---+---+
                   |
                port/PHY

Below the objects are described.  For each object a brief description
is provided along with a summary of the kinds of operations the object
supports and a summary of key resources of the object (mmio regions
and irqs).

   -DPMAC (Datapath Ethernet MAC): represents an Ethernet MAC, a
    hardware device that connects to an Ethernet PHY and allows
    physical transmission and reception of Ethernet frames.
       -mmio regions: none
       -irqs: dpni link change
       -commands: set link up/down, link config, get stats,
         irq config, enable, reset

   -DPNI (Datapath Network Interface): contains TX/RX queues,
    network interface configuration, and rx buffer pool configuration
    mechanisms.
       -mmio regions: none
       -irqs: link state
       -commands: port config, offload config, queue config,
        parse/classify config, irq config, enable, reset

   -DPIO (Datapath I/O): provides interfaces to enqueue and dequeue
    packets and do hardware buffer pool management operations.  For
    optimum performance there is typically one DPIO per CPU.  This allows
    each CPU to perform simultaneous enqueue/dequeue operations.
       -mmio regions: queue operations, buffer mgmt
       -irqs: data availability, congestion notification, buffer
              pool depletion
       -commands: irq config, enable, reset

   -DPBP (Datapath Buffer Pool): represents a hardware buffer
    pool.
       -mmio regions: none
```

```
            -irqs: none
            -commands: enable, reset

        -DPMCP (Datapath MC Portal): provides an MC command portal.
         Used by drivers to send commands to the MC to manage
         objects.
            -mmio regions: MC command portal
            -irqs: command completion
            -commands: irq config, enable, reset

    Object Connections
    ------------------
    Some objects have explicit relationships that must
    be configured:

       -DPNI <--> DPMAC
       -DPNI <--> DPNI
       -DPNI <--> L2-switch-port
          A DPNI must be connected to something such as a DPMAC,
          another DPNI, or L2 switch port.  The DPNI connection
          is made via a DPRC command.

              +-------+  +-------+
              | DPNI  |  | DPMAC |
              +---+---+  +---+---+
                  |          |
                  +==========+

       -DPNI <--> DPBP
          A network interface requires a 'buffer pool' (DPBP
          object) which provides a list of pointers to memory
          where received Ethernet data is to be copied.  The
          Ethernet driver configures the DPBPs associated with
          the network interface.

    Interrupts
    ----------
    All interrupts generated by DPAA2 objects are message
    interrupts.  At the hardware level message interrupts
    generated by devices will normally have 3 components--
    1) a non-spoofable 'device-id' expressed on the hardware
    bus, 2) an address, 3) a data value.

    In the case of DPAA2 devices/objects, all objects in the
    same container/DPRC share the same 'device-id'.
    For ARM-based SoC this is the same as the stream ID.


DPAA2 Linux Driver Overview
---------------------------

This section provides an overview of the Linux kernel drivers for
DPAA2-- 1) the bus driver and associated "DPAA2 infrastructure"
drivers and 2) functional object drivers (such as Ethernet).

As described previously, a DPRC is a container that holds the other
types of DPAA2 objects.  It is functionally similar to a plug-and-play
bus controller.

Each object in the DPRC is a Linux "device" and is bound to a driver.
```

```
The diagram below shows the Linux drivers involved in a networking
scenario and the objects bound to each driver.  A brief description
of each driver follows.



                                          +------------+
                                          | OS Network |
                                          |   Stack    |
                +------------+            +------------+
                | Allocator  |. . . . . . |  Ethernet  |
                |(dpmcp,dpbp)|            |   (dpni)   |
                +-.----------+            +---+---+----+
                  .          .                ^   |
                  .           .     <data avail, |   |<enqueue,
                 .             .     tx confirm> |   | dequeue>
    +-------------+             .               |   |
    | DPRC driver |              .          +---+---V----+     +----------+
    |   (dprc)    |               . . . . . .| DPIO driver|     |   MAC    |
    +----------+--+                          |  (dpio)    |     | (dpmac)  |
               |                             +------+-----+     +-----+---+
               |<dev add/remove>                    |                 |
               |                                    |                 |
          +----+--------------+                     |              +--+---+
          |   mc-bus driver   |                     |              | PHY  |
          |                   |                     |              |driver|
          | /fsl-mc@80c000000 |                     |              +--+---+
          +-------------------+                     |                 |
                                                    |                 |
 ================================ HARDWARE =========|=================|======
                                                   DPIO               |
                                                     |                |
                                                   DPNI---DPBP        |
                                                     |                |
                                                   DPMAC              |
                                                     |                |
                                                    PHY ---------------+
 ===================================================|=======================

A brief description of each driver is provided below.

    mc-bus driver
    -------------
    The mc-bus driver is a platform driver and is probed from an
    "/fsl-mc@xxxx" node in the device tree passed in by boot firmware.
    It is responsible for bootstrapping the DPAA2 kernel infrastructure.
    Key functions include:
       -registering a new bus type named "fsl-mc" with the kernel,
        and implementing bus call-backs (e.g. match/uevent/dev_groups)
       -implemeting APIs for DPAA2 driver registration and for device
        add/remove
       -creates an MSI irq domain
       -do a device add of the 'root' DPRC device, which is needed
        to bootstrap things

    DPRC driver
    -----------
    The dprc-driver is bound to DPRC objects and does runtime management
    of a bus instance.  It performs the initial bus scan of the DPRC
    and handles interrupts for container events such as hot plug.
```

```
Allocator
----------
Certain objects such as DPMCP and DPBP are generic and fungible,
and are intended to be used by other drivers.  For example,
the DPAA2 Ethernet driver needs:
   -DPMCPs to send MC commands, to configure network interfaces
   -DPBPs for network buffer pools

The allocator driver registers for these allocatable object types
and those objects are bound to the allocator when the bus is probed.
The allocator maintains a pool of objects that are available for
allocation by other DPAA2 drivers.

DPIO driver
-----------
The DPIO driver is bound to DPIO objects and provides services that allow
other drivers such as the Ethernet driver to receive and transmit data.
Key services include:
    -data availability notifications
    -hardware queuing operations (enqueue and dequeue of data)
    -hardware buffer pool management

There is typically one DPIO object per physical CPU for optimum
performance, allowing each CPU to simultaneously enqueue
and dequeue data.

The DPIO driver operates on behalf of all DPAA2 drivers
active in the kernel--  Ethernet, crypto, compression,
etc.

Ethernet
--------
The Ethernet driver is bound to a DPNI and implements the kernel
interfaces needed to connect the DPAA2 network interface to
the network stack.

Each DPNI corresponds to a Linux network interface.

MAC driver
----------
An Ethernet PHY is an off-chip, board specific component and is managed
by the appropriate PHY driver via an mdio bus.  The MAC driver
plays a role of being a proxy between the PHY driver and the
MC.  It does this proxy via the MC commands to a DPMAC object.
```

6.11.2 DPAA2 Resource Management Tool (restool) User Manual

Restool is a Linux user space program that allows DPAA2 objects to be created, destroyed, and manipulated. Its primary documentation is in the style of a Linux man page.

The Management Complex architecture uses a hardware object called a “container” (or DPRC) to hold I/O resources and hardware objects for use by GPP software contexts.

DPRCs can be created and populated in two different ways:

- at MC initialization during system boot in a configuration file called a “DPL file”
- dynamically at runtime

This document describes how restool can be used to do dynamic management of MC resources in the context of Linux. Key resource management operations include:

- listing containers and their contents
- creating/destroying containers
- creating/destroying new MC objects
- move object between parent container and child container
- establishing connections between MC objects

6.11.2.1 DPRC commands

6.11.2.1.1 list command

The **list** command lists all containers in the system.

SYNTAX:

restool dprc list

ARGUMENTS:

none

EXAMPLE:

List all the containers in the system

```
$ restool dprc list
  dprc.1
      dprc.2
      dprc.3
```

The container hierarchy (parent-child relationships) is shown by indentation.

6.11.2.1.2 show command

The **show** command displays the contents (objects and resources) of a DPRC/container.

SYNTAX:

restool dprc show <*container*>

restool dprc show <*container*> **--resources**

restool dprc show <*container*> **--resource-type**=<resource-type>

ARGUMENTS:

<*container*>

A string specifying the target dprc—e.g. “dprc.2”.

The container argument value “mc.global” is special and refers to the global

container of resource pools inside the Management Complex.

--resources

Display a container’s resource count for each resource (instead of displaying

objects/resources)

--resource-type <resource-type>

Specifies the type of resource to list. The resource-type argument is a string specifying

the resource name—e.g. “mcp”.

EXAMPLE :

Show all objects in dprc 2:

```
$ restool dprc show dprc.2
dprc.2 contains 6 objects:
object label plugged state
dpni.7 xyz  plugged
dpni.8 abc  plugged
dpio.2 plugged
dpio.3 unplugged
dpcon.9 plugged
dpbp.1 plugged
```

Show all resources in dprc 2:

```
$ restool dprc show dprc.2 --resources
bpid: 16
fqid: 100
channel: 4
qpr: 2
cgid: 2
```

Show dprc with no objects in it:

```
$ restool dprc show dprc.4
(empty)
```

Show all buffer pool IDs in dprc 2:

```
$ restool dprc show dprc.2 --resource-type=bp
bp.35 - bp.36
bp.50
bp.52 - bp.63
```

Show all MC portal IDs in the global MC container:

```
$ restool dprc show mc.global --resource-type=mcp
mcp.30 - mcp.250
```

6.11.2.1.3 info command

The **info** command displays detailed information about a specific container.

SYNTAX:

restool dprc info <*dprc-object*> [**--verbose**]

ARGUMENTS :

<*dprc-object*>

Specifies which container to show detailed info for. The object argument is a string

specifying the container name—e.g. “dprc.2”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dprc info dprc.2
container id: 2
icid: 2
portal id: 5
version: 0.0
dprc options: 0x3
    DPRC_CFG_OPT_SPAWN_ALLOWED
    DPRC_CFG_OPT_ALLOC_ALLOWED
object label: nadk’s dprc

$ restool dprc info dprc.2 --verbose
container id: 2
icid: 2
portal id: 5
version: 0.0
dprc options: 0x3
    DPRC_CFG_OPT_SPAWN_ALLOWED
    DPRC_CFG_OPT_ALLOC_ALLOWED
object label: nadk-usage-dprc
number of mappable regions: 1
number of interrupts: 1
interrupt 0's mask: 0
interrupt 0's status: 0x1
```

6.11.2.1.4 create command

The **create** command creates a new child DPRC under the specified parent. The name/id of the

object created is displayed to stdout.

SYNTAX:

restool dprc create <parent-container> [**--options**=<*options-mask*>] [**--label**=<*object’s-label*>]

OPTIONS :

<parent-container>

--options=<*options-mask*>

Where <options-mask> is a comma separated list of DPRC options:

```
            DPRC_CFG_OPT_SPAWN_ALLOWED
            DPRC_CFG_OPT_ALLOC_ALLOWED
            DPRC_CFG_OPT_OBJ_CREATE_ALLOWED
            DPRC_CFG_OPT_TOPOLOGY_CHANGES_ALLOWED
            DPRC_CFG_OPT_IOMMU_BYPASS
            DPRC_CFG_OPT_AIOP
            DPRC_CFG_OPT_IRQ_CFG_ALLOWED
```

--label=<*object’s-label*>

Specify a label for the newly created object. It is kind of an alias for that object.

Length of the string is 15 characters maximum.

Say --label=”nadk’s dprc”

EXAMPLE:

Create a child DPRC under parent dprc.1 with default options:

```
$ restool dprc create dprc.1
dprc.9 is created under dprc.1
```

Create a child DPRC under parent dprc.1 with default options, with label “nadk’s dprc”:

```
$ restool dprc create dprc.1 --label=”nadk’s dprc”
dprc.11 is created under dprc.1
```

6.11.2.1.5 destroy command

The **destroy** command destroys the specified DPRC.

SYNTAX:

restool dprc destroy <container> **--help**

OPTIONS:

<container>

--help

Displays help for the command.

EXAMPLE:

Destroy a specified DPRC, say dprc.2:

```
$ restool dprc destroy dprc.2
dprc.2 is destroyed
```

6.11.2.1.6 assign command

The **assign** command moves an object or resource from a parent container to a child container. Object (dpni, dpbp, etc) assignment is always explicit and the exact object id to be assigned must be specified. Resources (e.g. mcp, bp, fq, etc) are assigned by type and count.

SYNTAX:

restool dprc assign <parent-*container*> [**--child**=<child-*container*>] **--object**=<*object*> **--plugged**=<*state*>

This syntax changes the plugged state. The child-container must be the same as parent-container, or omit --target option. It is not possible to change the plugged state of a dprc.

restool dprc assign <parent-*container*> [**--child**=<child-*container*>] **--object**=<*object*>

This syntax moves one object from parent-container to another, so the target-container must be different from the parent-container. Limitation: cannot move dprc from one container to another.

restool dprc assign <parent-*container*> [**--child**=<child-*container*>] --resource-type=<*type*> **-- count**=<*number*>

This syntax moves a resource from parent-container to a child-container. If the childcontainer is the same as the parent-container, the resource will be taken from the parent of parent-container and will be assigned to the parent-container.

ARGUMENTS :

<*container*>

Specifies the parent container from which the object will be moved.

--object=<*object*>

Specifies the object to assign— value is a string specifying object name and ID (e.g. dpni.5)

--child=<*child-container*>

Specifies the destination container for the operation. Valid values are any child container. (The target container may be the same as the parent container, allowing “assign to self”)

--plugged=<*state*>

Specifies the plugged state of the object (valid values are 0 or 1)

--resource-type=<*type*>

String specifying the resource type to assign (e.g “mcp”, “fq”, “cg”, etc). To see valid resources that may be assigned use the “dprc show <container> --resources” command.

--count=<*number*>

Number of resources to assign.

EXAMPLE:

Set the plugged state of dpni.5. Note source and destination containers are the same.

```
$ restool dprc assign dprc.1 --object=dpni.5 --child=dprc.1
--plugged=1
$ restool dprc assign dprc.1 --object=dpni.5 --plugged=1
```

Unset the plugged state of dpni.5. Note source and destination containers are the same.

```
$ restool dprc assign dprc.1 --object=dpni.5 --child=dprc.1
--plugged=0
$ restool dprc assign dprc.1 --object=dpni.5 --plugged=0
```

Move dpni.5 from dprc.1 (parent) to dprc.3 (child):

```
$ restool dprc assign dprc.1 --object=dpni.5 --child=dprc.3
```

Move 3 mcp resources from dprc.1 (parent) to dprc.2 (child):

```
$ restool dprc assign dprc.1 --resource-type=mcp --count=3
--child=dprc.2
```

6.11.2.1.7 unassign command

The **unassign** command moves an object or resource from a child container to a parent container

SYNTAX:

restool dprc unassign <*container*> **--object**=<*object*> [**--child**=<child-*container*>]

restool dprc unassign <*container*> **--resource-type**=<*type*> **--count** <*number*> [**--child**=<child-*container*>]

ARGUMENTS :

<container>

Specifies the container to which the object will be moved.

--object=*<object>*

Specifies the object to unassign— value is a string specifying object name and ID (e.g.

dpni.5)

--child=<child-*container*>

Specifies the container from which the object/resource will be moved from.

--plugged=*<plugged-state>*

Specifies the plugged state of the object (valid values are 0 or 1)

--resource-type=*<type>*

String specifying the resource type to assign (e.g “mcp”, “fq”, “cg”, etc)

--count=*<number>*

Number of resources to unassign.

EXAMPLE:

Unassign 3 mcp resources from dprc.2 (child) to dprc.1 (parent):

```
$ restool dprc unassign dprc.1 --resource-type=mcp --count=3
--child=dprc.2
```

Unassign dpni.5 from dprc.3 (child) to dprc.1 (parent):

```
$ restool dprc unassign dprc.1 --object=dpni.5 --child=dprc.3
```

6.11.2.1.8 set-quota command

The **set-quota** command sets quota policies for a child container, specifying the number of resources a child may take from its parent container. But remember a parent can assign any number of resource to its child if it wants to, and if it has enough resources to assign. So the quota is effective only when the child dprc does have enough resource and it wants to borrow resource from its parent. It could only “borrow” the quota number of resources from its parent.

SYNTAX:

restool dprc set-quota *<parent-container>* **--resource-type**=*<type>* **--count**=*<number>*

--child-container=*<container>*

ARGUMENTS :

<parent-container>

Specifies the parent container.

--resource-type=*<type>*

String specifying the resource type to set the quota for (e.g “mcp”, “fq”, “cg”, etc)

--count=*<number>*

Max number of resources the child is able to allocate.

--child-container=<*container*>

EXAMPLE:

Set a quota of 10 mcp resource that child container dprc.5 may take from parent dprc.1:

```
$ restool dprc set-quota dprc.1 --resource-type=mcp --count=10
   --child-container=dprc.5
```

6.11.2.1.9 set-label command

The **set-label** command sets label for any objects excluding dprc.1

SYNTAX:

restool dprc set-label <*object*> **--label**=<*label*>

ARGUMENTS :

<*object*>

Specifies the object to be set.

--label=<*label*>

String specifying the label, maximum length is 15 characters.

EXAMPLE:

Set label of dprc.4 to “mountain view”:

```
$ restool dprc set-label dprc.4 --label=”mountain view”
```

6.11.2.1.10 connect command

The **connect** command connects 2 objects, creating a link between them.

SYNTAX:

restool dprc connect <*container*> **--endpoint1**=<*object*> **--endpoint2**=<*object*>

ARGUMENTS :

<*container*>

A string specifying the target dprc—e.g. “dprc.2”.

--endpoint1=<*object*>

Specifies the first endpoint object.

--endpoint2=<*object*>

Specifies the second endpoint object.

EXAMPLE:

The connect command connects a network object such as a DPNI to a peer object such as a DPMAC or DPSW port.

Connect dpni.2 to dpmac.5:

```
$ restool dprc connect dprc.2 --endpoint1=dpni.2 --endpoint2=dpmac.5
```

Connect dpni.2 to dpsw.1 interface 7:

```
$ restool dprc connect dprc.2 --endpoint1=dpni.2 --endpoint2=dpsw.1.7
```

6.11.2.1.11 disconnect command

The **disconnect** command removes the link between two objects. Either endpoint can be specified as the target of the operation.

SYNTAX:

restool dprc disconnect <*container*> **--endpoint**=<*object*>

ARGUMENTS:

<*container*>

A string specifying the target dprc—e.g. “dprc.2”

--endpoint=<*object*>

Specifies the first endpoint object.

EXAMPLE:

Remove the link between dpni.2 and dpmac.5

```
$ restool dprc disconnect dprc.2 –endpoint=dpni.2
```

6.11.2.1.12 generate-dpl command

The **generate-dpl** command prints to the standard output a DPL syntax file describing the specified container

SYNTAX:

restool dprc generate-dpl <*container*>

ARGUMENTS:

<*container*>

A string specifying the target dprc—e.g. “dprc.2”

EXAMPLE:

```
Generate a DPL for dprc.1
```

```
$ restool dprc generate-dpl dprc.1
```

6.11.2.2 DPNI Commands

6.11.2.2.1 help command

The **help** command displays usage information for the DPNI object

SYNTAX:

restool dpni help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpni help
usage: restool dpni <command> [--help][ARGS…]
Where <command> can be:
    info - displays detailed information about a DPNI object.
    create - creates a child DPNI under the root DPRC
    destroy - destroys a child DPNI under the root DPRC

For command-specific help, use the --help option available for each command.
```

6.11.2.2.2 info command

The **info** command displays detailed information about a specific dpni object.

SYNTAX:

restool dpni info <*dpni-object*> [**--verbose**]

ARGUMENTS :

<*dpni-object*>

Specifies which dpni object to show detailed info for. The dpni-object argument is a

string specifying the object name—e.g. “dpni.7”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpni info dpni.7
dpni version: 5.0
dpni id: 7
plugged state: plugged
endpoint: dpmac.2, link is down
link status: 0 - down
mac address: 00:00:00:00:00:07
dpni_attr.options value is: 0x190
    DPNI_OPT_DIST_HASH
    DPNI_OPT_UNICAST_FILTER
    DPNI_OPT_MULTICAST_FILTER
max senders: 8
max traffic classes: 1
max distribution's size per RX traffic class:
    class 0's size: 15
max unicast filters: 16
max multicast filters: 64
max vlan filters: 0
max QoS entries: 0
max QoS key size: 0
max distribution key size: 4

$ restool dpni info dpni.7 --verbose
dpni version: 5.0
dpni id: 7
plugged state: plugged
endpoint: dpmac.2, link is down
link status: 0 - down
```

```
    mac address: 00:00:00:00:00:07
    dpni_attr.options value is: 0x190
        DPNI_OPT_DIST_HASH
        DPNI_OPT_UNICAST_FILTER
        DPNI_OPT_MULTICAST_FILTER
    max senders: 8
    max traffic classes: 1
    max distribution's size per RX traffic class:
        class 0's size: 15
    max unicast filters: 16
    max multicast filters: 64
    max vlan filters: 0
    max QoS entries: 0
    max QoS key size: 0
    max distribution key size: 4
    number of mappable regions: 0
    number of interrupts: 1
    interrupt 0's mask: 0
    interrupt 0's status: 0
```

6.11.2.2.3 create command

The **create** command creates a new DPNI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpni create --mac-addr=<*addr*> [OPTIONS]

ARGUMENTS :

--mac-addr=<*addr*>

String specifying primary MAC address (e.g., 00:00:05:00:00:05)

OPTIONS :

--max-senders=<*number*>

maximum number of different senders; will be used as the number of dedicated TX flows;

In case it isn't power-of-2 it will be ceiling to the next power-of-2 as HW demand it; 0 will

be treated as 1

--options=<*options-mask*>

Where <*options-mask*> is a comma separated list of DPNI options:

```
            DPNI_OPT_ALLOW_DIST_KEY_PER_TC
            DPNI_OPT_TX_CONF_DISABLED
            DPNI_OPT_PRIVATE_TX_CONF_ERR_DISABLED
            DPNI_OPT_DIST_HASH
            DPNI_OPT_DIST_FS
            DPNI_OPT_UNICAST_FILTER
            DPNI_OPT_MULTICAST_FILTER
            DPNI_OPT_VLAN_FILTER
            DPNI_OPT_IPR
            DPNI_OPT_IPF
            DPNI_OPT_VLAN_MANIPULATION
            DPNI_OPT_QOS_MASK_SUPPORT
            DPNI_OPT_FS_MASK_SUPPORT
```

--max-tcs=<*number*>

Specifies the maximum number of traffic-classes

--max-dist-per-tc=<*dist-size*>,<*dist-size*>,…

Comma separated list of counts specifying the maximum distribution's size per RX traffic-

class

--max-unicast-filters=<*number*>

maximum number of unicast filters; 0 will be treated as 16

--max-multicast-filters=<*number*>

maximum number of multicast filters; 0 will be treated as 64

--max-vlan-filters=<*number*>

maximum number of vlan filters; '0' will be treated as '16'

--max-qos-entries=<*number*>

if max_tcs > 1, declares the maximum entries for the QoS table; '0' will be treated as '64'

--max-qos-key-size=<*number*>

maximum key size for the QoS look-up; '0' will be treated as '24' which enough for IPv4 5-tuple

--max-dist-key-size=<*number*>

maximum key size for the distribution; '0' will be treated as '24' which enough for IPv4 5-tuple

EXAMPLE:

Create a DPNI, specifying MAC address, with all default options:

```
$ restool dpni create --mac-addr=00:00:05:00:00:05
dpni.9 is crated under dprc.1
```

Create a DPNI, specifying MAC address, and some options:

```
$ restool dpni create --mac-addr=00:00:05:00:00:05
--options=DPNI_OPT_MULTICAST_FILTER,DPNI_OPT_UNICAST_FILTER
dpni.11 is created under dprc.1
```

6.11.2.2.4 destroy command

The **destroy** command destroys a DPNI.

SYNTAX:

restool dpni destroy <*dpni-object*>

ARGUMENTS :

<*dpni-object*>

Specifies which DPNI to destroy.

EXAMPLE:

```
$ restool dpni destroy dpni.9
dpni.9 is destroyed
```

6.11.2.3 DPIO Commands

6.11.2.3.1 help command

The **help** command displays usage information for the DPIO object

SYNTAX:

restool dpio help

ARGUMENTS:

none

EXAMPLE:

```
    $ restool dpio help
    usage: restool dpio <command> [--help][ARGS…]
    Where <command> can be:
    info - displays detailed information about a DPIO object.
    create - creates a DPIO under the root DPRC
    destroy - destroys a DPIO under the root DPRC

    For command-specific help, use the --help option available for each command.
```

6.11.2.3.2 info command

The **info** command displays detailed information about a specific dpio object.

SYNTAX:

restool dpio info <*dpio-object*> [**--verbose**]

ARGUMENTS :

<*dpio-object*>

Specifies which dpio object to show detailed info for. The dpio-object argument is a

string specifying the object name—e.g. “dpio.7”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
# restool dpio info dpio.1
    dpio version: 3.0
    dpio id: 1
    plugged state: plugged
    offset of qbman software portal cache-enabled area: 0x20000
    offset of qbman software portal cache-inhibited area: 0x4020000
    qbman software portal id: 0x2
    dpio channel mode is: DPIO_LOCAL_CHANNEL
    number of priorities is: 0x8
    # restool dpio info dpio.1 --verbose
    dpio version: 3.0
    dpio id: 1
    plugged state: plugged
    offset of qbman software portal cache-enabled area: 0x20000
    offset of qbman software portal cache-inhibited area: 0x4020000
    qbman software portal id: 0x2
    dpio channel mode is: DPIO_LOCAL_CHANNEL
```

```
    number of priorities is: 0x8
    number of mappable regions: 2
    number of interrupts: 1
    interrupt 0's mask: 0
    interrupt 0's status: 0x8
```

6.11.2.3.3 create command

The **create** command creates a new DPIO. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpio create [OPTIONS]

OPTIONS :

--channel-mode=<*mode*>

Where <*mode*> is one of:

```
            DPIO_LOCAL_CHANNEL
            DPIO_NO_CHANNEL
```

Default value is DPIO_LOCAL_CHANNEL .

--num-priorities=<*number*>

Valid values for <*number*> are 1-8. Default value is 8.

EXAMPLE:

Create a DPIO with all default options:

$ restool dpio create

dpio.10 is created under dprc.1

Create a DPIO, specifying number of priorities:

```
    $ restool dpni create –num-priorities=4
    dpio.2 is created under dprc.1
```

6.11.2.3.4 destroy command

The **destroy** command destroys a DPIO.

SYNTAX:

restool dpio destroy <*dpio-object*>

ARGUMENTS :

<*dpio-object*>

Specifies which DPIO to destroy.

EXAMPLE:

```
    $ restool dpio destroy dpio.9
    dpio.9 is destroyed
```

6.11.2.4 DPSW Commands

text

6.11.2.4.1 help command

The **help** command displays usage information for the DPSW object

SYNTAX:

restool dpsw help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpsw help
    usage: restool dpsw <command> [--help][ARGS…]
    Where <command> can be:
    info - displays detailed information about a DPSW object.
    create - creates a DPSW under the root DPRC
    destroy - destroys a DPSW under the root DPRC

    For command-specific help, use the --help option available for each command.
```

6.11.2.4.2 info command

The **info** command displays detailed information about a specific dpsw object.

SYNTAX:

restool dpsw info <*dpsw-object*> [**--verbose**]

ARGUMENTS :

<*dpsw-object*>

Specifies which object to show detailed info for. The dpsw-object argument is a string

specifying the object name—e.g. “dpsw.2”

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
    $ restool dpsw info dpsw.1
dpsw version: 6.0
dpsw id: 1
plugged state: unplugged
endpoints:
endpoint state: -1
        interface 0: No object associated
endpoint state: -1
        interface 1: No object associated
endpoint state: -1
        interface 2: No object associated
endpoint state: -1
        interface 3: No object associated
dpsw_attr.options value is: 0x1
        DPSW_OPT_FLOODING_DIS
max VLANs: 8
max FDBs: 8
DPSW frame storage memory size: 0
```

```
number of interfaces: 4
current number of VLANs: 1
current number of FDBs: 1
```

6.11.2.4.3 create command

The **create** command creates a new DPSW. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpsw create --num-ifs=<*number*> [OPTIONS]

ARGUMENTS :

--num-ifs=<*number*>

Number of external and internal interfaces.

OPTIONS :

--options=<*options-mask*>

Where <*options-mask*> is a comma separated list of DPSW options:

```
            DPSW_OPT_FLOODING_DIS
```

```
            DPSW_OPT_MULTICAST_DIS
```

```
            DPSW_OPT_CTRL_IF_DIS
```

```
            DPSW_OPT_FLOODING_METERING_DIS
```

```
            DPSW_OPT_METERING_EN
```

--max-vlans=<*number*>

Maximum Number of VLAN's. Default is 16.

--max-fdbs=<*number*>

Maximum Number of FDB's. Default is 16.

--num-fdb-entries=<*number*>

Number of FDB entries. Default is 1024.

--fdb-aging-time=<*number*>

Default FDB aging time in seconds. Default is 300 seconds.

--max-fdb-mc-groups=<*number*>

Number of multicast groups in each FDB table. Default is 32.

EXAMPLE:

Create a 4-port switch with all default options:

```
    $ restool dpsw create --num-ifs=4
    dpsw.8 is created under dprc.1
```

Create a 4-port switch with options:

```
$ restool dpsw create –num-ifs=4 –max-vlans=8 –max-fdb-mc-groups=300
--options=DPSW_OPT_TC_DIS,DPSW_OPT_FLOODING_DIS
dpsw.2 is created under dprc.1
```

6.11.2.4.4 destroy command

The **destroy** command destroys a DPSW.

SYNTAX:

restool dpsw destroy <*dpsw-object*>

ARGUMENTS :

<*dpsw-object*>

Specifies which DPSW to destroy.

EXAMPLE:

```
$ restool dpsw destroy dpsw.8
dpsw.8 is destroyed
```

6.11.2.5 DPBP Commands

6.11.2.5.1 help command

The **help** command displays usage information for the DPBP object

SYNTAX:

restool dpbp help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpbp help
usage: restool dpbp <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPBP object.
create - creates a DPBP under the root DPRC
destroy - destroys a DPBP under the root DPRC

For command-specific help, use the --help option available for each command.
```

6.11.2.5.2 info command

The **info** command displays detailed information about a specific dpbp object.

SYNTAX:

restool dpbp info <*dpbp-object*> [**--verbose**]

ARGUMENTS :

<*dpbp-object*>

Specifies which dpbp object to show detailed info for. The dpbp-object argument is a

string specifying the object name—e.g. “dpbp.3”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpbp info dpbp.1
dpbp version: 2.0
dpbp id: 1
plugged state: plugged
buffer pool id: 0
```

6.11.2.5.3 create command

The **create** command creates a new DPBP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpbp create [OPTIONS]

OPTIONS:

--container=<*container_name*>

Specifies the parent container name. e.g. dprc.2, dprc.3 etc.

If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPBP:

```
$ restool dpbp create
dpbp.2 is created under dprc.1
```

6.11.2.5.4 destroy command

The **destroy** command destroys a DPBP.

SYNTAX:

restool dpbp destroy <*dpbp-object*>

ARGUMENTS :

<*dpbp-object*>

Specifies which DPBP to destroy.

EXAMPLE:

```
$ restool dpbp destroy dpbp.2
dpbp.2 is destroyed
```

6.11.2.6 DPCON Commands

text

6.11.2.6.1 help command

The **help** command displays usage information for the DPCON object.

SYNTAX:

restool dpcon help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpcon help
usage: restool dpcon <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPCON object.
create - creates a DPCON under the root DPRC
destroy - destroys a DPCON under the root DPRC

For command-specific help, use the --help option available for each command.
```

6.11.2.6.2 info command

The **info** command displays detailed information about a specific dpcon object.

SYNTAX:

restool dpcon info <*dpcon-object*> [**--verbose**]

ARGUMENTS :

<*dpcon-object*>

Specifies which dpcon object to show detailed info for. The dpcon-object argument is

a string specifying the object name—e.g. “dpcon.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpcon info dpcon.1
dpcon version: 2.0
dpcon id: 1
plugged state: plugged
qbman channel id to be used by dequeue operation: 40
number of priorities for the DPCON channel: 8
```

6.11.2.6.3 create command

The **create** command creates a new DPCON. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpcon create [OPTIONS]

OPTIONS :

--num-priorities=<*number*>

Specifies the number of priorities, valid values are 1-8. Default is 1.

EXAMPLE:

Create a DPCON with 4 priorities:

```
$ restool dpcon create --num-priorites=4
dpcon.8 is created under dprc.1
```

6.11.2.6.4 destroy command

The **destroy** command destroys a DPCON.

SYNTAX:

restool dpcon destroy <*dpcon-object*>

ARGUMENTS :

<*dpcon-object*>

Specifies which DPCON to destroy.

EXAMPLE:

```
$ restool dpcon destroy dpcon.9
```

dpcon.9 is destroyed

6.11.2.7 DPCI Commands

6.11.2.7.1 help command

The **help** command displays usage information for the DPCI object.

SYNTAX:

restool dpci help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpci help
usage: restool dpci <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPCI object.
create - creates a DPCI under the root DPRC
destroy - destroys a DPCI under the root DPRC

For command-specific help, use the --help option available for each command.
```

6.11.2.7.2 info command

The **info** command displays detailed information about a specific dpci object.

SYNTAX:

restool dpci info <*dpci-object*> [**--verbose**]

ARGUMENTS :

<*dpci-object*>

Specifies which dpci object to show detailed info for. The dpci-object argument is a

string specifying the object name—e.g. “dpci.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpci info dpci.1
dpci version: 2.0
dpci id: 1
plugged state: plugged
num_of_priorities: 2
connected peer: dpci.4
peer's num_of_priorities: 2
link status: 0 - down
```

6.11.2.7.3 create command

The **create** command creates a new DPCI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpci create [OPTIONS]

OPTIONS :

--num-priorities=<*number*>

Specifies the number of priorities, valid values are 1 or 2. Default is 1.

EXAMPLE:

Create a DPCI with 4 priorities:

```
$ restool dpci create --num-priorites=2
dpci.8 is created under dprc.1
```

6.11.2.7.4 destroy command

The **destroy** command destroys a DPCI.

SYNTAX:

restool dpci destroy <*dpci-object*>

ARGUMENTS :

<*dpci-object*>

Specifies which DPCI to destroy.

EXAMPLE:

```
$ restool dpci destroy dpci.9
```

dpci.9 is destroyed

6.11.2.8 DPSECI Commands

6.11.2.8.1 help command

The **help** command displays usage information for the DPSECI object.

SYNTAX:

restool dpseci help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpseci help
usage: restool dpseci <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPSECI object.
create - creates a DPSECI under the root DPRC
destroy - destroys a DPSECI under the root DPRC
For command-specific help, use the --help option available for each command.
```

6.11.2.8.2 info command

The **info** command displays detailed information about a specific dpseci object.

SYNTAX:

restool dpseci info <*dpseci-object*> [**--verbose**]

ARGUMENTS :

<*dpseci-object*>

Specifies which dpseci object to show detailed info for. The dpseci-object argument is

a string specifying the object name—e.g. “dpseci.8”

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpseci info dpseci.1
dpseci version: 2.0
dpseci id: 1
plugged state: plugged
number of priorities: 1
dpci id: 1
```

6.11.2.8.3 create command

The **create** command creates a new DPSECI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpseci create [OPTIONS]

OPTIONS :

--priorities=<*priority1,priority2*>

DPSEC support 2 priorities that can be individually set. Valid values for <*priority1*> and <*priority2*> are 1-8. Default is 1.

EXAMPLE:

Create a DPSECI with 4 priorities:

```
$ restool dpseci create --priorites=2,4
dpseci.9 is created under dprc.1
```

6.11.2.8.4 destroy command

The **destroy** command destroys a DPSECI.

SYNTAX:

restool dpseci destroy <*dpseci-object*>

ARGUMENTS :

<*dpseci-object*>

Specifies which DPSECI to destroy.

EXAMPLE:

```
$ restool dpseci destroy dpseci.9
```

dpseci.9 is destroyed

6.11.2.9 DPDMUX Commands

6.11.2.9.1 help command

The **help** command displays usage information for the DPDMUX object.

SYNTAX:

restool dpdmux help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpdmux help
usage: restool dpdmux <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPDMUX object.
create - creates a DPDMUX under the root DPRC
destroy - destroys a DPDMUX under the root DPRC

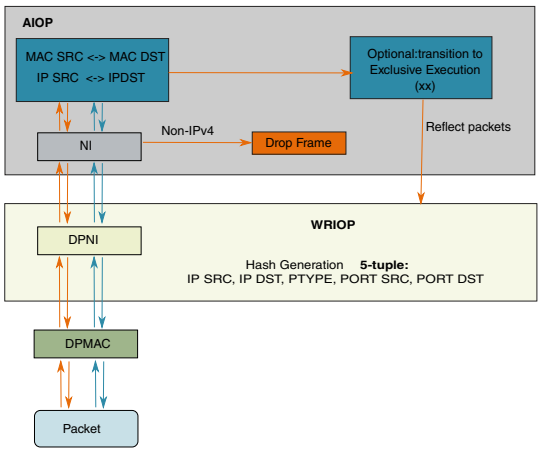
For command-specific help, use the --help option available for each command.
```

6.11.2.9.2 info command

The **info** command displays detailed information about a specific dpdmux object.

SYNTAX:

restool dpdmux info <*dpdmux-object*> [**--verbose**]

ARGUMENTS :

<*dpdmux-object*>

Specifies which dpdmux object to show detailed info for. The dpdmux-object

argument is a string specifying the object name—e.g. “dpdmux.2”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpdmux info dpdmux.0
dpdmux version: 4.1
dpdmux id: 0
plugged state: plugged
endpoints:
endpoint state: 0
        interface 0: dpmac.1, link is down
endpoint state: 0
        interface 1: dpni.0, link is down
endpoint state: 0
        interface 2: dpni.1, link is down
dpdmux_attr.options value is: 0x2
        DPDMUX_OPT_BRIDGE_EN
DPDMUX address table method: DPDMUX_METHOD_MAC
DPDMUX manipulation type: DPDMUX_MANIP_NONE
number of interfaces (excluding the uplink interface): 3
DPDMUX frame storage memory size: 0
control interface ID: 0
```

6.11.2.9.3 create command

The create command creates a new DPDMUX. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpdmux create --num-ifs=<*number*> [OPTIONS]

ARGUMENTS :

--num-ifs=<*number*>

Number of virtual interfaces (excluding the uplink interface).

OPTIONS :

--method=<*dmat_method*>

Where <*dmat_method*> defines the method of the DPDMUX address table. A valid value is one of the following:

```
            DPDMUX_METHOD_NONE
            DPDMUX_METHOD_C_VLAN_MAC
            DPDMUX_METHOD_MAC
            DPDMUX_METHOD_C_VLAN
            DPDMUX_METHOD_S_VLAN
```

Default is `DPDMUX_METHOD_C_VLAN_MAC`

--manip=<*manip*>

Where <*manip*> defines the DPDMUX required manipulation operation. A valid value is one of the following:

```
            DPDMUX_MANIP_NONE
```

```
            DPDMUX_MANIP_ADD_REMOVE_S_VLAN
```

Default is DPDMUX_MANIP_NONE

--options=<*options-mask*>

```
            DPDMUX_OPT_BRIDGE_EN
```

Default is 0 (don't set any options)

--max-dmat-entries=<*number*>

max entries in DPDMUX address table. Default is 64.

--max-mc-groups=<*number*>

Number of multicast groups in DPDMUX table. Default is 32 groups.

EXAMPLE:

Create a DPDMUX with all default options:

```
$ restool dpdmux create --num-ifs=4
    dpdmux.11 is created under dprc.1
```

6.11.2.9.4 destroy command

The **destroy** command destroys a DPDMUX.

SYNTAX:

restool dpdmux destroy <*dpdmux-object*>

ARGUMENTS :

<*dpdmux-object*>

Specifies which DPDMUX to destroy.

EXAMPLE:

```
    $ restool dpdmux destr
```

6.11.2.10 DPMCP Commands

6.11.2.10.1 help command

The **help** command displays usage information for the DPMCP object.

SYNTAX:

restool dpmcp help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpmcp help
usage: restool dpmcp <command> [--help][ARGS…]
Where <command> can be:
    info - displays detailed information about a DPMCP object.
    create - creates a DPMCP under the root DPRC
    destroy - destroys a DPMCP under the root DPRC

For command-specific help, use the --help option available for each command.
```

6.11.2.10.2 info command

The **info** command displays detailed information about a specific dpmcp object.

SYNTAX:

restool dpmcp info <*dpmcp-object*> [**--verbose**]

ARGUMENTS :

<*dpmcp-object*>

Specifies which dpmcp object to show detailed info for. The dpmcp-object argument

is a string specifying the object name—e.g. “dpmcp.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpmcp info dpmcp.5
dpmcp version: 1.0
dpmcp object id/portal id: 5
plugged state: plugged
```

6.11.2.10.3 create command

The **create** command creates a new DPMCP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpmcp create

EXAMPLE:

Create a DPMCP:

```
$ restool dpmcp create
dpmcp.15 is created under dprc.1
$ restool dpmcp create
MC error: No resource (status 0x8)
// when you see this error, it usually means no free portal available at this time.
```

6.11.2.10.4 destroy command

The **destroy** command destroys a DPMCP.

SYNTAX:

restool dpmcp destroy <*dpmcp-object*>

ARGUMENTS :

<*dpmcp-object*>

Specifies which DPMCP to destroy.

EXAMPLE:

```
$ restool dpmcp destroy dpmcp.9
dpmcp.9 is destroyed
```

6.11.2.11 DPMAC Commands

6.11.2.11.1 help command

The **help** command displays usage information for the DPMAC object.

SYNTAX:

restool dpmac help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpmac help
usage: restool dpmac <command> [--help][ARGS…]
Where <command> can be:
    info - displays detailed information about a DPMAC object.
    create - creates a DPMAC under the root DPRC
    destroy - destroys a DPMAC under the root DPRC

For command-specific help, use the --help option available for each command.
```

6.11.2.11.2 info command

The **info** command displays detailed information about a specific dpmac object.

SYNTAX:

restool dpmac info <*dpmac-object*> [**--verbose**]

ARGUMENTS:

<*dpmac-object*>

Specifies which dpmac object to show detailed info for. The dpmac-object argument

is a string specifying the object name—e.g. “dpmac.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpmac info dpmac.5
dpmcp version: 2.0
```

```
    dpmac object id/phy id: 5
    plugged state: plugged
```

6.11.2.11.3 create command

The **create** command creates a new DPMAC. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpmac create --mac-id=<*number*>

--mac-id=<*number*>

Specifies the mac id.

EXAMPLE:

Create a DPMAC with valid portal id:

```
    $ restool dpmac create --mac-id=15
    dpmac.15 is created under dprc.1
```

6.11.2.11.4 destroy command

The **destroy** command destroys a DPMAC.

SYNTAX:

restool dpmac destroy <*dpmac-object*>

ARGUMENTS :

<*dpmac-object*>

Specifies which DPMAC to destroy.

EXAMPLE:

```
    $ restool dpmac destroy dpmac.9
    dpmac.9 is destroyed
```

6.11.2.12 DPDCEI Commands

6.11.2.12.1 help command

The **help** command displays usage information for the DPDCEI object.

SYNTAX:

restool dpdcei help

ARGUMENTS:

none

EXAMPLE:

```
    $ restool dpdcei help
    usage: restool dpdcei <command> [--help][ARGS…]
    Where <command> can be:
    info - displays detailed information about a DPDCEI object.
    create - creates a DPDCEI under the root DPRC
    destroy - destroys a DPDCEI under the root DPRC
```

```
For command-specific help, use the --help option available for each command.
```

6.11.2.12.2 info command

The **info** command displays detailed information about a specific dpdcei object.

SYNTAX:

restool dpdcei info <*dpdcei-object*> [**--verbose**]

ARGUMENTS :

<*dpdcei-object*>

Specifies which dpdcei object to show detailed info for. The dpdcei-object argument

is a string specifying the object name, e.g. “dpdcei.2”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpdcei info dpdcei.5
dpdcei version: 0.0
dpdcei id: 5
plugged state: plugged
DPDCEI engine: DPDCEI_ENGINE_COMPRESSION
```

6.11.2.12.3 create command

The **create** command creates a new DPDCEI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpdcei create --engine=<*engine*> --priority=<*number*>

--engine=<*engine*>

Compression or decompression engine to be selected.

A valid value is one of the following:

```
            DPDCEI_ENGINE_COMPRESSION
```

```
            DPDCEI_ENGINE_DECOMPRESSION
```

--priority=<*number*>

Priority for DCE hardware processing (valid values 1-8)

EXAMPLE:

Create a DPDCEI:

```
$ restool dpdcei create --engine=DPDCEI_ENGINE_COMPRESSION --priority=2
dpdcei.0 is created under dprc.1
$ restool dpdcei create --engine=DPDCEI_ENGINE_COMPRESSION --priority=3
dpdcei.1 is created under dprc.1
```

6.11.2.12.4 destroy command

The **destroy** command destroys a DPDCEI.

SYNTAX:

restool dpdcei destroy <*dpdcei-object*>

ARGUMENTS :

<*dpdcei-object*>

Specifies which DPDCEI to destroy.

EXAMPLE:

```
$ restool dpdcei destroy dpdcei.9
dpdcei.9 is destroyed
```

6.11.2.13 DPAIOP Commands

6.11.2.13.1 help command

The **help** command displays usage information for the DPAIOP object.

SYNTAX:

restool dpaiop help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpaiop help
usage: restool dpaiop <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPAIOP object.
create - creates a DPAIOP under the root DPRC
destroy - destroys a DPAIOP under the root DPRC

For command-specific help, use the --help option available for each command.
```

6.11.2.13.2 info command

The **info** command displays detailed information about a specific dpaiop object.

SYNTAX:

restool dpaiop info <*dpaiop-object*> [**--verbose**]

ARGUMENTS :

<*dpaiop-object*>

Specifies which dpaiop object to show detailed info for. The *dpaiop-object* argument

is a string specifying the object name—e.g. “dpaiop.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpaiop info dpaiop.5
```

dpmcp version: 1.0

dpmcp id: 5

plugged state: plugged

dpaiop server layer version: 2.1.3

DPAIOP state: DPAIOP_STATE_RUNNING

6.11.2.13.3 create command

The **create** command creates a new DPAIOP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpaiop create --aiop-id=<*number*> --aiop-container=<*container-name*>

ARGUMENTS :

--aiop-container=<*container-name*>

Specifies the AIOP container name, e.g. dprc.3, dprc.4, etc.

OPTIONS :

--aiop-id=<*number*>

Specifies the AIOP ID. Currently aiop container could only hold one dpaiop. Valid

number is 0. Default number is 0.

EXAMPLE:

Create a DPAIOP:

```
$ restool dpaiop create --aiop-id=0 --aiop-container=dprc.3
dpaiop.0 is created under dprc.3

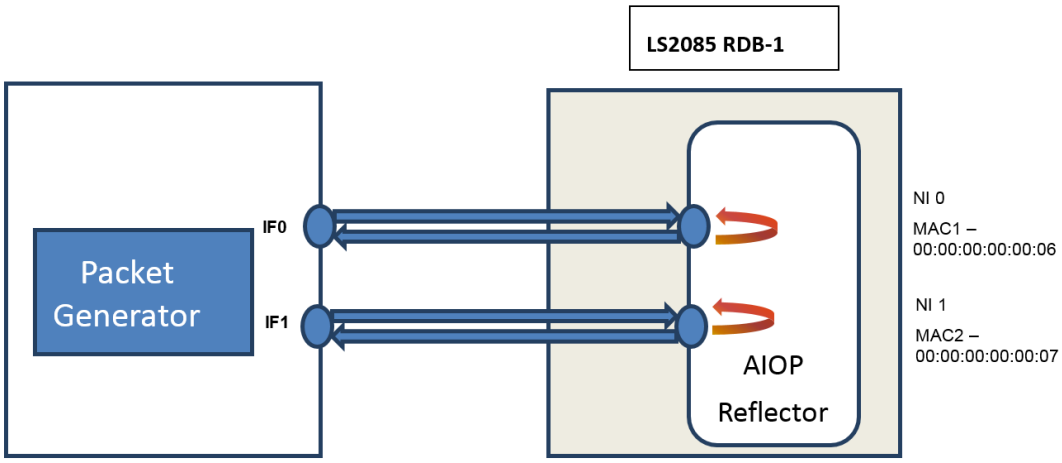
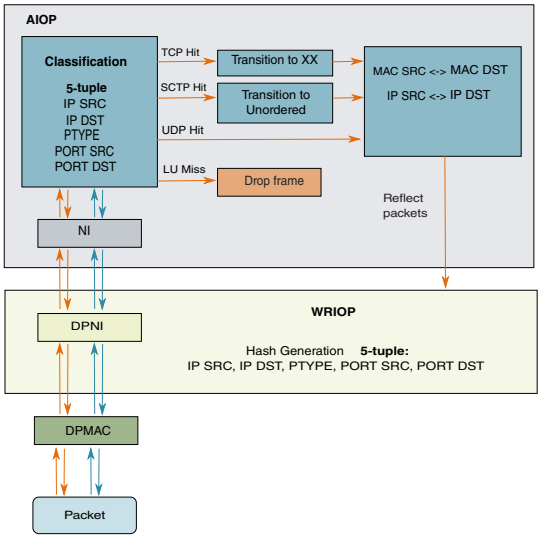
$ restool dpaiop create --aiop-container=dprc.3
dpaiop.0 is created under dprc.3
```

6.11.2.13.4 destroy command

The **destroy** command destroys a DPAIOP.

SYNTAX:

restool dpaiop destroy <*dpaiop-object*>

ARGUMENTS :

<*dpaiop-object*>

Specifies which DPAIOP to destroy.

EXAMPLE:

```
$ restool dpio destroy dpaiop.9
dpaiop.9 is destroyed
```

6.11.2.14 Limitation

CR:ENGR00364876 DPNI: calling dprc_disconnect on a DPNI<->DPNI connection isn't fully disconnected

CR:ENGR00364611 0x3e8 error code for connect command when connection is already present

CR:ENGR00364610 allow dprc connect/disconnect across different DPRCs

Objects that are in use (i.e. plugged) should not be destroyed, but restool and MC allows this, resulting in an undefined result. Now restool does not allow plugged objects to be destroyed.

CR: ENGR00356070 - “dpsw create” fail, “dpdmux create” has limited functionality, In MC firmware 8.0.0, failure in dpmac create, dpdmux create/destroy, dpseci create, dpsw create

CR: ENGR00342091 – “dpdmux create” issue

CR:ENGR00356428 - dprc unassign does not work

CR: ENGR00356708 - mc hang before mcp resources are exhausted, there is only 256 (0-255) portals availvle while it claims 512 MC portals (0-511)

CR: ENGR00361302 - dprc's label has garbage info

CR:ENGR00361796 – cannot create dpseci

ENGR361940 kernel crashes with restool v0.9

6.12 DPAA2 User Manual

DPAA2 is a hardware-level networking architecture found on some NXP SoCs. This section provides technical information on this architecture mainly for software developers.

6.12.1 DPAA2 User Manual PDF

Click here to access the DPAA2 User Manual PDF.

6.13 DPAA2 API Reference Manual

The following sections contains general APIs for DPAA2.

6.13.1 DPAA2 API Reference Manual PDF

Click here to access the DPAA2 API Reference Manual PDF.

6.14 AIOP

6.14.1 AIOP Sample Applications

The Advanced I/O Processor (AIOP) hardware, an optional component of the DPAA2 architecture, is a C-programmable engine that enables power efficient packet-oriented processing. This section provides sample applications that can be used by customers to exercise functionality of offloading packet processing in AIOP.

6.14.1.1 Creating AIOP containers

This section describes how to dynamically create Data Path Resource Containers owned by AIOP and how to load AIOP ELF by using AIOP Tool.

cd /usr/aiop/scripts

In this folder there are two scripts based on restool available:

- `dynamic_aiop_only.sh` - script used by reference applications only with AIOP side, not interacting with other components
- `dynamic_aiop_root.sh` - script used by reference applications that interact with Linux Kernel running on GPPS

Scripts do not require any command line argument as input. Sample execution flow is shown below:

```
# ./dynamic_aiop_only.sh
Creating AIOP Container
Assigned dpbp.1 to dprc.2
Assigned dpbp.2 to dprc.2
Assigned dpbp.3 to dprc.2
Assigned dpni.1 to dprc.2
Connecting dpni.1<------->dpmac.1
Assigned dpni.2 to dprc.2
Connecting dpni.2<------->dpmac.2
AIOP Container dprc.2 created
----- Contents of AIOP Container: dprc.2 -----
dprc.2 contains 5 objects:
object          label           plugged-state
dpni.2                          plugged
dpni.1                          plugged
dpbp.3                          plugged
dpbp.2                          plugged
dpbp.1                          plugged
-----
==================================================
Creating AIOP Tool Container
Assigned dpaiop.0 to dprc.3
Assigned dpmcp.22 to dprc.3
AIOP Tool Container dprc.3 created
----- Contents of AIOP Tool Container: dprc.3 -----
dprc.3 contains 2 objects:
object          label           plugged-state
dpaiop.0                        plugged
dpmcp.22                        plugged
-----
==================================================
Performing VFIO mapping for AIOP Tool Container (dprc.3)
Performing vfio mapping for dprc.3
[  796.531485] vfio-fsl-mc dprc.3: Binding with vfio-fsl_mc driver
[  796.540756] vfio-fsl-mc dpaiop.0: Binding with vfio-fsl_mc driver
[  796.547364] vfio-fsl-mc dpmcp.22: Binding with vfio-fsl_mc driver
========== Summary ================================
 AIOP Container: dprc.2
 AIOP Tool Container: dprc.3
==================================================
```

To load the AIOP binary by using AIOP Tool, the AIOP Tool Container is necessary. In the above case is `dprc.3`. The following command should be executed:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_reflector.elf
AIOP Image (aiop_reflector.elf) loaded successfully.
```

To ensure that AIOP image is indeed loaded and running, the AIOP console can be checked by executing one of the following commands:

```
# aiop_tool status -g dprc.3
AIOP Tile Status:
         Major Version: 2, Minor Version: 1
         Service Layer:- Major Version: 0, Minor Version: 7, Revision: 1
         State: DPAIOP_STATE_RUNNING

root@ls2085ardb:~# cat /dev/fsl_aiop_console
        . . .
```

6.14.1.2 AIOP Packet Reflector application

6.14.1.2.1 AIOP Packet Reflector overview

The purpose of this sample application is to demonstrate a simple application data path on the AIOP. The application is a basic reflector for every IPv4 frame and works much like the NADK Packet Reflector application, except that it runs on the AIOP.

The application performs the following functions:

- Configure the fields used for the initial order scope hash generation. The fields are: the source address, the destination address, the protocol type fields from the IP header and the source port and the destination port from the L4 header. For every packet received by WRIOP a hashed Initial Ordering Scope (IOS) will be generated based on these values.
- Set Concurrent Execution (XC) as the initial packets processing mode. In the initial stage of processing, the packets are processed concurrently by many cores in their ordering scope.
- Drop non IPv4 packets.
- Switch the source and destination MAC and IP addresses of the received packets.
- Transition into Exclusive execution (XX) in order to restore packet order. This is optional and can be deactivated through define EXCLUSIVE_MODE
- Reflect back the packet on the same interface from which it was received.



Figure 31. AIOP Packet Reflector overview

6.14.1.2.2 Running the Reflector application

The reflector application should be runnning on the RDB.

The ELF for AIOP reflector is `/usr/aiop/bin/aiop_reflector.elf`. This can be loaded via AIOP Tool as section Creating AIOP Containers describes. For this application the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP Reflector Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_reflector.elf
```

To check if the AIOP Reflector Application loaded successfully, execute the following command in the Linux command shell:

```
# cat /dev/fsl_aiop_console | grep REFLECTOR
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g. NI 0) together with the associated MAC address.

```
REFLECTOR : Successfully configured ni0 (dpni.2)
REFLECTOR : dpni.2 <---connected---> dpmac.2 (MAC addr: 00:00:00:00:00:07)
REFLECTOR : Successfully configured ni1 (dpni.1)
REFLECTOR : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

NOTE

Although the AIOP container contains multiple three DPNIs (DPNI6, DPNI7 and DPNI10), the AIOP Reflector will use only the DPNIs that have a DPMAC as endpoint (on which it can succesfully configure the order scope). The others will be skipped in application initialization.

During frame processing, the AIOP Logger will print the following brief information about every reflected packet:

- AIOP Core number on which the frame was processed
- Received MAC source and destination addresses
- Received IP source and destination addresses

```
# busybox tail –f /dev/fsl_aiop_console

RX on NI 0 | CORE:15
      MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
      IP_SRC: 192.85.1.1 IP_DST: 192.0.0.1

RX on NI 0 | CORE:14
      MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
      IP_SRC: 192.85.1.2 IP_DST: 192.0.0.1

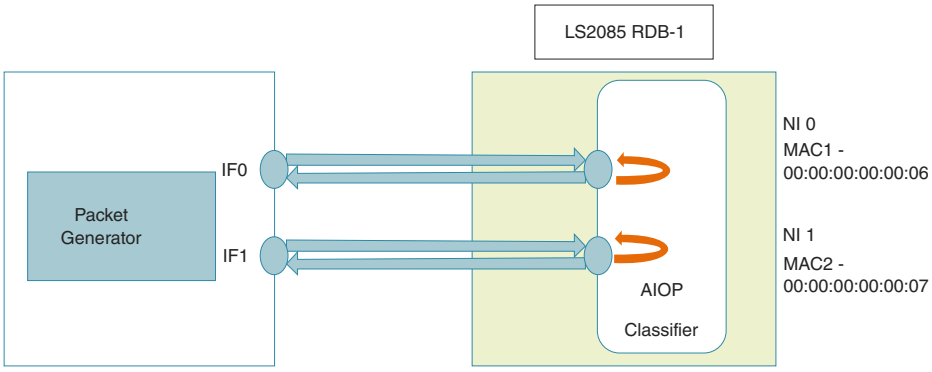
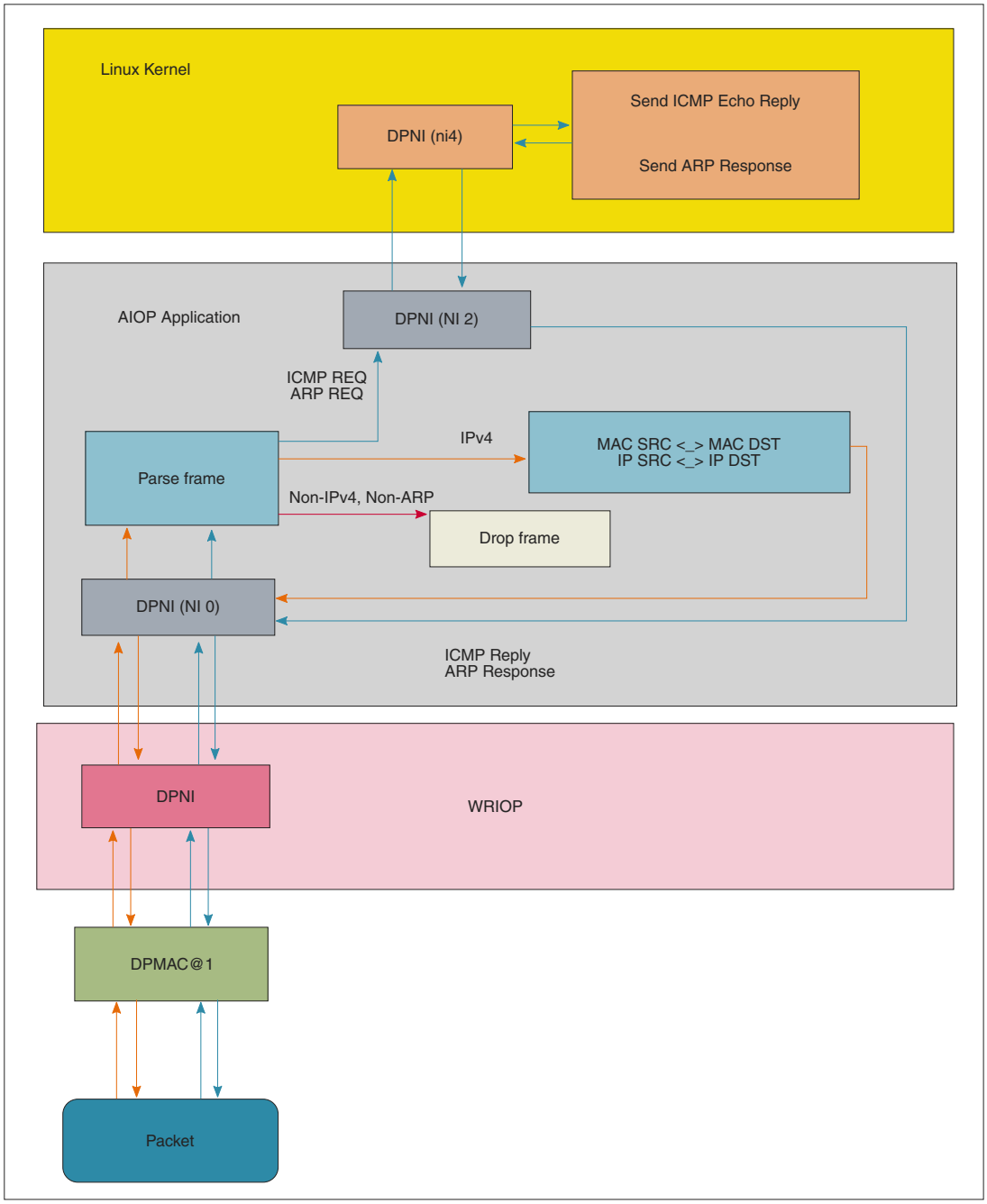
 . . .

RX on NI 0 | CORE:9
      MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
      IP_SRC: 192.85.1.3 IP_DST: 192.0.0.1
```

6.14.1.2.3 Generating traffic to test AIOP Reflector application

An external traffic source is needed to pass traffic to the reflector application. This could be another RDB or some other traffic generator. Two optical 10G ports need to be connected to MAC1 and MAC2 of RDB-1.

The following graphic illustrates how traffic moves from a packet generator to RDB-1 board



Figure 32. Generating traffic for AIOP Packet Reflector

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1 and a second port to MAC2 of RDB-1.

On RDB-1 ensure that both AIOP network interfaces are in link up state:

```
$ cat /dev/fsl_aiop_console | grep REFLECTOR
REFLECTOR : ni0 link is UP
REFLECTOR : ni1 link is UP
```

Generate an IPv4 frames for each of the two ports:

```
        MAC source:         ANY
        MAC destination:    00:00:00:00:00:06 for MAC1
                            00:00:00:00:00:07 for MAC2
        IP source:          ANY
        IP destination:     ANY
```

On RDB-1, the AIOP Logger will print a brief information about every frame that is being reflected.

6.14.1.3 AIOP Packet Classifier application

6.14.1.3.1 AIOP Packet Classifier overview

The purpose of this sample application is to demonstrate how to perform a simple Classification on the AIOP. The application will apply a Classification criteria for every IPv4 frame and will reflect back only accepted frames.

There are three execution modes listed below:

- Exclusive execution (XX): This mode forces atomic processing of packets. Each packet is processed in order of its arrival before the next packet is processed.
- Concurrent execution (XC): This mode processes packets within a flow concurrently on the AIOP. Packets may become misordered as part of this parallel processing. To restore packet order, the data path transitions to exclusive execution mode before transmitting the forwarded packets.
- Unordered. In this mode, packet ordering is not considered and concurrent packet processing takes place.

The application performs the following functions:

- Put network interfaces in promiscuous mode in order to allow packet reception regardless of its MAC destination address.

- Configure the fields used for the initial order scope hash generation. The fields are: the source address, the destination address, the protocol type fields from the IP header and the source port and the destination port from the L4 header. For every packet received by WRIOP a hashed Initial Ordering Scope (IOS) will be generated based on these values.
  - Set Concurrent Execution (XC) as the initial packets processing mode. In the initial stage of processing, the packets are processed concurrently by many cores in their ordering scope.
  - Classify the packets to enable different processing modes to be run based on the type of traffic. Classification is based on the following fields: IPv4 source and destination addresses, Protocol number, Source and Destination Layer 4 ports. The packets processing mode is selected as a result of the classification as follows:
  - TCP packets are processed in exclusive execution mode (XX)
  - UDP packets are processed in concurrent execution mode (XC). In order to keep the UDP packets in order, the application must transition to exclusive execution mode before sending packets to the destination port.
  - SCTP packets are processed with no respect regarding their arrival order (Unordered).
  - The packets not matching the classification criteria are dropped.

In order to determine the processing mode in which a packet should be processed, when a lookup hit is obtained, the lookup result will return the new processing mode to which the application should transition

- Switch the source and destination MAC and IP addresses of the received packets.

Reflect back the accepted packets on the same interface from which they were received



Figure 33. AIOP Packet Classifier overview

The Exact Match table in AIOP that will be used for table lookup will be populated with the following entries:

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
198.20.1.1	198.19.1.0	6 (TCP)	1024	1025
198.20.1.1	198.19.1.64	17 (UDP)	1024	1025
198.20.1.1	198.19.1.128	132 (SCTP)	1024	1025

For each entry in the table a mask with value 0xE0 is applied on the last byte of the IP Destination address, allowing a number of 32 IP Destination Address to be received. The following traffic flows will generate a HIT in the Classification table:

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
198.20.1.1	198.19.1.0 .. 198.19.1.31	6 (TCP)	1024	1025
198.20.1.1	198.19.1.64 .. 198.19.1.95	17 (UDP)	1024	1025
198.20.1.1	198.19.1.128 .. 198.19.1.159	132 (SCTP)	1024	1025

6.14.1.3.2 Running the Classifier application

The classifier application should be run on the RDB-1 system.

The ELF for AIOP Classifier is `/usr/aiop/bin/aiop_classifier.elf`. This can be loaded via AIOP tool as section Creating AIOP Containers describes. For this application the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP Classifier Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_classifier.elf
```

To check if the AIOP Classifier Application loaded successfully, execute the following command in the Linux command shell:

```
# cat /dev/fsl_aiop_console | grep CLASSIFIER
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g NI 0) together with the associated MAC address.

```
CLASSIFIER : Successfully configured exact table match
CLASSIFIER : Successfully configured ni0 (dpni.2)
CLASSIFIER : dpni.2 <---connected---> dpmac.2 (MAC addr: 00:00:00:00:00:07)
CLASSIFIER : Successfully configured ni1 (dpni.1)
CLASSIFIER : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

NOTE

Although the AIOP container contains multiple NIs the AIOP Reflector will use only the NIs that have a DPMAC as endpoint (on which it can succesfully configure the order scope).

During frame processing, the AIOP Logger will print the following brief information about every reflected packet:

- AIOP Core number on which the frame was processed
- Received MAC source and destination addresses

- Received IP source and destination addresses

```
# busybox tail -f /dev/fsl_aiop_console

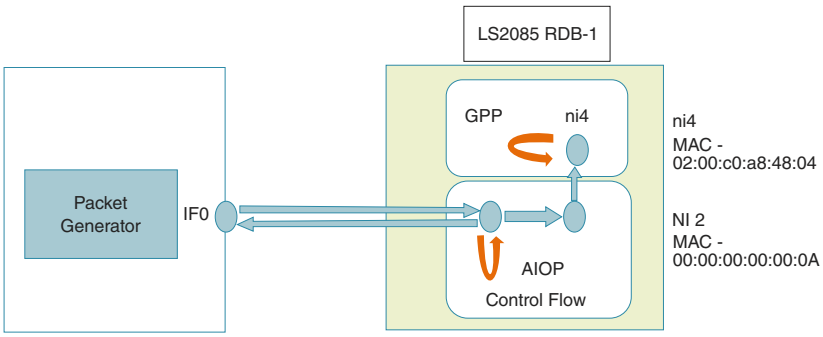
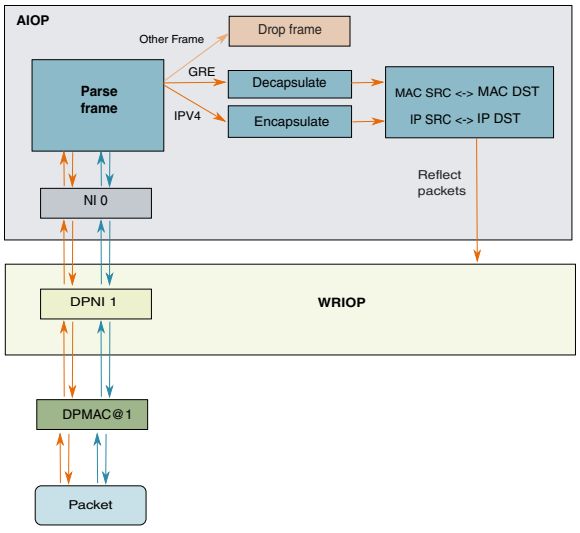
RX on NI 0 | CORE:15
   MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
   IP_SRC: 198.20.1.1 IP_DST: 198.19.1.0

RX on NI 0 | CORE:14
   MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
   IP_SRC: 198.20.1.1 IP_DST: 198.19.1.64

. . .

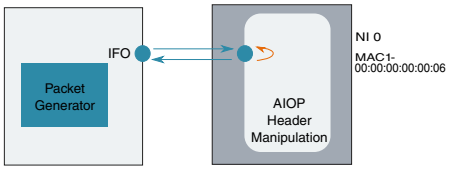
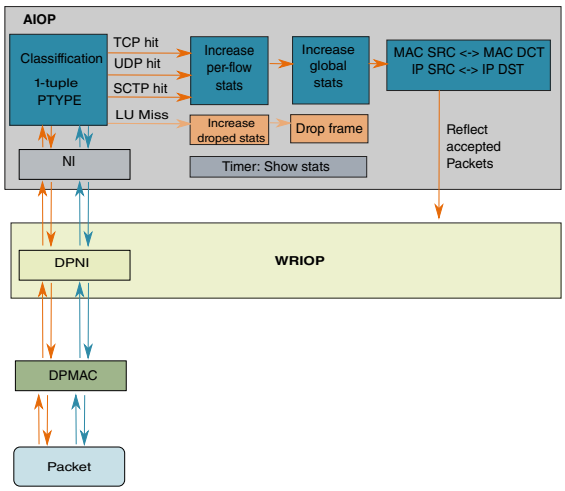
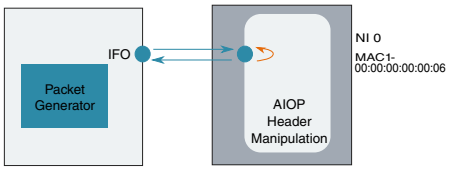
RX on NI 1 | CORE:9
   MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-07
   IP_SRC: 198.20.1.1 IP_DST: 198.19.1.128
```

6.14.1.3.3 Generating traffic to test AIOP Classifier application

An external traffic source is needed to pass traffic to the classifier application. This could be another RDB or some other traffic generator. Two optical 10G ports need to be connected to MAC1 and MAC2 of RDB-1.

The following graphic illustrates how traffic moves from a packet generator to RDB-1 board.



Figure 34. Generating traffic for AIOP Classifier

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1 and a second port to MAC2 of RDB-1.

On RDB-1 ensure that both AIOP network interfaces are in link up state:

```
$ cat /dev/fsl_aiop_console | grep CLASSIFIER
...
CLASSIFIER : ni0 link is UP
CLASSIFIER : ni1 link is UP
```

Generate IPv4 frames for each of the two ports:

- MAC source: ANY
- MAC destination: ANY

- For IP & Layer 4, the accepted values are show in the following table:

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
198.20.1.1	198.19.1.0 .. 198.19.1.31	6 (TCP)	1024	1025
198.20.1.1	198.19.1.64 .. 198.19.1.95	17 (UDP)	1024	1025
198.20.1.1	198.19.1.128 .. 198.19.1.159	132 (SCTP)	1024	1025

The AIOP Logger will print a brief information about every frame that passed the Classification criteria.

NOTE

Traffic can be generated using frames available in the provided classifier.pcap file. The traffic available is shown in the table below.

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
198.20.1.1	198.19.1.0 .. 198.19.1.63	6 (TCP)	1024	1025
198.20.1.1	198.19.1.64 .. 198.19.1.127	17 (UDP)	1024	1025
198.20.1.1	198.19.1.128 .. 198.19.1.191	132 (SCTP)	1024	1025

This file can be used to generate the traffic from a traffic generator (e.g.: a packet generator or another RDB board). Location on board for this file, after bringup, is in `/usr/aiop/traffic_files/classifier.pcap`. On RDB-1, the AIOP Reflector Classifier Application will drop half of the frames in each range as a result of classification look-up miss.

6.14.1.4 AIOP Control Flow application

6.14.1.4.1 AIOP Control Flow overview

The purpose of this sample application is to demonstrate a simple AIOP-GPP communication using a DPNI-DPNI connection. The application filters ICMP and ARP request frames and sends them to GPP where Linux Kernel replies with ICMP Echo Reply and ARP Response.

Application has the following required connections:

- DPNI to DPNI connection: a network interface from AIOP that provides connection to another network interface on GPP
- DPNI to DPMAC connection: a network interface from AIOP connected to external traffic (physical wired connection, e.g. XFI or SGMII as links)

The application performs the following functions:

Initialization:

All network interfaces are in promiscuous mode.

Depending on a network interface's' connected endpoint object type, there are two different frame processing callbacks.

NOTE

Failures in performing the above actions lead to dropping the packets received on that network interface or in no link between AIOP and GPP.

Runtime

For every packet received on AIOP network interface connected to external traffic:

- Drop non IPv4 packets and non ARP packets
- Detect, using Parser, if packets received are either ICMP Echo Request or ARP Request. In this case send packets to Linux Kernel using GPP connection
- For ARP or IPv4 packets other than ICMP Echo Request and ARP Request: switch the source and destination MAC and IP addresses of the received packets and reflect back the packet on the same interface from which it was received.

For every packet received from GPP:

- Drop all packets that are not ICMP Echo Reply and not ARP response
- Print brief information about the accepted packets
- Forward ICMP Echo Reply and ARP response packets to the NI havingconnected to traffic generatorDPMAC as endpoint and callback config and callback configured.



Figure 35. AIOP Control Flow overview

6.14.1.4.2 Running the Control Flow application

The ELF for AIOP Classifier is `/usr/aiop/bin/aiop_control_flow.elf`. This can be loaded via AIOP tool as section Creating AIOP Containers describes. For this application the script `dynamic_aiop_root.sh` must be executed before loading the image:

```
# ./dynamic_aiop_root.sh
```

To load the AIOP Classifier Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_control_flow.elf
```

To check if AIOP Control Flow Application loaded successfully execute the following command in the Linux command shell:

```
$ cat /dev/fsl_aiop_console | grep CONTROL_FLOW
```

The command output should display the number of NIs that were successfully configured together with the specific MAC addresses that are provided to the AIOP Control Flow Application:

```
CONTROL_FLOW : Successfully configured ni0 (dpni.2)
CONTROL_FLOW : dpni.2 <---connected---> dpni.0
CONTROL_FLOW : Successfully configured ni1 (dpni.1)
CONTROL_FLOW : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

On RDB board, to enable AIOP-GPP communication via DPNI-DPNI, it is required to configure the Linux network interface:

```
# ip link set dev ni0 down
# ip addr flush dev ni0
# ip addr add 6.6.6.1/8 dev ni0
# ip link set dev ni0 up
```

After executing the commands for configuring Linux network interface, in AIOP console will be displayed a message for the NI 0 (DPNI 2) link up due to a DPNI link event:

```
CONTROL_FLOW : ni0 link is UP
```

It is also required to create a static entry in the ARP table, so that Linux responds directly to AIOP:

```
# arp -s 6.6.6.10 <MAC_address_of_traffic_generator>
# arp -n
Address     HWtype     HWaddress                              Flags Mask     Iface
6.6.6.10    ether      <MAC_address_of_traffic_generator>     CM             ni4
```

During frame processing, the AIOP Logger will print the following brief information about every received packet on all interfaces:

- Received MAC source and destination addresses
- Received IP source and destination addresses
- Traffic generator IP address is 6.6.6.10
- For ARP Frames AIOP Logger will print: operation code (OPCODE), Sender and Target Protocol Address.
- For ICMP Frames AIOP Logger will print: ICMP Type and ICMP Code.

```
RX on NI 0
   MAC_SA: 00-00-00-00-00-02 MAC_DA: 02-00-c0-a8-48-01
   IP_SRC: 6.6.6.10 IP_DST: 6.6.6.1
   ICMP_TYPE: 8 ICMP_CODE: 0
```

```
RX on NI 0
   MAC_SA: 00-00-00-00-00-02 MAC_DA: 02-00-c0-a8-48-01
   ARP_OPCODE: 1 S_ADDR: 6.6.6.10 T_ADDR: 6.6.6.1

RX on NI 0
   MAC_SA: 00-00-00-00-00-06 MAC_DA: 00-10-94-00-00-10
   IP_SRC: 192.0.0.1 IP_DST: 192.85.1.2

. . .
```

Assign an IP address to the Linux interface for SSH connections to RDB-1 board, for example using the following command:

```
# ip addr add 192.168.1.20/24 dev eth0
# ip link set dev eth0 up
```

Alternatively, if the DHCP server is active on the network, run following command to get an IP address automatically, for example using the following command:

```
# udhcpc -i eth0
```

6.14.1.4.3 Generating traffic to test AIOP Control Flow application

An external traffic source is needed to pass traffic to the control flow application. This could be another RDB or some other traffic generator. The following graphic illustrates traffic flow between Packet Generator and RDB board.



Figure 36. Generating traffic for AIOP Control Flow

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1.

Generate IPv4 frames for each of the two ports:

- MAC source: 00:00:00:00:00:02 (MAC address of the traffic generator)
- MAC destination: 02:00:C0:A8:48:00 (MAC of ni0 in Linux)
- For ARP and ICMP, the values that are sent to Linux GPP are:

IP Source Address	IP Destination Address	Protocol Type	ICMP Type	ICMP Code
6.6.6.10	6.6.6.1	01 (ICMP)	08 (Echo Request)	00

For other ICMP fields use any value:

Protocol Type	Sender Hardware Address	Sender Protocol Address	Target Hardware Address	Target Protocol Address
00 01 (ARP Request)	<<MAC Address of Traffic Generator>>	6.6.6.10	00:00:00:00:00:00	6.6.6.1

- For TCP or UDP use any data in IP source/destination and header specific.

Traffic can be injected in board only after the network interface connected to trafficgenerator (external traffic) is up due to internal event in AIOP:

```
CONTROL_FLOW : ni0 link is UP
```

Open two AIOP consoles (using SSH). On one of them show the AIOP Logger using command described below. When injecting traffic, on RDB-1, the AIOP Logger will print a brief information about every frame that is being processed.

```
$ busybox tail -f /dev/fsl_aiop_console
```

On the other console start packet capture, using the following command, to sniff communication between AIOP and GPP:

```
$ tcpdump -i ni1 -w aiop_control_flow.pcap
```

6.14.1.5 AIOP Header Manipulation application

6.14.1.5.1 AIOP Header Manipulation overview

The purpose of this application is to demonstrate how to perform a simple GRE tunneling using header manipulation operations available in AIOP. Application offers support only for IPv4 packets and requires one DPNI interface, acting as tunnel interface. The packet header content is used to determine the type of operation to be performed on the packet: encapsulation or decapsulation



Figure 37. AIOP Header Manipulation overview

The application performs the following operations:

- For regular IPv4 packets without inner IP header: creates a basic IP header, computes outer IP checksum, creates a basic GRE header (all flags 0, GRE version 0, encapsulated protocol: IPv4), inserts these headers between ETH and IP headers and switches the source and destination MAC addresses

- For packets having an inner IP and GRE headers: deletes the outer IP and GRE headers, switches the source and destination MAC and IP addresses
- Discards non-IPv4 and non-GRE packets
- Re-runs parser and prints the packet after header manipulation occurred. This is optional and can be deactivated by defining the macro RERUN_PARSER
- Sends back packets on same network interface they were received from.

6.14.1.5.2 Running the Header Manipulation application

The ELF for AIOP Classifier is `/usr/aiop/bin/aiop_header_manip.elf`. This can be loaded via AIOP tool as described in Creating AIOP containers on page 515. For this application the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP Classifier Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_header_manip.elf
```

To check if the AIOP Classifier Application loaded successfully, execute the following command in the Linux command shell:

```
# cat /dev/fsl_aiop_console | grep HEADER_MANIP
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g ni0) together with the associated MAC address.

```
HEADER_MANIP : Successfully configured ni0 (dpni.6)
HEADER_MANIP : dpni.6 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

NOTE

Although the AIOP container contains multiple NIs the AIOP Reflector will use only the NIs that have a DPMAC as endpoint (on which it can succesfully configure the order scope).

During packet processing, the AIOP Logger prints the following brief information about every received packet on all interfaces:

- Received IP source and destination addresses (for the inner header and for the outer IP header in case packet is encapsulated)
- GRE Flags, version and encapsulated protocol, in case the packet is encapsulated.

If parser re-running was enabled the information above is printed, in order to show the result of encapsulation/decapsulation header manipulation operations performed.

```
# busybox tail -f /dev/fsl_aiop_console

Decapsulated Frame | FD Len 124 | SEG Len 124
  OUTER IP HEADER
    PROTO: 1 IP_SRC: 192.168.1.10, IP_DST: 10.171.77.121, TOTAL LEN: 110, IHL: 20

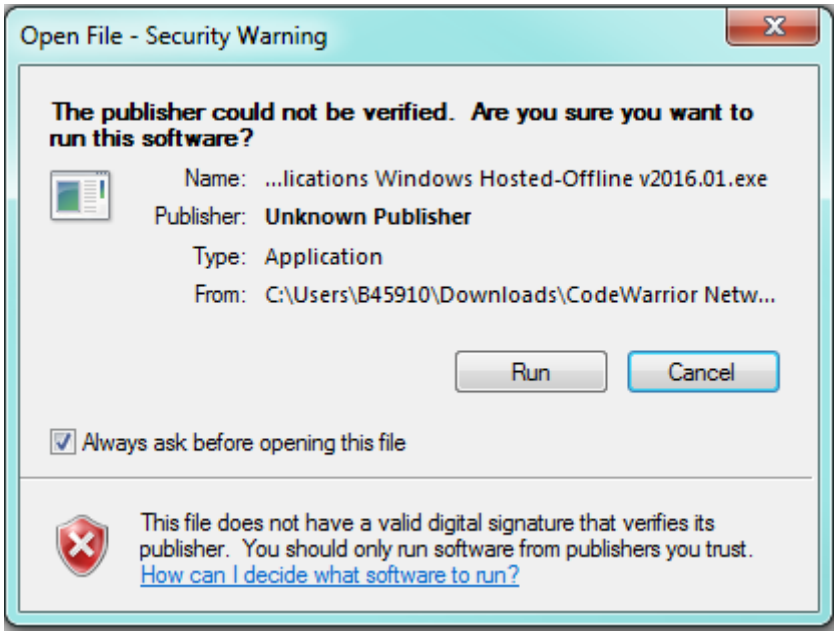
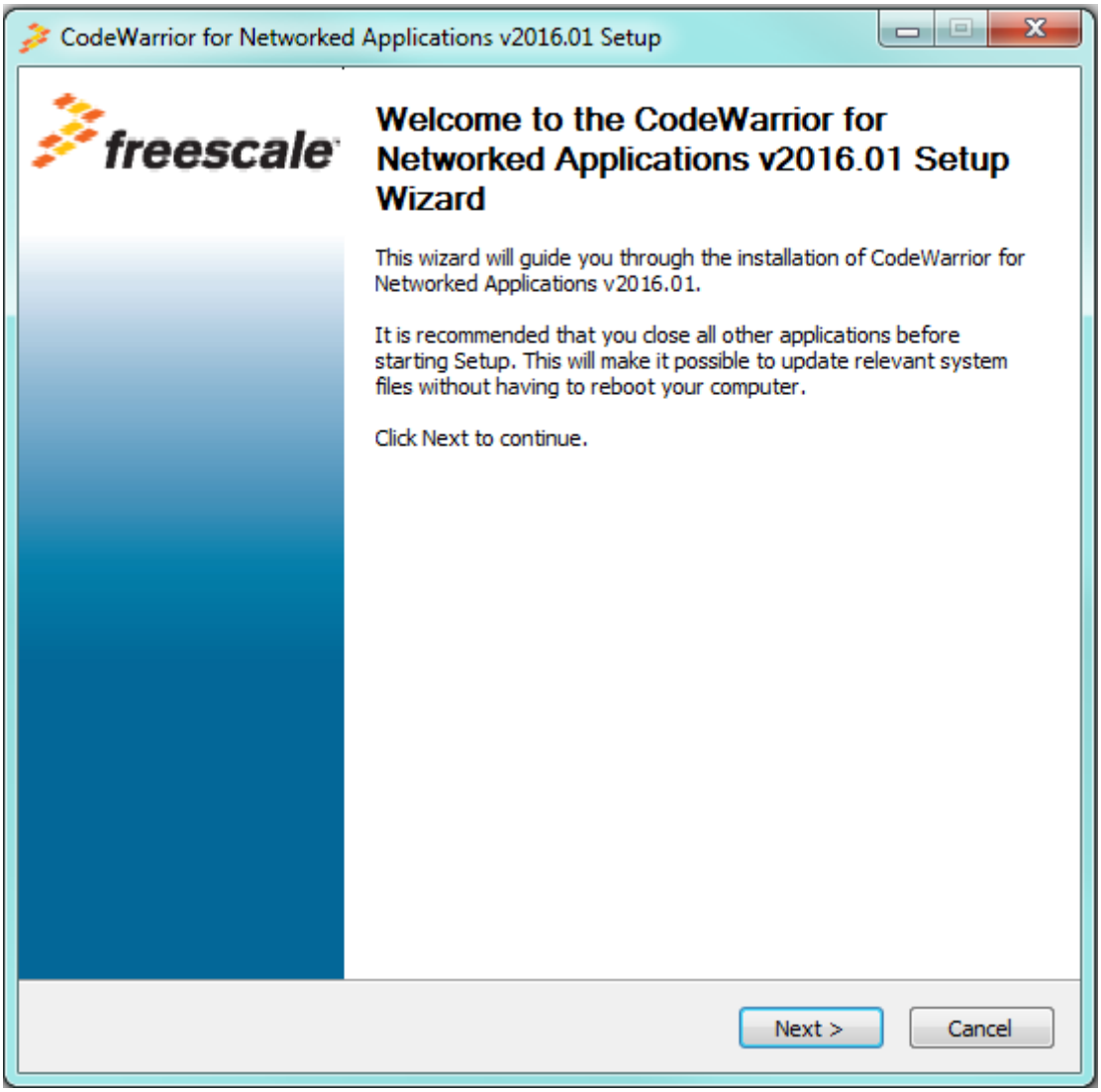
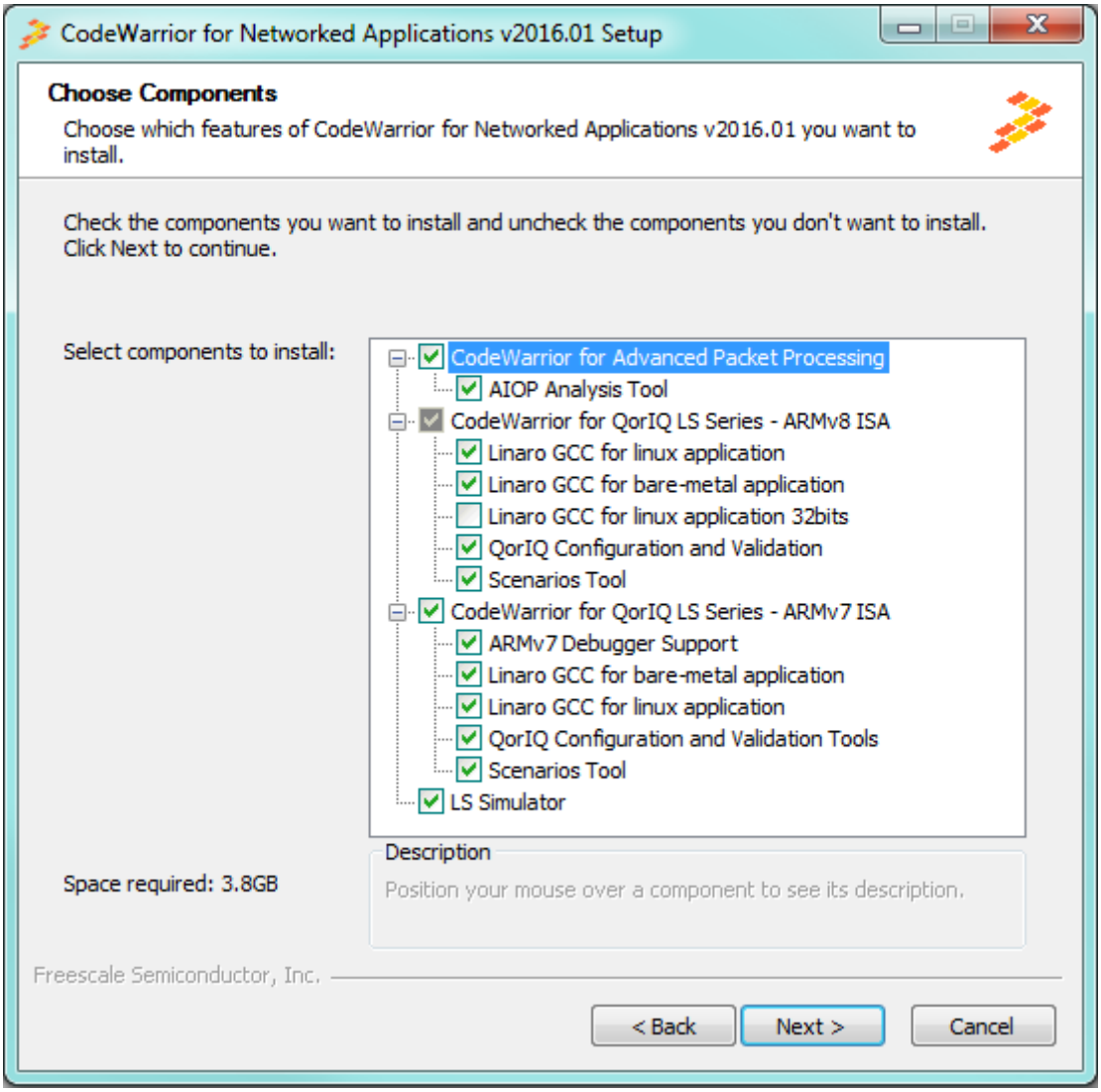
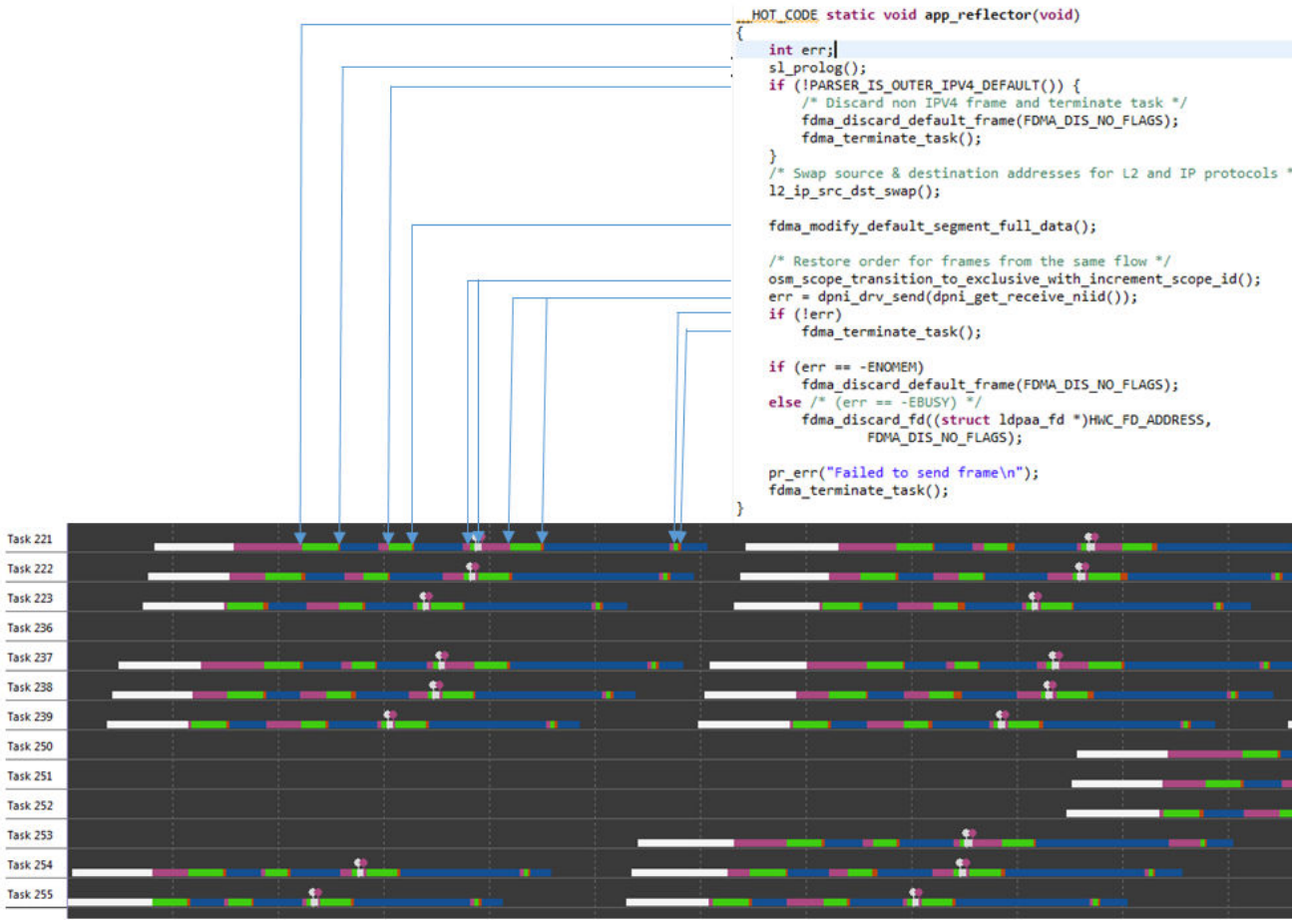
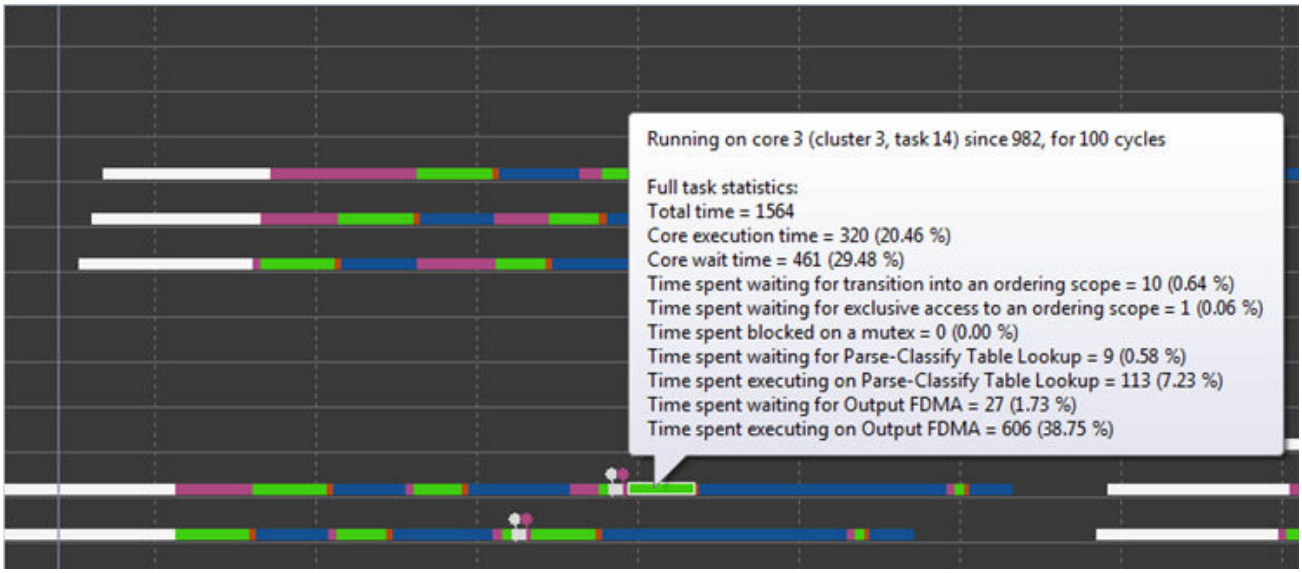
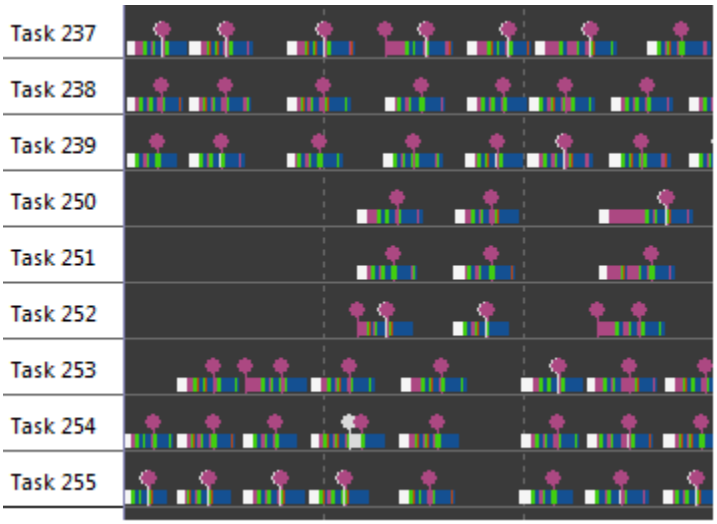
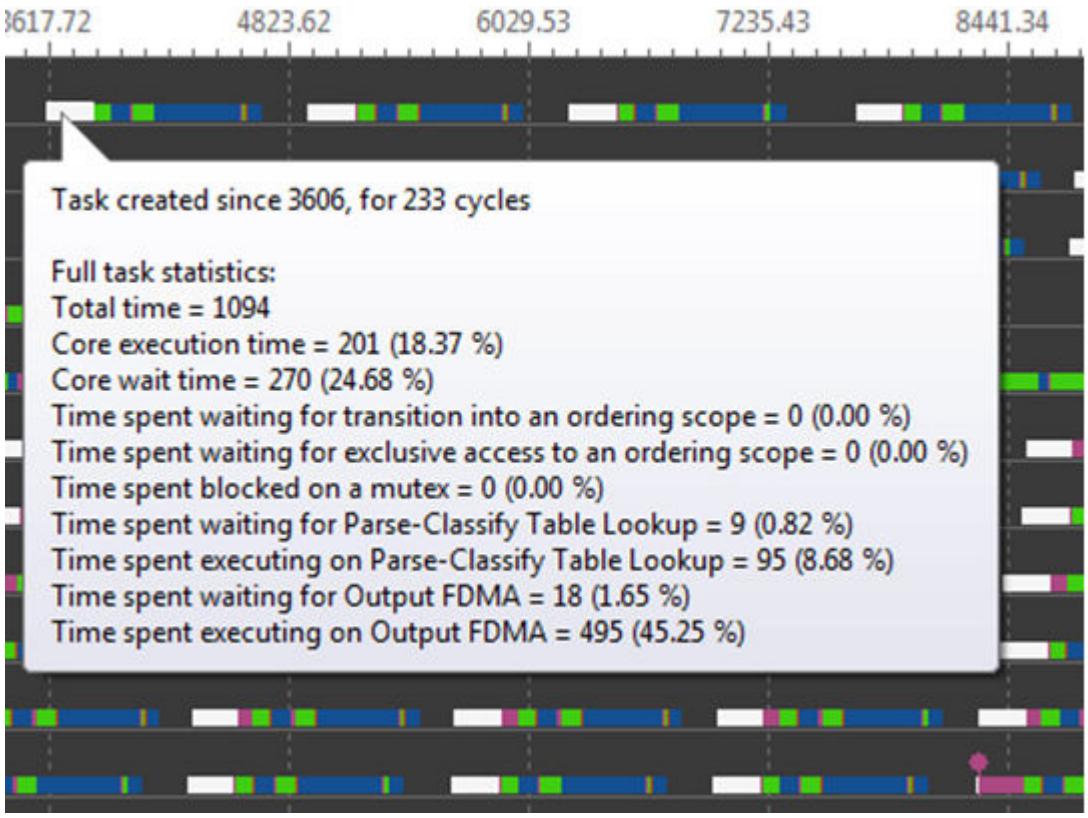
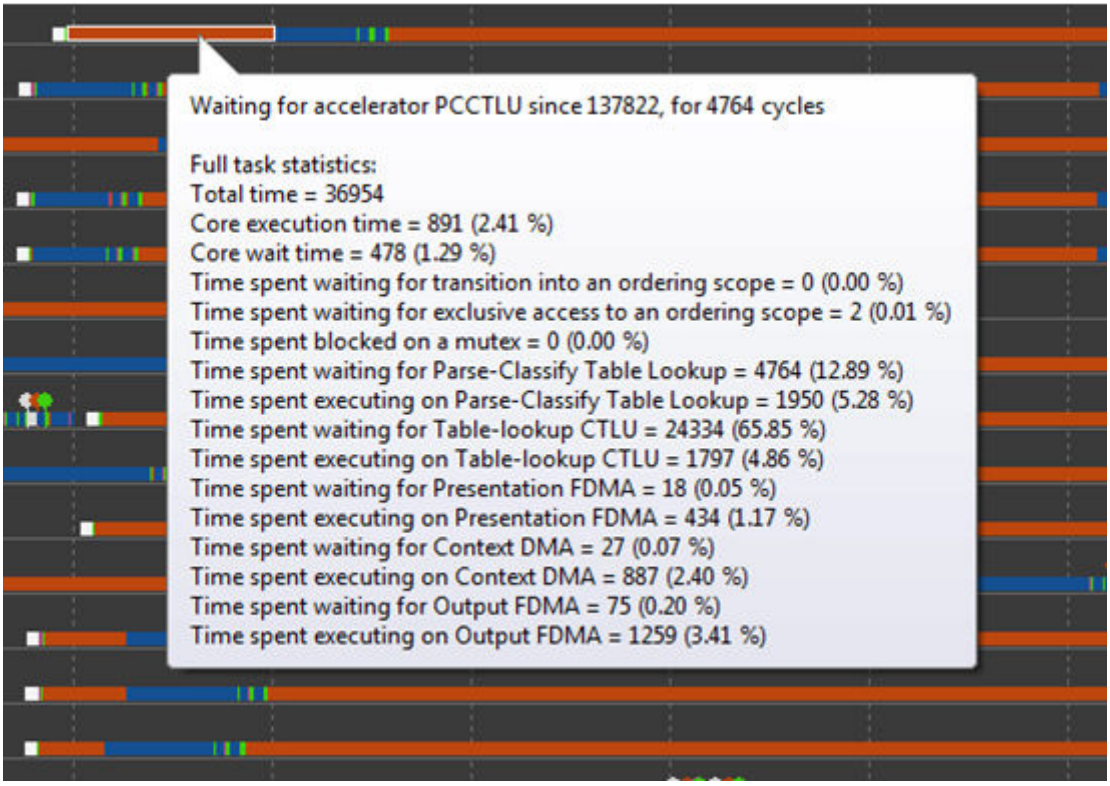
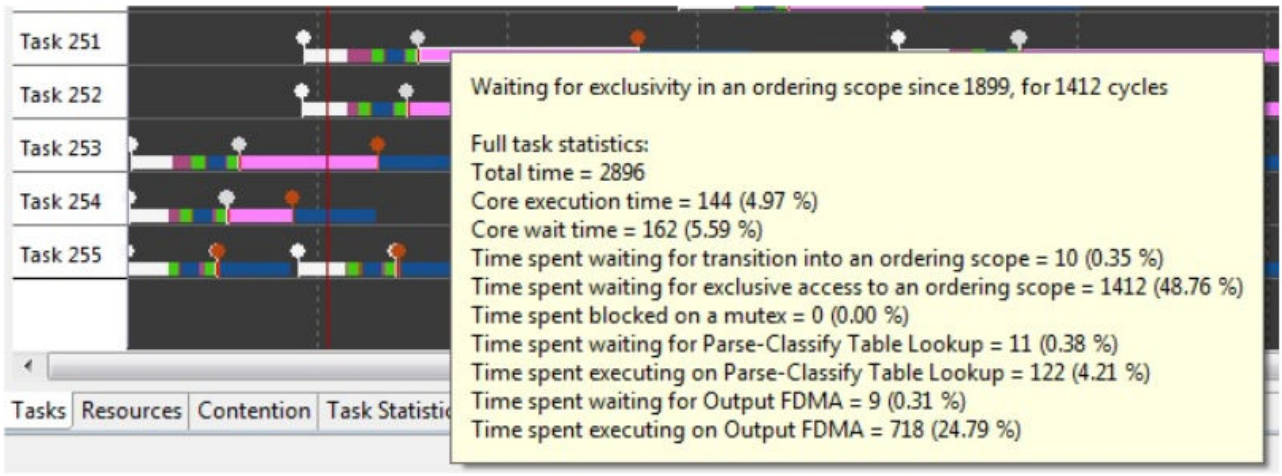
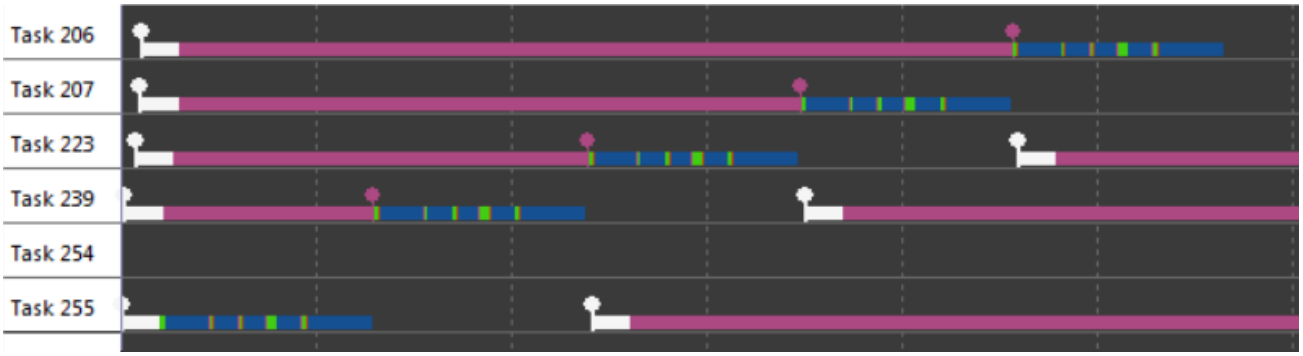
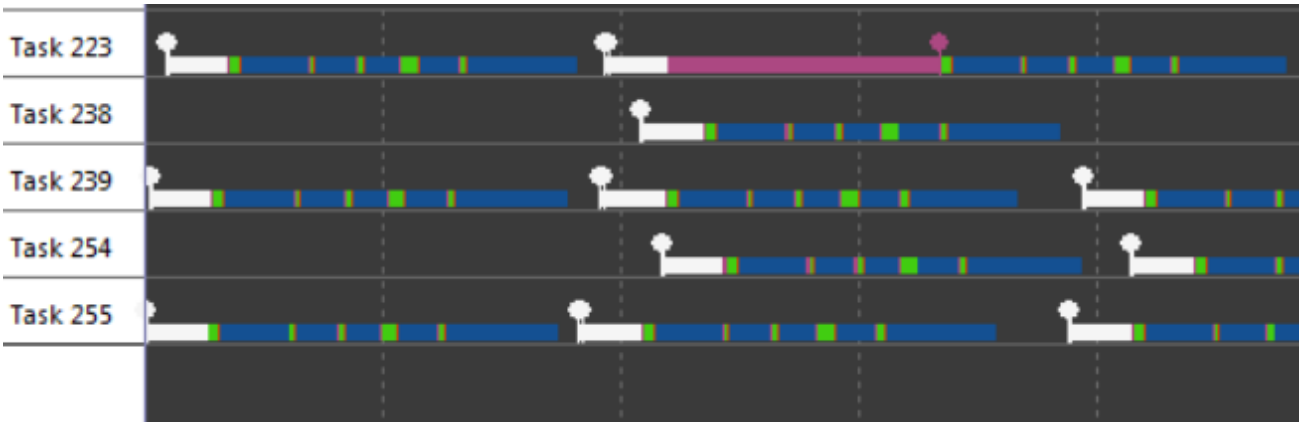
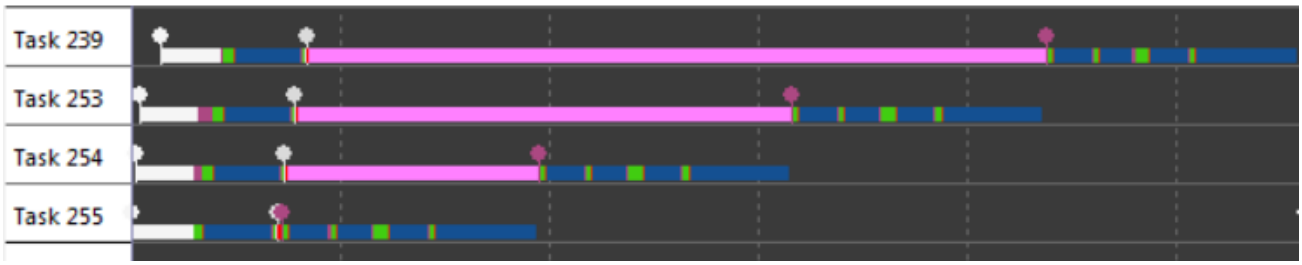
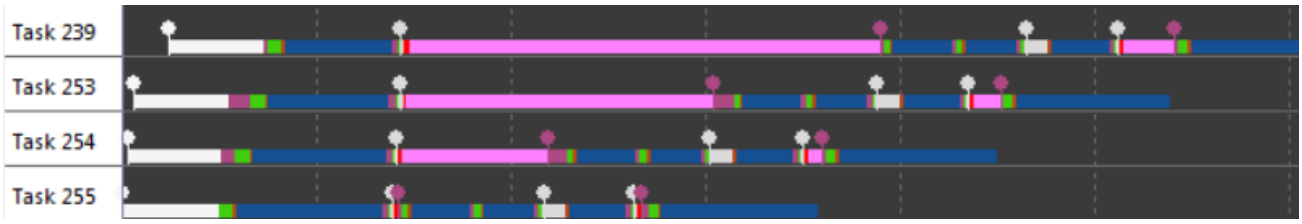
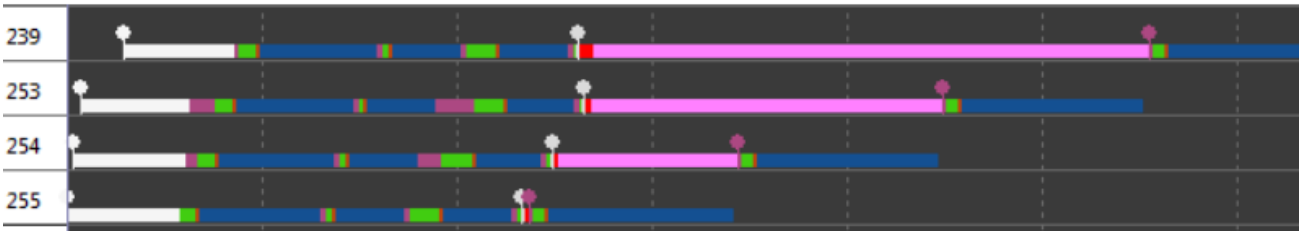
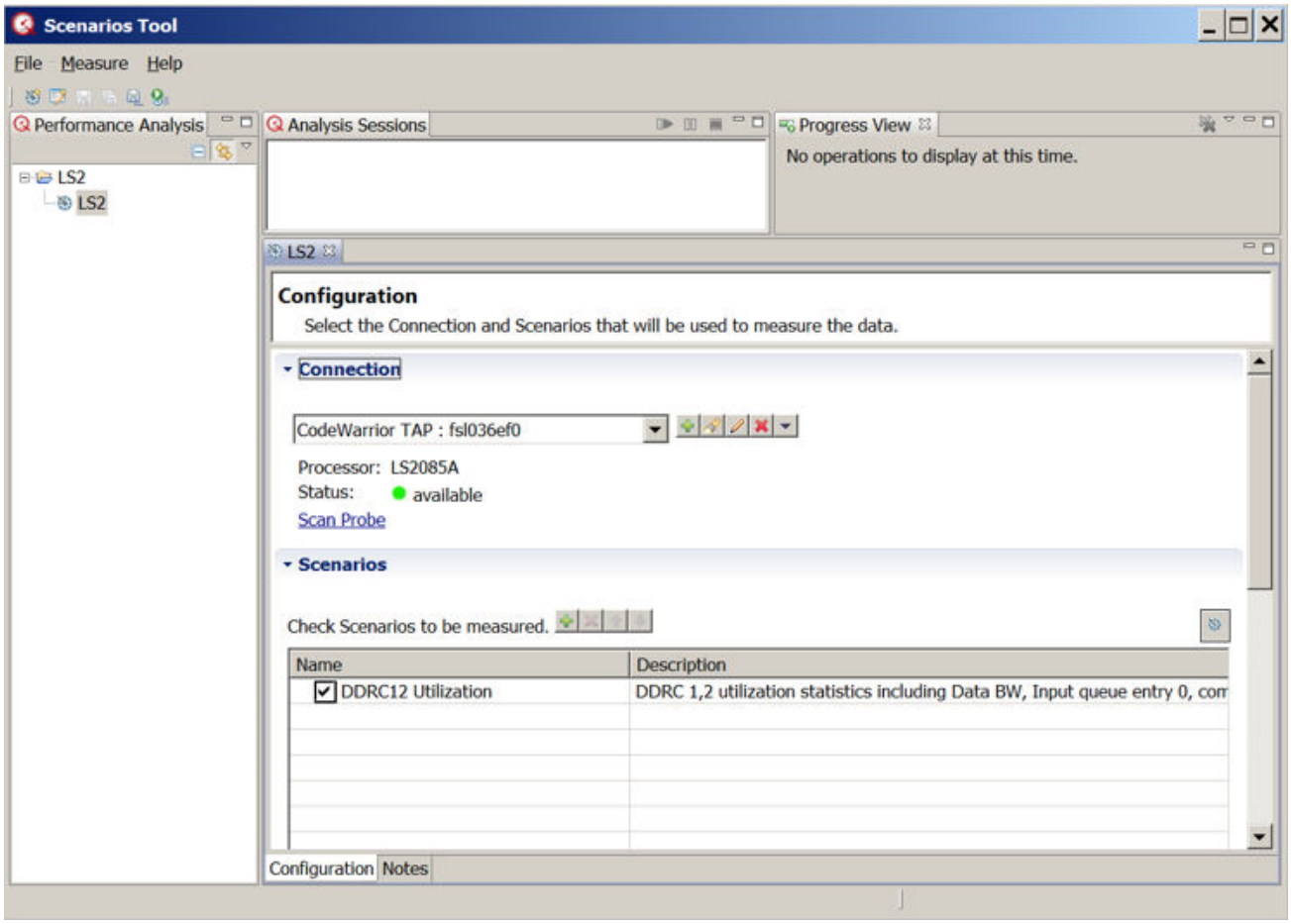
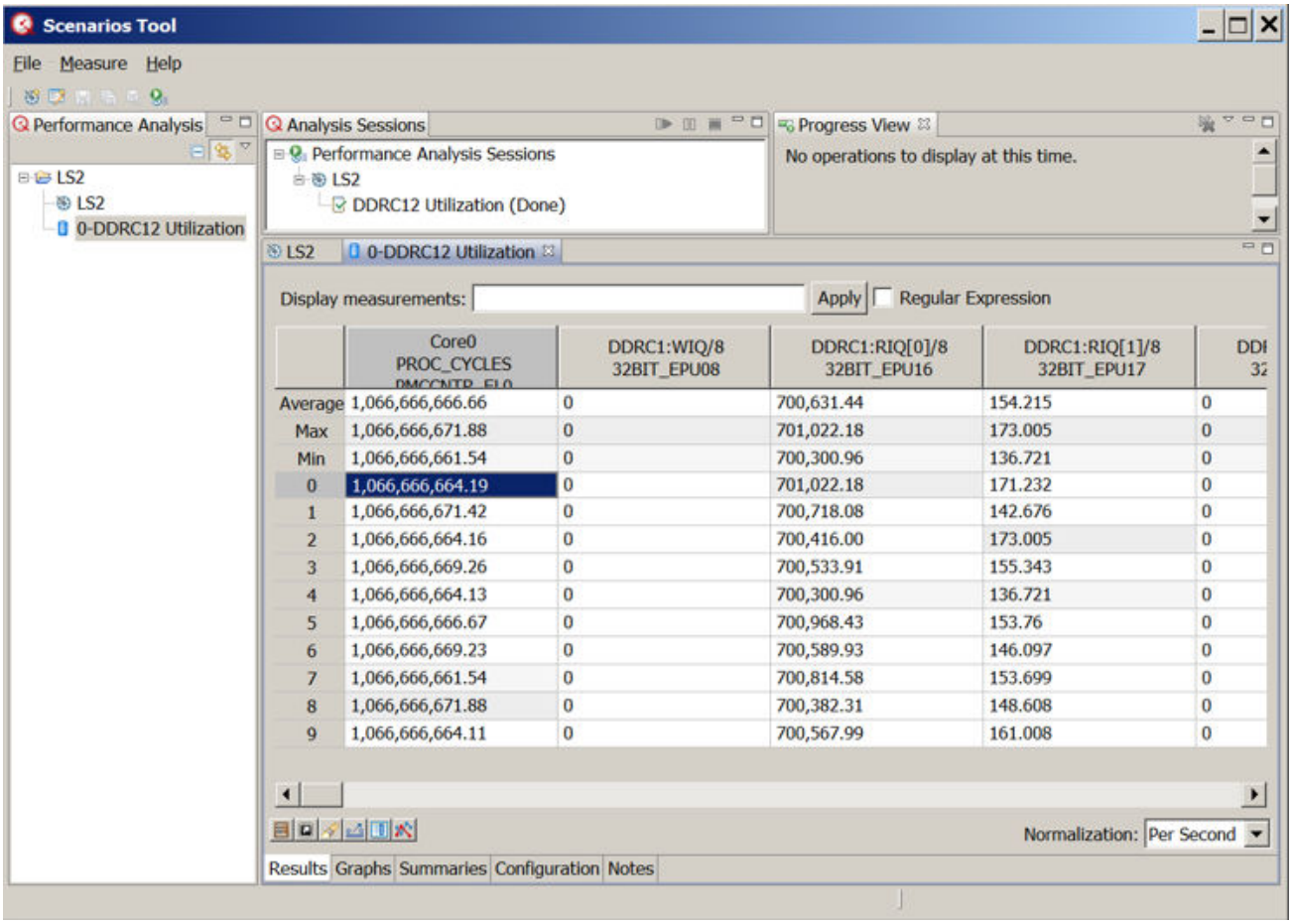
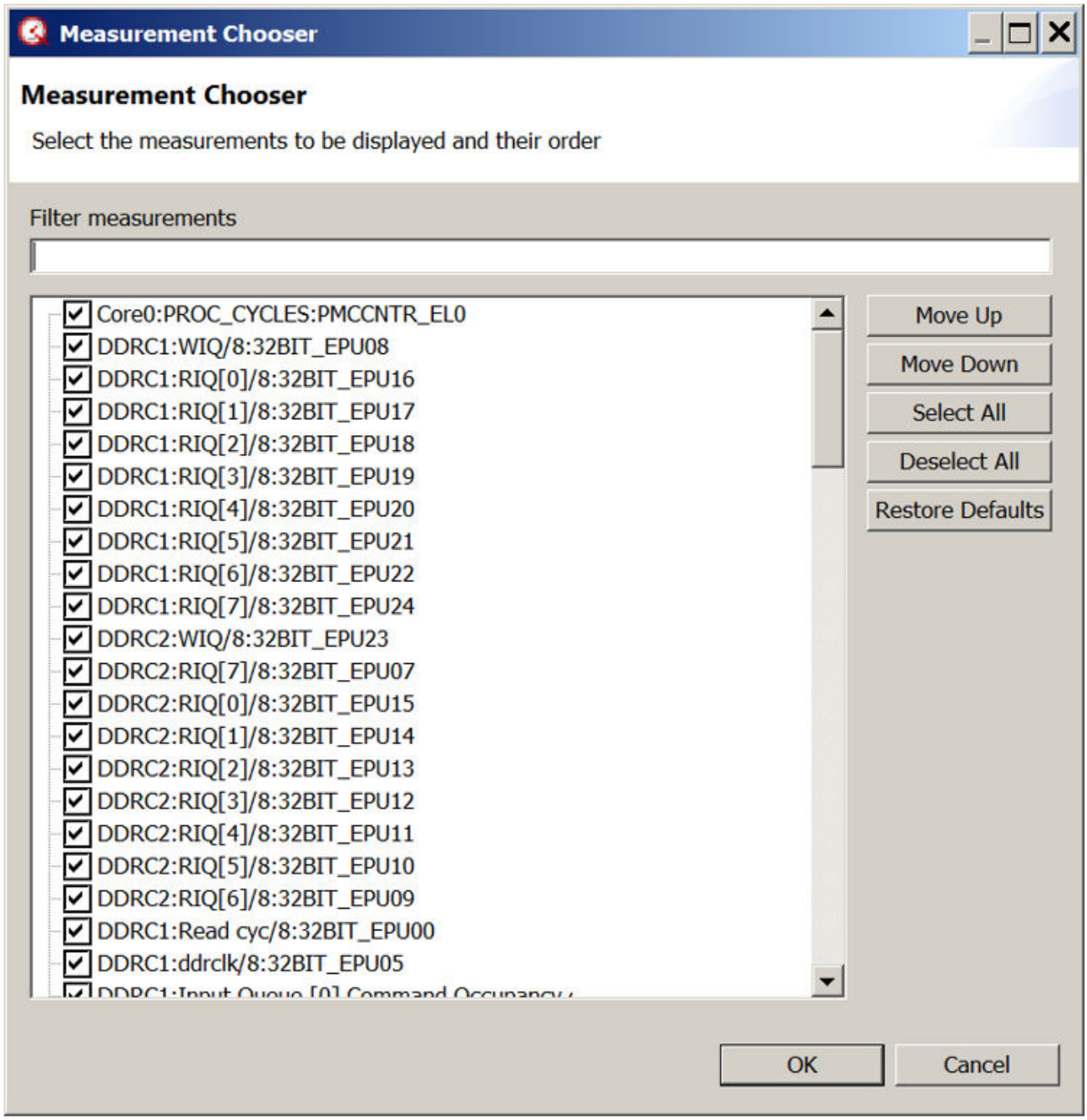
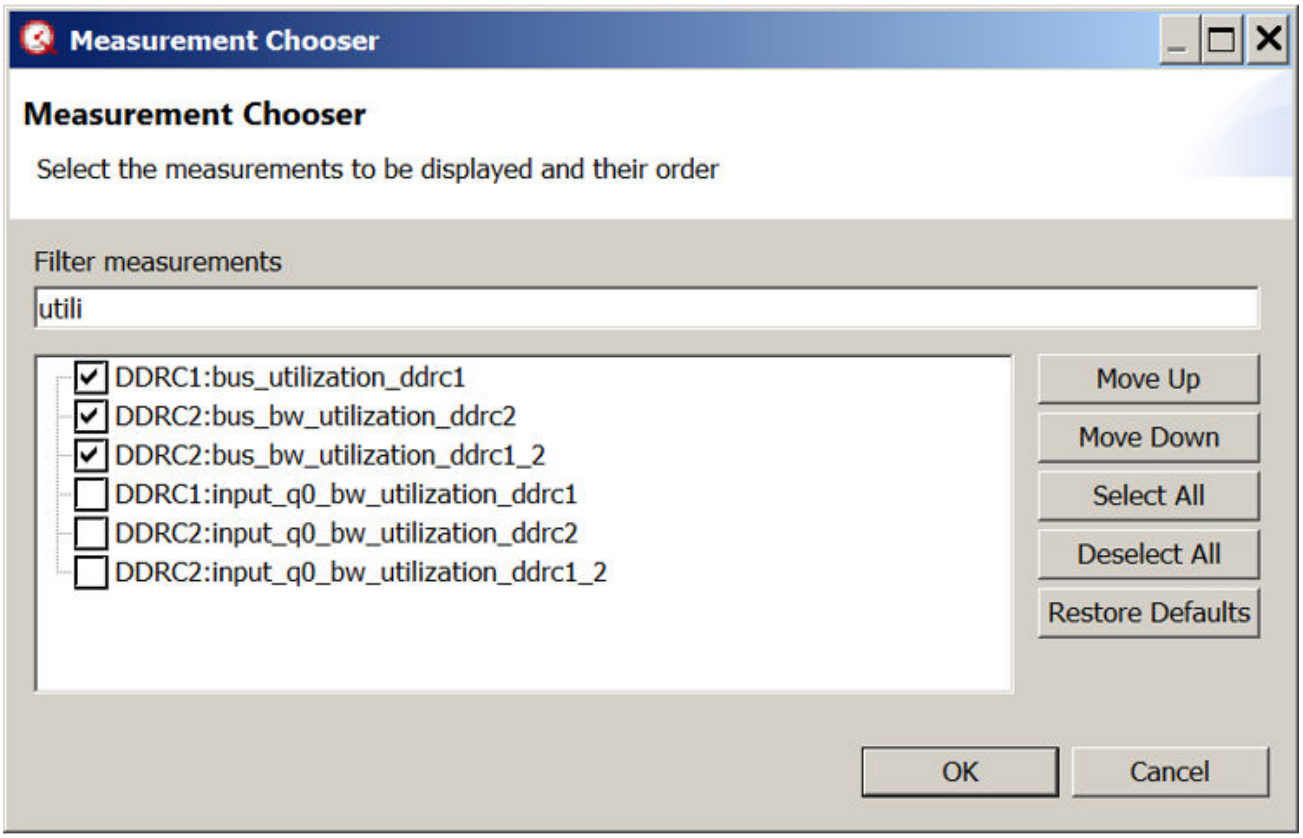
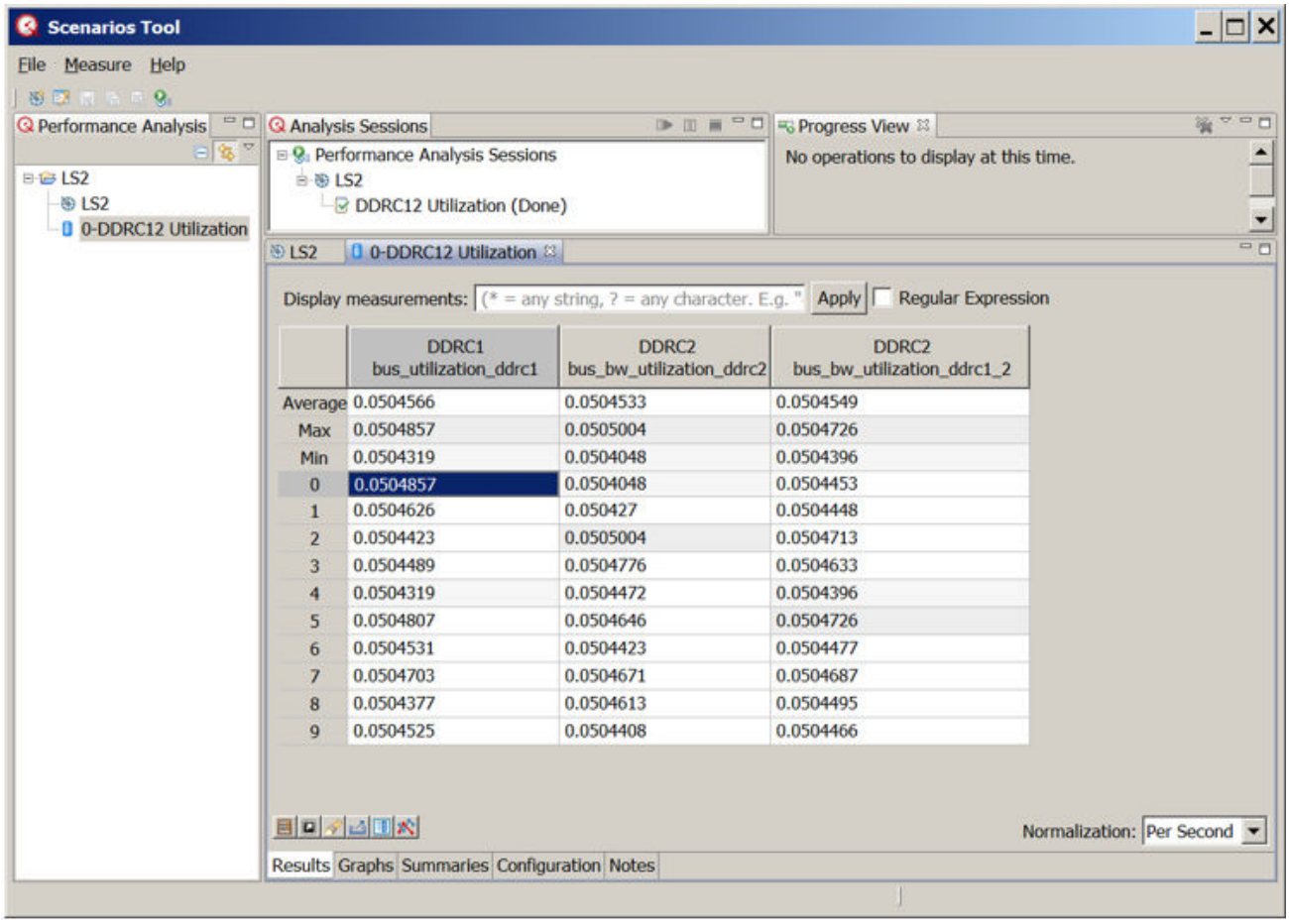
Encapsulated Frame | FD Len 148 | SEG Len 124
  OUTER IP HEADER
    PROTO: 47 IP_SRC: 122.122.122.122, IP_DST: 138.138.138.138, TOTAL LEN: 134, IHL: 20
  Generic Routing Encapsulation
    FLAGS: 0x00000, VERSION: 0x000, PTYPTE: 0x0800
  INNER IP HEADER
    PROTO: 1 IP_SRC: 192.168.1.10, IP_DST: 10.171.77.121, TOTAL LEN: 110, IHL: 20
```

```
Encapsulated Frame | FD Len 148 | SEG Len 128
  OUTER IP HEADER
    PROTO: 47 IP_SRC: 10.8.8.8, IP_DST: 10.7.7.7, TOTAL LEN: 134, IHL: 20
  Generic Routing Encapsulation
    FLAGS: 0x00000, VERSION: 0x000, PTYPTE: 0x0800
  INNER IP HEADER
    PROTO: 6 IP_SRC: 198.168.2.21, IP_DST: 10.18.1.1, TOTAL LEN: 106, IHL: 20

Decapsulated Frame | FD Len 120 | SEG Len 120
  OUTER IP HEADER
    PROTO: 6 IP_SRC: 10.18.1.1, IP_DST: 198.168.2.21, TOTAL LEN: 106, IHL: 20
```

6.14.1.5.3 Generating traffic to test AIOP Header Manipulation application

An external traffic source is needed to pass traffic to the classifier application. This could be another RDB or some other traffic generator. One optical 10G ports need to be connected to MAC1 or MAC2 of RDB-1.

The following diagram illustrates how traffic moves from a packet generator to RDB-1 board.



Figure 38. Generating traffic for AIOP Header Manipulation

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1.

On RDB-1 ensure that both AIOP network interface is in link up state:

```
$ cat /dev/fsl_aiop_console | grep HEADER_MANIP
HEADER_MANIP: NI 0 link is UP
```

Generate IPv4 packets:

- Common values:
  - MAC destination: 00:00:00:00:00:06 (address of MAC1)
  - MAC source: ANY
  - Outer IP source: ANY
  - Outer IP destination: ANY
- For GRE encapsulated packets:
  - Version: 0
  - GRE Flags and version: 0x0000
  - GRE Encapsulated Protocol: 0x0800 (IP)
  - Inner IP source: ANY
  - Inner IP destination: ANY

On RDB-1, the AIOP Logger prints brief information about every packet that is being encapsulated/decapsulated and if parser re-running is enabled the AIOP Logger prints brief information about the packet after the GRE encapsulation/decapsulation header manipulations occurred.

The encapsulated packets using GRE by AIOP application have the following pre-defined values:

- Outer IP Version: 4
- Outer IP source: 122.122.122.122
- Outer IP destination: 138.138.138.138
- GRE Version: 0
- GRE Flags and version: 0x0000
- GRE Protocol Type: 0x0800 (IP)
- Inner IP source: IP source from original packet
- Inner IP destination: IP destination from original packet

The packets that were decapsulated by AIOP will have the inner IP source and destination addresses and MAC addresses swapped.

6.14.1.6 AIOP Statistics application

6.14.1.6.1 AIOP Statistics overview

The purpose of this application is to demonstrate the usage of the AIOP atomic operations in order to update software definedstatistics counters. There are two types of statistics counters:

- Global: total number of the received packets, total number of the received bytes , total number of the accepted packets, total number of the rejected packets
- Per-flow: total number of the recived packets and total number of the received bytes, for each packet matching the applied classification criteria



Figure 39. AIOP Statistics overview

The application performs the following operations:

- Classifies the packets to enable different counters update. Classification is based only on the Protocol Number field. Only the TCP, UDP and SCTP protocols are accepted. The packets not matching the classification criteria are dropped and number of dropped packets is increased. [GI1]
- Increases the global statistics and the statistics for each packet matching the classification criteria (per-flow statistics).
- Swap the MAC and IP addresses
- Sends back packets on same network interface they were received from.

- By default, the global and per flow statistics are displayed at every 60 seconds. In order achieve this, a timer is used. The timer time interval, measured in seconds, can be changed by setting the value of the macro definition APP_TMAN_TIMER_DURATION, at compile time.

Note: Both global and per-flow statistics are stored in the DP-DDR (Data Path DDR) memory partition. For the global statistics, a contiguous block of memory is obtained and for each flow a buffer from DP-DDR is obtained by using pool-based allocation.

6.14.1.6.2 Running the AIOP Statistics application

The statistics application should be run on the RDB-1 system.

The ELF for AIOP reflector is `/usr/aiop/bin/aiop_statistics.elf`. This can be loaded via AIOP tool as section Creating AIOP Containers describes. For this application the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP Statistics Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_statistics.elf
```

To check if the AIOP Statistics Application loaded successfully, execute the following command in the Linux command shell:

```
# cat /dev/fsl_aiop_console | grep STATISTICS
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g ni0) together with the associated MAC address.

```
STATISTICS : Created TMI id=0x2
STATISTICS : Created timer for showing statistics, handle=0x102
STATISTICS : Application initialized successfully
STATISTICS : Successfully configured ni0 (dpni.2)
STATISTICS : dpni.2 <---connected---> dpmac.2 (MAC addr: 00:00:00:00:00:07)
STATISTICS : Successfully configured ni1 (dpni.1)
STATISTICS : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
STATISTICS : ni1 link is UP
```

Once the application started, at every 60 seconds (or at the time interval defined by APP_TMAN_TIMER_DURATION macro) the AIOP Logger prints all statistics counters values. Default value should be 0 for all counters.

```
AIOP received 47753801 packets (5921471324 bytes)
        12585359 dropped packets, 35168442 accepted packets, 35168442 transmitted packets
         * PROTO=0x6: received: 12601479 packets (1562583396 bytes)
         * PROTO=0x11: received: 12608080 packets (1563401920 bytes)
         * PROTO=0x84: received: 9958883 packets (1234901492 bytes)
```

6.14.1.6.3 Generating traffic to test AIOP Statistics application

An external traffic source is needed to inject packets into the application. This could be another RDB-1 or some other traffic generator. One optical 10G port needs to be connected to MAC1 or MAC2 port of the RDB

The following diagram illustrates how traffic moves from a packet generator to RDB-1 board.



Figure 40. Generating traffic for AIOP Statistics

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1 and a second port to MAC2 of RDB-1.

On RDB-1 ensure that both AIOP network interfaces are in link up state:

```
$ cat /dev/fsl_aiop_console | grep STATISTICS
STATISTICS : NI 0 link is UP
```

Generate IPv4 frames for each of the two ports:

- MAC source: ANY
- MAC destination: 00:00:00:00:00:06 (address of MAC1)
- For IP & Layer 4, the accepted values are show in the following table:

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
ANY	ANY	6 (TCP)	ANY	ANY
ANY	ANY	17 (UDP)	ANY	ANY
ANY	ANY	132 (SCTP)	ANY	ANY

On RDB-1, the AIOP Logger prints at every 60 seconds (or at the time interval configured with the APP_TMAN_TIMER_DURATION macro) the global statistics and the per-flow statistics for each matched protocol (TCP, UDP and SCTP).

6.14.2 AIOP Tool User's Guide

6.14.2.1 Introduction

This document contains information about the usage of AIOP Tool application, including compilation and execution steps. This document is a part of the series intended to help developers use DPAA2 software on NXP's LS family of network processors. This document is intended to get users up and running quickly.

6.14.2.1.1 Conventions

This document uses the following conventions:

`Courier`	`Is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.`

6.14.2.1.2 Definitions and acronyms

AIOP	Advanced I/O Processor
DPAA	Data Path Acceleration Architecture
DPRC	Data Path Resouce Container
GPP	General Purpose Processor (or, in context of Linux applications, user-space)
MC	Management Complex
ODP	Open Data Plane Software

6.14.2.2 DPAA2 Software

For information about the DPAA2 software, see the DPAA2 Software Overview on page 410 section. For users that are unfamiliar with DPAA2 software, it is recommended to read that content before proceeding.

6.14.2.3 Product Description

6.14.2.3.1 Overview

What is AIOP?

AIOP or Advanced I/O Processor is an optional component of the DPAA2 architecture. It is a C-programmable engine that is optimized for packet processing and can be used on a SoC in conjunction with the general purpose cores and the other elements of the DPAA2 architecture. AIOP executes a software image containing the packet processing logic.

What is AIOP Tool?

AIOP Tool is a Linux user space application which performs the following operations on an AIOP:

1. Loading an image on an AIOP
2. Starting (running) the image on an AIOP after it has been loaded
3. Extracting state information for an AIOP
4. Getting and setting time of day on an AIOP

6.14.2.3.2 Product features

AIOP Tool provides following broad functions:

- Provides a command line interface for executing a set of operations
  - Command line will accept an operation from user, along with the appropriate corresponding arguments.
  - The DPRC to work on is also accepted as an argument on the command line.
- Operation for Loading AIOP Image to an AIOP
  - Command line will accept an AIOP image file.
  - Whether a reset of the AIOP should be done before loading or not can be controlled using a toggle argument passed on command line. For performing reset of AIOP before load operation, an optional toggle argument can be passed to the load operation.
    - Current hardware revision (rev1) doesn't support reset and it is expected to work in subsequent releases. Thus, in future, the same tool can be used without any modifications.
  - Result of load API call will be provided to the caller.

- Operation for Resetting the AIOP
  - On execution, this would put the AIOP in Reset state for loading another image.
    - In the current hardware revision (rev1), this is not supported and would result in error. In subsequent versions of LS family hardware, this is expected to be operational. Once that is done, with the combination of load and run operations, multiple images can be loaded on AIOP (serially) without a hardware reset.
  - Response value of API execution will be provided to caller.
- Operation for getting and setting time of day on AIOP
  - Once an AIOP is running, 2 operations for getting and setting the time of day on AIOP are available.
  - Success or failure of the API calls would be provided to caller in case of time setting. In case of time getting operation, time since epoch would be shown.
- Operation for printing AIOP status
  - On execution of this operation, status of the AIOP is provided to the caller. This includes the current state and version information.
- Help menu will be printed by command line for displaying usage of above operations and their corresponding arguments.

6.14.2.4 System Requirements

This chapter describes the environment requirements for executing the AIOP Tool.

6.14.2.4.1 Environment required

1. MC Firmware version 9.0.x for LS2085A (RDB and QDS) EAR-6.0
2. Uboot and Linux compatible with LS2085A (RDB and QDS) EAR-6.0
3. AIOP image (ELF) is required as input to the tool

6.14.2.5 AIOP Tool Usage

6.14.2.5.1 Run time pre-requisites

Besides the environment pre-requisites listed in Environment required on page 534, the following execution time pre-requisites exist:

1. AIOP Tool requires a DPRC which has an AIOP included in it. It should be a non-root DPRC containing at least a dpmcp and dpaiop object.
   - This DPRC can be statically defined (static DPL) or dynamically created (using `restool`).
   - Please refer Steps For Dynamic DPRC Suitable For AIOP Tool Using restool on page 540 below for steps for dynamically creating a suitable DPRC using `restool`.
   - Also, refer `dynamic_AIOP_dpl.sh` script delivered as part of the EAR 6.0 ODP release. This is a script version of dynamic DPRC creation steps mentioned in Steps For Dynamic DPRC Suitable For AIOP Tool Using restool on page 540.
2. VFIO support in Linux kernel
   - AIOP is a hardware device which AIOP Tool attempts to access from Linux userspace. For exchanging information between AIOP Tool and AIOP, a secure memory area is required which can be accessed by both. Linux VFIO or Virtual Function I/O, provides a framework for securely exposing certain memory area which is equally accessible by AIOP Tool and AIOP hardware. VFIO would map the addressing understood by AIOP hardware to that of addressing managed by user-space application.

- This application assumes that the `dpaiop` device would be mapped into VFIO driver so as to use DMA mapping between AIOP Tool and AIOP hardware and for pushing SMMU addresses for AIOP hardware's access to user-space memory
- Refer Sample VFIO Binding Script on page 540 for a sample script to perform binding of a DP object with VFIO.

3. Valid AIOP image for loading
   - For sample, refer to `cmdif_integ_dbg.elf` image provided along with '`aiopsl`' repo. This can be verified using GPP application `odp_cmdif_demo` in ODP.

6.14.2.5.2 Environment setting

For executing the AIOP Tool, a valid DPRC is required. There are three ways to define a DPRC:

1. Provide DPRC through '`-g`' command line tool. For e.g., `aiop_tool <sub-command> -g dprc.4`.
2. If not provided through command line, the binary expects the environment variable `DPRC` defined.
3. If neither an argument nor environment variable is provided, binary assumes a default value of DPRC as `dprc.5`. (Current hard-coded in the code)

6.14.2.5.3 Command line arguments

All variants of the execution follow a standard pattern:

```
aiop_tool <sub-command> <Arguments>
```

`<Sub-command>` includes `help`, `load`, `gettod`, `settod`, `status` and `reset`.

`<Arguments>` are either mandatory (for e.g., image file path), or optional (for e.g., DPRC name).

All variants assume that the VFIO binding has already been performed on the command line for the DPRC on which command would be executed. Further, following samples/snippets assume that DPRC would be passed on command line, though all three methods mentioned in Environment setting on page 535 are valid.

Following table describes all the variants of the AIOP Tool execution.

Argument/command pattern	Description
`aiop_tool` **`help`**	Displays help including various sub-commands and their output.

Table continues on the next page...

Table continued from the previous page...

<table>
<tr><th>Argument/command pattern</th><th>Description</th></tr>
<tr><td><code>aiop_tool load</code></td><td>Loading a valid ELF image onto the AIOP and starting it.
Valid arguments are:
<table>
<tr><td>-g <DPRC name></td><td>Name of the DPRC containing the AIOP object. This is a non-root DPRC which is dedicated to the AIOP Tool.</td></tr>
<tr><td>-f <aiop image file></td><td>[Mandatory] Name, with path, of the AIOP image file to be loaded on the AIOP</td></tr>
<tr><td>-r</td><td>[Optional] Reset toggle; If provided, AIOP Tool will perform reset of AIOP before attempting load. By default, this would not be done.</td></tr>
<tr><td>-v</td><td>Verbose Output (More informative with INFO level messages)</td></tr>
<tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr>
</table>Following is the expected output of the command:
• In case of incorrect parameters (incorrect DPRC, wrong file path, or incorrect parameter), failure would be reported along with help usage.
• Success of failure, as reported by MC’s API; the AIOP Tool would convert the error into a readable string.</td></tr>
</table>

Table continues on the next page...

Table continued from the previous page...

<table>
<tr><th>Argument/command pattern</th><th>Description</th></tr>
<tr><td>aiop_tool gettod</td><td>Obtaining the Time of day on the AIOP.
Valid arguments are:<table><tr><td>-g <DPRC name></td><td>Name of the DPRC containing the AIOP Object. This is a non-root DPRC which is dedicated to the AIOP Tool.</td></tr><tr><td>-v</td><td>Verbose Output (More informative with INFO level messages)</td></tr><tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr></table>Following is the expected output of the command:
In case of incorrect parameters (incorrect DPRC), failure would be reported along with usage help.
Time of day (64Bit value) as returned by MC’s API in case of success.<pre>Time of day: 184082</pre>
Failure, as reported by MC’s API; the AIOP Tool would convert the error into a readable string.</td></tr>
<tr><td>aiop_tool settod</td><td>Set time on the AIOP.
Valid arguments are:<table><tr><td>-g <DPRC name></td><td>Name of the DPRC containing the AIOP Object. This is a non-root DPRC which is dedicated to the AIOP Tool.</td></tr><tr><td>-t sec_since_epoch</td><td>[Mandatory] Time, in milli-seconds, since epoch.</td></tr><tr><td>-v</td><td>Verbose Output (More informative with INFO level messages)</td></tr><tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr></table>Following is the expected output of the command:
In case of incorrect parameters (incorrect DPRC, non-integer time), failure would be reported along with usage help.
Success of failure as reported by MC’s API; the AIOP Tool would convert the error into readable string.</td></tr>
</table>

Table continues on the next page...

Table continued from the previous page...

<table>
<tr><th>Argument/command pattern</th><th>Description</th></tr>
<tr><td>aiop_tool status</td><td>Valid arguments are:
<table><tr><td>-g <DPRC name></td><td>Name of the DPRC containing the AIOP Object. This is a non-root DPRC which is dedicated to the AIOP Tool.</td></tr><tr><td>-v</td><td>Verbose Output (More informative with INFO level messages)</td></tr><tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr></table>
Following is the expected output of the command:
• In case of incorrect parameters (incorrect DPRC), failure would be reported along with usage help.
• Status, including current state, SL version.
<pre>AIOP running status:
  Major Version: 2, Minor Version: 1
  Service Layer:- Major Version: 0,
Minor Version: 7, Revision: 0
  State: DPAIOP_STATE_RUNNING</pre>• Success of failure as reported by MC’s API; the AIOP Tool would convert the error into readable string.</td></tr>
<tr><td>aiop_tool reset</td><td>Reset the AIOP. This is current feature support, dependent on support by hardware.
Valid arguments are:
<table><tr><td>-g <DPRC name></td><td>Name of the DPRC containing the AIOP Object. This is a non-root DPRC which is dedicated to the AIOP Tool.</td></tr><tr><td>-v</td><td>Verbose Output (More informative with INFO level messages)</td></tr><tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr></table>
Following is the expected output of the command:
• In case of incorrect parameters (incorrect DPRC), failure would be reported along with usage help.
• Success of failure as reported by MC’s API; the AIOP Tool would convert the error into readable string.</td></tr>
</table>

6.14.2.5.4 Command execution samples

1. Obtaining status of AIOP

```
$ aiop_tool status -g dprc.4
AIOP running status:
  Major Version: 0, Minor Version: 0
  Service Layer:- Major Version: 0, Minor Version: 0, Revision: 0
  State: DPAIOP_STATE_RESET_DONE
```

   Before loading the image, for this version where hardware Reset operation is not supported, the state DPAIOP_STATE_RESET_DONE can be verified using above example.

2. Loading an ELF image on AIOP

```
$ aiop_tool load -g dprc.4 -f cmdif_integ_dbg.elf
AIOP Image (cmdif_integ_dbg.elf) loaded successfully.
```

   The `load` sub-command loads as well as runs an AIOP. Once the image has been loaded, after a few seconds, the status output would look similar to:

```
$ aiop_tool status -g dprc.4
AIOP running Status:
  Major Version: 2, Minor Version: 1
  Service Layer:- Major Version: 0, Minor Version: 7, Revision: 0
  State: DPAIOP_STATE_RUNNING
```

3. Get time of day from AIOP

```
$ aiop_tool gettod -g dprc.4
Time of day: 184082
```

4. Set time of day on AIOP

```
$ aiop_tool gettod -g dprc.4 -t 100010
<No output, if successful>
```

6.14.2.6 Known Limitations

1. Current hardware revision doesn't support Reset operation for AIOP. This restriction enforces that for loading an image through AIOP Tool, the state of the AIOP should be `DPAIOP_STATE_ RESET_DONE`.
2. It assumed that the DPRC passed to AIOP Tool has a single AIOP (`dpaiop`) and MC portal (`dpmcp`). This application would parse the contents of DPRC and stop on first found instances of `dpaiop` and `dpmcp` – even if multiple instances have been defined. The order of finding would be dependent on how objects are parsed from the sysfs directory.
3. Due to a limitation of the MC API, AIOP Tool instance doesn't automatically exit once executed (not a run-to-completion model). As soon as the AIOP image is successfully loaded, the AIOP Tool application will keep looping without relieving the foreground shell. Following are some consequences of this limitations:
   a. Once executed, the AIOP Tool application sits in foreground without releasing the Linux Shell it ran on. To obtain the Linux shell on which AIOP Tool was executed, AIOP Tool application should be pushed into background (or started in background) through Linux shell semantics.
   b. Once executed and backgrounded, another instance of the AIOP Tool over same DPRC cannot be executed. This also implies that status requests for the AIOP cannot be done, either through same instance or through another instance of the application. To obtain the AIOP status, use other available methods.

6.14.2.7 Sample VFIO Binding Script

```
#/*
# * Sample bind script for VFIO.
# */

DPRC_4=/sys/bus/fsl-mc/devices/dprc.4

if [ -e /sys/module/vfio_iommu_type1 ];
then
    echo "#1) Enabling interrupts"
    echo 1 > /sys/module/vfio_iommu_type1/parameters/allow_unsafe_interrupts
else
    echo "No VFIO support available."
    exit
fi

if [ -e $DPRC_4 ];
then
    echo "#1.1) dprc container driver override"
    echo vfio-fsl-mc > /sys/bus/fsl-mc/devices/dprc.4/driver_override
    echo "#1.2) Bind dprc.4 to VFIO driver"
    echo dprc.4 > /sys/bus/fsl-mc/drivers/vfio-fsl-mc/bind
fi

if [ -e /dev/vfio ];
then
    ls /dev/vfio/
else
    echo "No VFIO support available."
fi
```

6.14.2.8 Steps For Dynamic DPRC Suitable For AIOP Tool Using restool

Assuming that a DPL is available which has AIOP container defined in it, this section provides information and sample commands for creating a DPRC which is suitable for use with the AIOP Tool.

AIOP Tool expects a DPRC which contains at least one dpmcp and a dpaiop object. Further, the DPRC should be a non-root DPRC – so that user-space application (AIOP Tool) can access its content.

All the commands would be performed using `restool` compatible with the MC version for which AIOP Tool is targeted. The sample steps below assume that there are enough resources to create a MC portal (`dpmcp`) and the AIOP object (`dpaiop`).

1. Creating a new DPRC which is child object of root DPRC, `dprc.1`.

   ```
   $ restool dprc create dprc.1 --
   options=DPRC_CFG_OPT_SPAWN_ALLOWED,DPRC_CFG_OPT_ALLOC_ALLOWED,DPRC
   _CFG_OPT_IRQ_CFG_ALLOWED
   ```

   The output would be similar to:

   ```
   dprc.2 is created under dprc.1
   ```

   Note the new DPRC name, `dprc.2`, in the above output sample. Steps hereafter assume that new DPRC is `dprc.2`; please replace DPRC name in subsequent steps where required.

State of newly created DPRC can be confirmed using following command:

```
$ restool dprc info dprc.2
```

Output would be similar to:

```
container id: 2
icid: 26
portal id: 3
version: 5.1
dprc options: 0x43
DPRC_CFG_OPT_SPAWN_ALLOWED
DPRC_CFG_OPT_ALLOC_ALLOWED
DPRC_CFG_OPT_IRQ_CFG_ALLOWED
```

2. *In case dpaiop object already exists in any DPRC, skip to Step 3 below.* The steps below are for creating a `dpaiop` object in the `dprc.1`. This requires information about the AIOP container (`dprc.3` has been assumed below).

```
$ restool dpaiop create --aiop-id=0 --aiop-container=dprc.3
```

In the above command, it is assumed that the AIOP container is dprc.3. The output would be similar to:

```
dpaiop.0 is created under dprc.3
```

The `dpaiop.0` object would be available in the `dprc.1` after this command.

3. Assuming that a `dpaiop` object, `dpaiop.0`, is available in the root `dprc.1`; Unplug the `dpaiop.0` in the `dprc.1` so that it can be assigned to a new DPRC.

```
$ restool dprc assign dprc.1 --child=dprc.1 --object=dpaiop.0 --
plugged=0
```

Check the status of dpaiop.0 object using

```
restool dprc show dprc.1
```

to confirm if it has been put in '`unplugged`' state. If `dpaiop` object is already part of another DPRC, it has to be assigned to unplugged and assigned to `dprc.1` before it can be assigned to a targeted container. Steps for that would be similar to:

Unplugging the dpaiop object state

```
$ restool dprc assign <dprc.X> --object=dpaiop.0 --plugged=0
```

Above command sample assumes dpaiop.0 is present in `dprc.X` Thereafter, move the dpaiop.0 object to dprc.1

```
$ restool dprc unassign dprc.1 --child=dprc.X --object=dpaiop.0
```

Hereafter, `dpaiop.0` is available in `dprc.1` to be moved to a target DPRC – as described in steps below.

4. Assign the `dpaiop.0` object to `dprc.2` and toggle state to `plugged`. This would work only if `dpaiop.0` is already unplugged in root DPRC.

```
$ restool dprc assign dprc.1 --child=dprc.2 --object=dpaiop.0 --
plugged=1
```

Check the status of `dpaiop.0` object using

```
restool dprc show dprc.2
```

to confirm if it has been put in 'plugged' state.

5. Create a new MC portal object

```
$ restool dpmcp create
```

This result in an output similar to:

```
dpmcp.4 is created under dprc.1
```

New MC portal dpmcp.4 has been created under root DPRC. This needs to be moved to dprc.2 – as shown in next step.

NOTE

Note: It is possible to re-use an existing MC portal using steps similar to those shown below. It is possible that dpmcp object is already bound to certain driver (for e.g. VFIO driver), in which case, it needs to be unbound before being moved out.

6. Move dpmcp.4 from dprc.1 to dprc.2 and move to 'plugged' state.

```
$ restool dprc assign dprc.1 --child=dprc.2 --object=dpmcp.4 --
plugged=1
```

Check the status of dpmcp.4 object using

```
restool dprc show dprc.2
```

to confirm if it has been moved into dprc.2 and put into 'plugged' state.

7. Confirm that both the objects, dpmcp.4 and dpaiop.0, are part of dprc.2

```
$ restool dprc show dprc.2
```

Output would be similar to:

```
dprc.2 contains 2 objects:
object       label      plugged-state
dpaiop.0                plugged
dpmcp.4                 plugged
```

Hereafter, the DPRC dprc.2 is ready to be used with AIOP Tool once VFIO binding is done. See Sample VFIO Binding Script on page 540 above for information on VFIO binding.

6.14.3 AIOP User Manual

6.14.3.1 AIOP User Manual PDF

Click here to access the AIOP Manual PDF.

6.14.4 AIOP Program Profiling

6.14.4.1 Overview

This document describes techniques for performance enhancements for software developed for AIOP. The reader should understand basic AIOP architecture and service layer API.

The structure of this document is the following:

- Explain the performance capabilities and limits of different processing elements of AIOP that should be taken into account at design time and in optimization time.
- Specific methods for identifying bottlenecks for different processing elements and memory subsystem

Readers should also have access to the CodeWarrior Development Studio for Advanced Packet Processing product inside a larger software suite called CodeWarrior Development Suites for Networked Applications (CW4NET), which includes the AIOP Analysis Tool and Scenarios tool.

AIOP Analysis Tool trace examples are used throughout this document. Users should be familiar with this tool since it is regularly used to debug and profile AIOP. Meanwhile, the Scenarios Tool is crucial when measuring memory bandwidth.

6.14.4.2 AIOP Program Design: Budgets Per Processing Elements

AIOP developers must consider the available capacity for each hardware element and design their programs accordingly.

For example, the LS2085A AIOP has 16 cores running at 800 MHz. This means that 800 x 16 = 12,800 million core cycles per second are available.

If instructions per cycle (IPC) is about 0.75 for each core, cores can execute about 10,000 million instructions per second. In order to handle 10 million frames per second, a budget of 1,000 instructions per each frame is available.

In order to achieve 17 million packets per second (MPPS) (which is approximately line rate for 2 x 10GE for 128-byte packet), maximum 588 instructions can be used to process a packet.

The Performance Capacity per Resource table below shows performance budgets for some AIOP operations.

Table 113. Performance Capacity per Resource

	Use case and other Information	**Performance capacity – Rev1**
Cores	16 x 800 MHz MCPS (million cycles per second) x 0.75 IPC = 9600 Instraction per second	16x800Mhz
OSM (Ordering Scope Manager)	Enter/exit pair, transitions	80 MOPS (million operations per second)
TMan (Timers Manager)	Timer commands per second (assuming 10M timers)	1 M [Timer fires/sec]
	Timer tasks initiated per second (assuming 10M timers)	1 M [commands/sec]
STE (Stats Engine)	Number of STE commands per second	68 M [commands/sec]
CDMA/FDMA (Context/Frame DMA)	17 MPPS packet presentations/ enqueues with 3 CDMA operation combined	
Tables	Exact Match (EM) key size up to 124 B;	51 MOPS

Table continues on the next page...

Table 113. Performance Capacity per Resource (continued)

	LPM key sizes, 4 byte EM + 4 byte LPM (IPv4) or 4byte EM + 16 byte LPM (IPv6)	17 MOPS
	Management commands on a 10 K rules balanced tree on PEB	500 K [commands/sec]
	Total 5 lookups at 17 MPPS: 3 Exact Match + 1 LPM + 1 MFLU	
	17 MOPS. ACL key size up to 56 B	17 MOPS
Parser	Max parser performance capacity	34 MOPS
	Max accesses to CTLU at 17 Mpps, including parser	6
	Typical use case at 17 Mpps	5 lookups + 1 parser

6.14.4.3 AIOP Program Profiling and Performance Tuning

AIOP programs have tight performance requirements and developers need to profile their applications in order to improve their performance characteristics.

There are two stages in profiling AIOP applications:

1. Finding bottlenecks in the application
2. Fixing bottlenecks found for each specific application

An AIOP task is a sequence of jobs executing on different processing elements. The performance of the overall task is dictated by the performance of the weakest element, which makes the weakest element a bottleneck of the application.

Take for example a task that involves receiving a frame, doing a look up, and sending the frame to another interface.

This task uses the core, FDMA, CDMA and TLU processing elements. Assuming that the core is going to run for 1000 cycles per task and that the task has 2 FDMA jobs, 2 CDMA jobs and one LPM CTLU job, how many tasks per second can the AIOP handle?

To answer this question, calculate the number of jobs each processing element can handle. Use the following assumptions to solve the example above:

- 16 cores running at 800 MHz
- Based on table for rev1, we can see that AIOP can perform 17 million of operations of 3 FDMAs + 2 CDMAs per second
- CTLU can perform 17 million LPM lookups per second

Because there are 16 x 800 MHz = 12,800 million cycles per second, 12,800/1,000 = 12.8 million tasks per second can be processed.

FDMA/CDMA: 17 millions per second = throughput of 17 million tasks

CTLU: 17/1 = throughput of 17 million tasks

Based on this analysis, the estimated maximum performance boundary for this application will be 12.8 tasks per second and its bottleneck is core performance. We do not take into account other possible artifacts, such as synchronization constrains etc. just for simplification of initial analysis.

The analysis above is very useful for initial estimation during the design and implementation stages as it allows the programmer to design the program with specific performance goals in mind. For more information on performance

characteristics of different processing elements see AIOP Program Design: Budgets Per Processing Elements on page 543.

Installing the AIOP Analysis Tool

In order to find bottlenecks within applications, AIOP tools are used; one in particular is the AIOP Analysis Tool.

The AIOP Analysis Tool is a component of CodeWarrior Development Studio for Advanced Packet Processing product inside a larger software suite called CodeWarrior Development Suites for Networked Applications (CW4NET). The AIOP Analysis Tool is available for download on the NXP Semiconductors website.

To download the AIOP Analysis Tool, click on the "Downloads" tab and under "CodeWarrior Development Studio for Advanced Packet Processing", select "Latest Version". Click on "CodeWarrior for Advanced Packet Processing Evaluation / Updates" and download the installer. After the download is complete, run the installer:



Figure 41. CodeWarrior for Networked Applications Installer

After running the installer, follow the Setup Wizard and Click "Next":



Figure 42. CodeWarrior for Networked Applictions Setup Wizard

The AIOP Analysis Tool has to be selected within the components to install. After confirming the selection, the installer will download the tools automatically according to the selection:



Figure 43. CodeWarrior for Networked Applictions Tool Selection

For more information, please refer to the CodeWarrior Development Studio for Advanced Packet Processing.

Bottleneck analysis of existing application

This section explains in detail how to find bottlenecks within applications using the AIOP Analysis Tool. This tool captures the scheduler trace, presents it in graphical way and provides some statistics based on this trace.

The figure below shows a code snippet and its associated trace for a simple reflector program:



Figure 44. Trace of Reflector Application

The trace measures the length of each job in the task and shows the utilization of resources. (hover the mouse to highlight a specific job):



Figure 45. Task Information

What is the bottleneck of this application? In the following sections, we will show how you can find bottlenck based on this trace

6.14.4.4 FDMA/CDMA

The implementation of the CDMA/FDMA module is the following:

- FDMA and CDMA are actually one hardware block sharing the same resources.
- Thread = an FDMA/CDMA context which executes job on behalf of a task. 64 are present in rev1.
- Foreground slot = a HW execution resource that is loaded with a thread (and its context) in order to perform work on the thread (execute its command). 16 are present in rev1.
- Background slot = a HW resource that holds a suspended thread. This is a thread that is assigned (from a software point of view), but is blocked waiting for a high latency action to complete.
- Thread switching = moving FDMA threads between foreground (execution) and background (suspended) slots.

In the trace window we can see the number of assigned threads. This number is not precise due to how we measure it, but it can provide a good estimate.

Table 114. FDMA/CDMA Jobs

Core 13	63.12% Utilization
Core 14	63.97% Utilization
Core 15	63.23% Utilization
PC-CTLU	1.43 job average
TL-CTLU	11.86 job average
P.FDMA	3.65 job average
CDMA	20.91 job average
O.FDMA	10.49 job average
Total tasks	167
Tasks created	9,060,691.863 tasks/sec
Tasks finished	8,666,748.738 tasks/sec

We see here that the following number of threads are utilized:

1.43 + 11.86 + 3.65 + 20.91 + 10.49 = 48.34

48 is smaller than 64 so we are not running short of number of FDMA/CDMA threads.

Example of overall task analysis

Here we look at task trace of a reflector application and and analyzing it trying to find a bottlneck.

First, determine how many tasks are executing in parallel with the AIOP by looking at the following trace:



Figure 46. Trace Analysis

For simplicity we show only snapshot of tasks with higher numbers. Task scheduler begins scheduling with those tasks as well. Trace tool only shows tasks that were active at some point.

On core 15, only 6 tasks (Task 250 to Task 255) are executing, instead of 16 tasks. There are two possible reasons for this:

- The AIOP is underutilized; not enough traffic is sent to it.
- The FDMA/CDMA is a bottleneck as *only FDMA/CDMA can push back* to the work scheduler.

If enough traffic is sent to saturate the AIOP, but not all AIOP tasks are utilized, then the FDMA/CDMA is the bottleneck of your application.

Bottlenecks caused by CDMA/FDMA:

- Run out of Foreground slots. This may happen when memory becomes very congested and the CDMA waits for transactions for a long time. This bottleneck can be identified by looking at the average number of CDMA/FDMA jobs. If this number is close to the number of threads, then the bottleneck is within the CDMA/FDMA.
- Runs out of Threads. This happens when all CDMA/FDMA foreground slots are busy executing. This condition rarely happens as the previous condition happens first.

Strategies for reducing FDMA/CDMA bottlenecks

- Reduce the size of the initial presentation. The FDMA block is sensitive to presentation size; reducing presentation size from 128 to 64 Bytes may significantly reduce the pressure on the FDMA. For example, the figure below shows that task creation job (which is using FDMA) for 64 Byte presentation takes 233 cycles, while a 128-Byte presentation can take over 500 cycles. For IPv6 frames (or for any bigger header frames), initially present a bigger area or re-represent more data after determining that one is dealing with an IPv6 frame.



Figure 47. Task Creation Job Using FDMA

- Reduce the number of CDMA calls. Look for opportunities to combine one or more reads or writes. This will reduce pressure on the CDMA block and memory. For example, when reading two integers, a and c, in struct such as:

```
struct {
int a;
int b;
int c;}
```

  It is better to read both integers in one read, even if that means also reading c.

- Mutex – Mutex on rev1 can be called 12 million times per second when multiple Mutex IDs are used and about 5 million times per second if a single ID is used.
- Try to align and pack all CDMA accesses. For example, when reading 64 bytes of data per packet, place it in DDR memory with a 64-byte alignment. In any case, try not to cross 64-byte boundaries when possible, as it will cause two accesses instead of one.
- Allow sufficient headroom for frame changes (inserts). There is an API that can override some parameters in the default storage profile, which should be called during the early initialization stage dpni_drv_register_rx_buffer_layout_requirements().
- Call once for multiple changes. For example multiple changes in a frame header can be made using a single FDMA call.
- Do not represent segments (for example frame data segments) when possible.
- While a frame is open for the AIOP task, it may be edited with FDMA commands, but the changes may not be visible to consumers outside the AIOP. There is no need to do an FDMA store command unless those updates must be visible to consumers outside the AIOP (e.g. an external memory).
- Some FDMA commands have options to encapsulate other commands, creating compound commands. Use compound commands when appropriate. See the table below.

Table 115. FDMA Compound Commands

Command	Combined Options
Present (open) Frame	Present frame data segment
Enqueue Frame	Terminate task Discard frame (when enqueue fails) Relinquish OSM exclusivity in current scope right after the enqueue to QMan is issued
Replace Segment Data	Represent Segment Data Close segment
Concatenate Frames	Close concatenated frame
Split Frame	Close new frame Present new frame data segment
Replicate Frame	Enqueue new frame, optionally Relinquish OSM exclusivity Discard source frame

6.14.4.5 Core Profiling

Strategies for reducing core bottlenecks

Core bottlenecks are easily identified by examining the core load in the trace tool. If the core load gets close to 100% percent, it means that the bottleneck is core cycles. Generally, it is a good problem to have after the final optimization stage as the core is the most valuable resource. If the application has a bottleneck with the core, that means the core is being used to its maximum. This table is an example of usage of trace tool for core profiling:

Table 116. Core Utilization

Range Time	1118954.00 cycles
Core 0	99.60% Utilization
Core 1	83.13% Utilization
Core 2	93.11% Utilization
Core 3	99.61% Utilization
Core 4	99.62% Utilization
Core 5	86.42% Utilization
Core 6	83.14% Utilization
Core 7	89.74% Utilization
Core 8	86.37% Utilization
Core 9	86.43% Utilization

Table continues on the next page...

Table 116. Core Utilization (continued)

Core 10	83.15% Utilization
Core 11	89.72% Utilization
Core 12	86.43% Utilization
Core 13	89.68% Utilization
Core 14	89.83% Utilization
Core 15	99.92% Utilization

Example task analysis for core utilization

What can be done to resolve this problem?

First, look at the trace, identify all core jobs and check that job lengths are reasonable and expected. If you find something not expected, examine code that is executed for this job and try to find ShRAM accesses, make sure code is running from iRAM (for performance sensitive code), no accesses to PEB, DDR or regicters are made in your code. Access to DDR can take up to 200 cycles and should be avoided.

It is a good idea to measure IPC of your program so that you know that it is in a reasonable range. We use scenario tool to measure it on AIOP. In order to measure IPC directly in a running program, open the following scenario: `aiop_throughput_IPC_ssram_access` (*warning: you must disable run-time stack check in AIOP by undefining STACK_OVERFLOW_DETECTION as this feature is utilizing the same resources as this scenario)*

The IPC for each core is measured based on the following formula:

INSTR_COMPLETED / (PLATFORM_CLK - CTS_NO_TASK_CYCLES)

Based on the formula, divide the *number of instructions completed by each specific core* by the *number of cycles that core was executing tasks* (was not idle).

After measurement, the result is similar to the figure below:

IPC:core0	IPC:core1	IPC:core7	IPC:core8
0.58505	0.584909	0.584624	0.584829
0.585066	0.584925	0.584643	0.584851
0.585033	0.584884	0.584592	0.584803
0.585045	0.584917	0.584611	0.584821
0.585035	0.584905	0.584606	0.584838
0.585051	0.584911	0.584616	0.584803
0.585045	0.584902	0.58462	0.584821
0.58505	0.584915	0.584626	0.584822
0.585058	0.584908	0.584635	0.58484
0.585043	0.584924	0.584619	0.584834

Figure 48. Example IPC for Each Core

Generally, if the IPC is lower than 0.70, we need to investigate why. One of the most common reasons for low IPC would be accessing ShRAM (Shared SRAM) in the program. However if the resulting IPC is a really low number, then the code must not be running from IRAM.

Here is how one can calculate the approximate IPC based on number of ShRAM accesses and accelerator calls. In the previous scenario, we can look at following data:

Instructions per packet	Shared SRAM accesses per packet	Accel calls per packet
198	10	4
198	10	4
198	9.99998	4
198	10	4
198	9.99999	4
198	10	4
198	9.99999	4
198	10	4
198	10	4
198	9.99999	4
198	10	4
198	10	4

Figure 49. Packet Information: ShRAM Accesses and Accelerator Calls

The formula below was used to approximate the number of cycles spent by the core as a function of the number of ShRAM accesses, accelerator calls, and instructions executed.

N_Core_Cycles = (N_Instr * CPI) + (N_ShRAM * 12) + N_Accel_Calls * 4

In our case we get:

N_Core_Cycles = 198 * 1 + 10 *12 + 4*4 = 332

IPC ~ 198/332 = 0.596

For the example above, this low IPC is due to a high number of ShRAM accesses.

Improving core performance

- Inline the code where possible, especially for functions that are part of a “hot path”. Most of SL APIs are already in-lined.
- Reduce the number of accesses to shared memory. Shared memory has 12 cycles latency and 10 accesses will incur at least 120 cycles.
- Place all per-packet code in IRAM. For that, qualify such code with `__HOT_CODE`. For example:

```
__HOT_CODE uint64_t shbp_acquire(uint64_t bp, struct icontext *ic)
{
    struct shbp shbp;
    uint32_t offset;
    uint64_t buf;
    int err;
```

Figure 50. Example of `__HOT_CODE`

- Never place any performance sensitive code in DDR.
- Avoid floating-point operations and operands. The e200 core emulates floating-point operations/operands, which causes performance to be very low. Use fixed-point operations instead, when required.

- Use all available cores and maximize possible tasks per core. Typically, it will not be an issue to use all the cores, but using all tasks could be challenging sometimes because of stack size. If the stack requires more than ~1500 available entries to maximize the number of tasks, do the following:
  1. Run the stack static analysis tool and check where you use most of the stack
  2. Restructure your code, so that:

     `a() -> b() ->c()` chain that will require a lot of stack

     is replaced with the following:

```
a();
b();
c();
```

6.14.4.6 Memory profiling

AIOP programs may use different types of memories:

- external (DDR, PEB)
- internal (ShRAM, Workspace)

DP-DDR (DDR3 controller)

DDR memories are the first suspect to be oversubscribed. During the design stage of the system, developers should keep DDR bandwidth in mind.

The bandwidth of DP-DDR is 4 Bytes x 1.6 GHz = 6.4 GB/s

In a perfect situation where everything is aligned, accesses are all 32 bytes in length. However, this is not usually the case, but this number can provide a good starting point.

Example task analysis for DDR usage

So, let's assume we have context data in DP-DDR. It has 32 byte size and it accesses each frame with 17 MFPS.

We will get 32 x 17 MFPS = 544 MB/s, which is 544 MB/s / 6.4 GB/s = 8.5% of bandwidth

What happens when tables are placed into this memory?

The following is how look-up hardware accesses memories:

- LPM IPv4 takes 4*(1+alpha/2) memory accesses
- LPM IPv6 takes 6*(1+alpha/2) memory accesses
- EM takes (1+alpha/2) memory accesses

Alpha is the fill factor which goes from 0 to 1. The worst case condition is when alpha is 1. This means the tables fully utilize the CTLU memory.

Each access is 64 bytes.

Assume that the EM table has been placed in DP-DDR. Calculate the load on DP-DDR when running at 17 MFPS load.

The load will be:

1 x 64 x 17 MFPS / 6.4 GB/s = 0.17 = 17%

DP-DDR can safely be loaded up to 60%. However at 60% load, expect a spike in latency of DP-DDR. For example, it is impossible to place the LPM table there at 17 MFPS:

4 x 64 x 17 MFPS / 6.4 GB/s = 0.61 = 61%

Improving memory bottlenecks

Recommendations for table placement

The decision for each table placement should be made based on the following parameters:

1. Frequency—how often the table is accessed.
2. Size of the table—big tables will not fit in PEB.
3. Availability of bandwidth or place in specific memory.

Some guidelines:

- If DP-DDR is populated, it should be utilized as much as possible up to 60% of bandwidth, in order to reduce pressure on system DDR

For small tables that are used relatively frequently PEB could be the best candidate.

For System DDR, the L3 cache must be enabled in rev1.

For example:

Small (several hundred entries) ACL tables that are accessed per packet at a very high rate (millions times per second) should go to PEB memory.

Big LPM tables that are accessed infrequently should be placed in DP-DDR. The same table that is accessed higher than 60% DP-DDR utilization should be placed in System-DDR.

As a practical approach, we suggest not to use DP-DDR at the first stage of development and leave this optimization for later stages of development.

Recommendations for data placement

Data can be placed in several types of memory:

- System DDR
  - Direct access by core (not recommended) is ~200 cycles
- DP-DDR
  - Direct access by core (not recommnded) is ~200 cycles
- Shared RAM
  - ~10-15 cycles access
- PEB
  - ~40 cycles access by core (not recommnded)

When deciding about placement of different types of data, first consider how this data scales:

- Data scaling with number of flows
  - As number of flows is typically high (more than thousands), it should be placed in DDR
- Frequently accessed data that is scaling as number of interfaces or application instances
  - Such data should be placed in ShRAM (as statically defined data). Exceptions could be made for data that is accessed in bulk. In that case, place the data in PEB, so that it could be brought from DMA to local memory as a whole structure and then several data fields can be accessed with zero latency

6.14.4.7 CTLU - Parser

The CTLU or parser becomes a bottleneck of the application when tasks wait for the CTLU or parser for a long time. For example, the trace below shows this situation:



Figure 51. CTLU - Parser Bottleneck Example

Based on the trace above, it is evident that the wait time for the parser (PCCTLU - "Parse Classify CTLU") is very significant. The following are reasons for the long wait time:

- Too many calls to the parser/TLU
- The memory that the TLU works with experiences high load, which results in high latency. For memory issues, refer to the Memory profiling on page 555 section.

Example task analysis for CTLU usage

Improving CTLU bottlenecks

Listed below are some ideas on how to alleviate load on the parser:

- One of the features of DPNI is to calculate gross running checksum for ingress frames. So, when frame is received form by AIOP, it has associated valid gross running checksum. It is a checksum of the entire frame. Once we change frame data, the checksum becomes invalid unless we update it. If the parser is called with validation flags enabled, and the gross running sum was set to 0, it will first recalculate the gross running sum of the entire frame. When appropriate, do not invalidate (set to 0) the gross running sum field if the parser will later be called with the validation flags enabled.
- In case a VLAN header was added or removed, it is possible to call the software functions `parser_push_vlan_update()` or `parser_pop_vlan_update()` to update the workspace parse results instead of calling the hardware parser routines. This method should be used when the ratio of calls to the hardware parser routine is beyond the performance capacity. See the AIOP Program Design: Budgets Per Processing Elements on page 543 section.

6.14.4.8 OSM

It is assumed the reader is already familiar with the details of OSM operation and use. Here we are only concerned with refining the use of OSM to enable best performance. It is also assumed the reader has a basic familiarity with the AIOP analysis tool to view what is going on with task/job scheduling within the AIOP.

OSM is used to add order constraints on the scheduling of jobs of a task within the AIOP based on network ordering requirements of the packet being processed. Therefore the goal in optimizing performance when using OSM is to minimize how often the task scheduler will block a task's progress based on order constraints. In other words OSM *inhibits* jobs of a task from being scheduled and optimization minimizes the time it does so.

The image below demonstrates using the AIOP analysis tool to show how contention in an ordering scope looks like. The mouse when hovered over the pink bar in Task 251 shows additional information about this phase of a task execution. It shows in this example that the task was blocked for 1412 cycles because it needs exclusive phase of a particular ordering scope but it is blocked because other tasks are ahead of it.



Figure 52. Task Waiting on Ordering

The obvious optimization here is to reduce the number of tasks in the same ordering scope at a given time and to minimize the time each of these tasks keeps the exclusive phase of any ordering scope. An optimized software design will do both as much as possible while remaining correct. Also in a perfect optimization, all tasks would be in different ordering scopes or all tasks would use zero cycles in the exclusive phases of an ordering scope but this is never possible. The AIOP analysis tool will help the software designer to gain experience in the practical use of ordering scopes.

Example task analysis for OSM

Consider a simple example of a task. This task extracts the destination IP address as a key and does a TLU lookup based on that key. The lookup result is a reference to system memory that contains an ARP table entry. The entry is copied from system memory to local workspace. A hit counter in the entry is incremented and the system copy updated to reflect that. The entry also contains a MAC address which is used to replace the destination MAC address in the packet received and then forwarded.

```
key.dst = _presentation_with_vlan.ipv4hdr.dst_addr;
key_desc.em_key = (void*)&key;
table_lookup_by_key(TABLE_ACCEL_ID_CTLU, tid_arp, key_desc, sizeof(struct simple_key),
&lookup_result);
cdma_read( (void*)&arp_entry, lookup_result.opaque0_or_reference, 16);
arp_entry.hits++;
cdma_write(lookup_result.opaque0_or_reference + offsetof(struct arp_entry, hits),
(void*)&arp_entry.hits, 8);
memcpy(&_initial_presentation.ethhdr.da, &arp_entry.mac, 6);
```

```
fdma_modify_default_segment_full_data();
fdma_store_and_enqueue_default_frame_fqid( DESTINATION_FQ, FDMA_EN_TC_TERM_BITS);
```

This is a made up example which does no error checking. Assume from this simple example that forwarded frame is required to be in the same order as the frame arrived on a network interface for any flow. This is not a very realistic example but it will suit the purpose of observing the performance when OSM operations are included.

The simple method to meet the order requirement in this application is to run all packets as if they are in a single flow (single common initial scope ID) and in exclusive mode for the duration.



Figure 53. Tasks Executing Sequentially

A snapshot from the AIOP analysis tool shows the result of running all packets as if they are from a single flow in the exclusive phase of a common ordering scope. In this image, blue bars are hardware jobs, green are software jobs, and red is blocked. It is immediately obvious each packet is processed one at a time until done before a subsequent packet starts processing. This is the baseline to improve upon.

Spread based on flow

AIOP tasks will run in parallel more effectively if tasks of different flows begin in different initial scope IDs. In this way they, by definition, do not compete. To demonstrate this, the example of the previous section is repeated, all tasks start as exclusive, but now the packets come from 32 different flows and each flow defines an initial scope ID.



Figure 54. Tasks in Different Flows Running Parallel

As seen from the AIOP analysis tool trace, many operations are occurring in parallel. Both hardware jobs (blue) and software jobs (green) are happening in parallel. In this snippet only one task is observed as blocked. This is the most simple and effective way to improve parallel operation in the AIOP. However, it is never possible to know that ingress traffic will be in a large number of flows at any instant in time.

The DPNI may or may not allow sufficient distribution (spread) of flows. However for every application there is a unique key comprised of header fields to uniquely identify a flow. This unique key is reduced to a hash and used as a new scope ID

when classification must be refined within the AIOP. This would be step one of a task and it can be performed in concurrent mode of the initial scope ID to maintain parallel operation.

Either way, if DPNI does or does not provide sufficient spread of flow, it is the application responsibility to pick what header fields uniquely identifies a flow. Use dpni_drv_set_order_scope() to specify initial order scope construction from header fields so that tasks are created with the best possible initial order scope ID. This is the preferred and best performing method. If that is not possible then the user may create a key composition ID (KeyID) from the header fields required during application initialization and use that KeyID to generate a hash suitable for order scope ID after tasks are launched. See key_composition_rule_create().

Run concurrent where possible

In general an AIOP task begins execution in the concurrent phase of the initial ordering scope and delays transition to a subsequent scope in the exclusive phase as long as it is practical. In our example task it is possible to run concurrently up to the point where the ARP table is read and updated. Consider a single flow where our example application runs concurrently and then moves to exclusive after the TLU lookup.

```
osm_scope_transition_to_exclusive_with_increment_scope_id();
```



Figure 55. Tasks in a Flow Blocking for Exclusivity

The TLU will perform lookup operations in an atomic fashion. Take advantage of this characteristic of the AIOP accelerators by using them from a concurrent phase of an ordering scope where ever possible. As can be seen in the AIOP analysis tool snippet, the first hardware job, in this case a TLU lookup, is performed in parallel with other tasks. The remainder of the task is run exclusively and parallel operation ceases.

Relinquish exclusivity quickly

In our example application the read-modify-write of the ARP table entry must be done exclusively. However the MAC address update does not because the data and frame are private to the task. A relinquish exclusivity is inserted following the ARP update and another transition to exclusive is inserted prior to forwarding to remain ordered.



Figure 56. Tasks of a Flow Relinquishing Exclusivity Affect

Consider alternatives

Our example takes advantage of OSM to create an exclusive phase to perform a critical operation, namely to read-modify-write the ARP table entry. Note this is an *ordered* excusive operation. That is tasks will update their ARP entry in the order of packet arrival. In this example the update is just a hit counter and does not require update in order.

It is possible to take advantage of two properties of the hardware accelerators here. First, the sequential update of the ARP entry hit counter can be performed by the STE (statistics engine). Second, the atomic operation of CDMA commands can

assure consistent values regardless of other readers or writers. Our example can take advantage of these properties to rewrite our ARP entry handling.

NOTE

The atomic operation of CDMA read and write operations is a specific enhancement over the first revision of the silicon. This atomic behavior allows a reader to be certain a writer will not corrupt an entry; the value read will be valid but not ordered with respect to the writer.

Rewriting this section will look like the following:

```
cdma_read( (void*)&arp_entry, lookup_result.opaque0_or_reference, 16);
ste_inc_counter( (lookup_result.opaque0_or_reference + offsetof(struct arp_entry,
hits)),STE_MODE_64_BIT_CNTR_SIZE);
```



Figure 57. Tasks of a Flow Making Use of Atomic Accelerator Operations

In our alternative method of using STE, all hardware accelerations with the exception of forwarding can now operate in parallel even though all packets are of a single flow. Cycle count for the first four packets forwarding is down to about 3,000 cycles.

OSM Capacity

OSM itself is a resource of limited capacity. To maintain maximum task throughput the number of transitions and enter/exit scope pairs should be limited to a total of about five. Beyond that the maximum task rate will start to decline. It is rare that a task of a complexity to require more than five OSM operations will be limited by OSM before it is limited by other hardware accelerator throughput. However the AIOP analysis tool will be the designer's main insight into where bottlenecks are occurring within the AIOP.

Use osm_scope_enter_to_exclusive_with_new_scope_id() and osm_scope_enter_to_exclusive_with_increment_scope_id() instead which pick the best hardware options by default and have little software overhead

Additional Guidelines

Experience with insight provided by using the AIOP analysis tool will guide the user to best practices for improving parallel operation and relieving bottlenecks.

- Primarily use transition increment forms of OSM commands as they are the best performers and naturally partition an application's overall design into steps from ingress to egress.
- Reduce cycles in exclusive phases as much as possible.
- Use accelerators in concurrent phases as much as possible.
- Create new scope IDs to refine (distribute) flows.
- Take advantage of atomic accelerator characteristics in concurrent phases.
- Avoid using the osm_scope_enter() function as its many options and overhead take many cycles.
  - Use osm_scope_enter_to_exclusive_with_new_scope_id() and osm_scope_enter_to_exclusive_with_increment_scope_id() instead which pick the best hardware options by default and have little software overhead

- Consider splitting a long operation requiring the exclusive phase into multiple exclusive phases.

These are only guidelines to consider while experience will guide the designer. The AIOP analysis tool gives insight into the behavior of tasks. Often times the interaction between decisions is not obvious and some trial and error is required for best results.

6.14.4.9 Statistics Engine

The statistics engine will stall the core when it is overloaded. It is relatively easy to see this scenario on trace – core jobs will become much longer around statistics engine (STE) calls and the IPC will go down. In order to isolate an STE bottleneck, remove some STE calls and measure the IPC of the application. If the IPC grows significantly, then an STE bottleneck is being experienced.

Below are important rules that developers need to be aware of when implementing statistics counters on AIOP during the design stage of an application. The most important factor that these rules address is DDR bandwidth limitation.

1. Though seemingly simple, it is important to use as few counters as possible. Reduce counters if possible, and make counters optional where possible.
2. Frequently used counters should not be placed in DDR but rather in internal memory (PEB or ShRAM in some cases).
3. If the context is not “read-only” and a lock of some sort (mutex or OSM based) is taken, it is good practice to put counters in that context and update them by the core without using STE (statistics engine). Similarly, if the number of counters is big, it is good practice to update them using the CDMA engine.
4. Use compound STE operations, which allows two counters to be updated in one operation.

There are restrictions on alignments of counters that the STE API has to follow. It can be found at STE section of AIOP Service Layer API Reference Manual.

6.14.4.10 IP Fragmentation (IPF)

For best performance it is recommended to work concurrently, and move to Exclusive Mode (XX) ordering only before enqueuing the last fragment. From this point (moving to exclusive before enqueue of the last fragment) transition to concurrent is not allowed. This way fragments of different frames will be interleaved but ordering will be kept between the last fragments of different frames.

6.14.4.11 IP Reassembly (IPR)

- Ordering scope
  - Call the IPR in Concurrent mode (XC)
  - Do a per-frame flow distribution, according to the IP identification field
  - Have at least two available OSM scope levels when calling IPR
- Place the lookup tables on the PEB memory
- The input frame (fragment) should be stored in a single buffer
- The frame buffer size should be larger or equal to (16 * <max number of fragments>) + any offset, headroom and annotation
- API configuration parameters
  - Do not enable external statistics
  - Use timeout mode to be per reassembled frame
- Send in-order fragments (relevant to the sender side, usually in closed systems)

Table 117. IPR configuration options for best performance

Flags and options	Best Performance
IPR_MODE_TABLE_LOCATION_PEB	Set
IPR_MODE_EXTENDED_STATS_EN	Cleared
IPR_MODE_IPV4_TO_TYPE	Set
IPR_MODE_IPV6_TO_TYPE	Set

6.14.4.12 IPSec

IPSec module supports IPSec encaspulation and decapsulation as a part of service layer. There are several ideas on how to achive the best performance for IPSec:

- Use tunnel mode.
- Use IPv4 frames and outer header.
- Do not enable transport mode pad check.
- Do not use the DSCP set option for tunnel mode.
- Use the system DDR as the IPsec FM context memory (currently not programmable).
- Do not enable UDP encapsulation in transport mode
- Enable reuse buffer mode

The following table describes the IPsec functional module configuration parameters value for achieving the best performance.

Table 118. IPsec Configuration Options for Best Performance

Flags & options	Best performance
IPSEC_FLG_TUNNEL_MODE	Set
IPSEC_FLG_TRANSPORT_PAD_CHECK	Cleared
IPSEC_FLG_BUFFER_REUSE	Set
IPSEC_ENC_OPTS_NAT_EN	Cleared
IPSEC_ENC_OPTS_NUC_EN	Cleared
IPSEC_FLG_ENC_DSCP_SET	Cleared
IPSEC_FLG_LIFETIME_KB_CNTR_EN	Cleared
IPSEC_FLG_LIFETIME_PKT_CNTR_EN	Cleared
IPSEC_FLG_LIFETIME_SEC_CNTR_EN	Cleared
IPSEC_OPTS_ESP_ESN	Cleared
IPSEC_OPTS_ESP_IPVSN	Cleared
IPSEC_DEC_OPTS_ARSNONE	Set
IPSEC_DEC_OPTS_ARS32	Cleared
IPSEC_DEC_OPTS_ARS128	Cleared
IPSEC_DEC_OPTS_ARS64	Cleared

6.14.4.13 Appendix A

DDR bandwidth measurement

The Scenarios Tools allows a real time measurement of DDR bandwidth for all the controllers. Use this tool as a help guide for the configuration procedure.

As an example, in the LS2085A, there are three DDR controllers. In ***Scenarios Tools*** those are called DDRC1, DDRC2 and DDRC3.

DDRC1 and DDRC2 are for system memory and DDRC3 is for DP-DDR.

Add the utilization measurement as shown below, then press the green “Launch” button in the toolbar.



Figure 58. Utilization Measurement in Scenarios Tool

The measurement will complete in a few seconds, and the table of results will be displayed, as shown below.



Figure 59. Utilization Measurement Table of Results

The initial table display will show all measured events and metrics. Use the Measurement Chooser dialog to select the measurements of interest, in this case the utilization metrics.



Figure 60. Measurement Chooser Dialog



Figure 61. Selecting Utilization Metrics



Figure 62. Example of DDR Utilization

In last figure, observe that DDR1 and DDR2 are only 5% utilized. Generally, utilization of lower than 65% is considered to be low and not affecting latency.

6.14.5 AIOP Service Layer API Reference Manual

6.14.5.1 AIOP Service Layer API Reference Manual PDF

Click here to access the AIOP Service Layer API Reference Manual PDF.

6.14.6 AIOP SDK Applications Debug

6.14.6.1 AN5165 AIOP SDK Applications Debug PDF

Click here to access the AN5165 AIOP SDK Applications Debug PDF.

Chapter 7
Linux Kernel Drivers

7.1 Audio

7.1.1 Synchronous Audio Interface (SAI) LS1021A

Description

The following describes how to configure and test SAI audio driver for LS1021A TWR board. The integrated I2S module is NXP's Synchronous Audio Interface (SAI). The codec is SGTL5000 stereo audio codec.

RCW configuration

Refer to the below table for the RCW binary files name and location for LS1021A TWR boards.

Board	RCW binary
LS1021A TWR	ls1021atwr/RSR_PPS_70/rcw_1000.bin

Kernel Configure Options Tree View

Kernel Configure Tree View Options	Description
Device Drivers ---> <*> I2C support ---> [*] Enable compatibility bits for old user-space [*] I2C device interface [*] I2C bus multiplexing support Multiplexer I2C Chip support ---> <*> Philips PCA954x I2C Mux/switches [*] Autoselect pertinent helper modules I2C Hardware Bus support ---> <*> IMX I2C interface <*> Voltage and Current Regulator Support ---> [*] Regulator debug support [*] Provide a dummy regulator if regulator lookups fail [*] Fixed voltage regulator support <*> Sound card support <*> Advanced Linux Sound Architecture -> [*] OSS PCM (digital audio) API [*] OSS PCM (digital audio) API - Include plugin system [*] Support old ALSA API [*] Verbose procfs contents ALSA for SoC audio support ---> <*> SoC Audio for Freescale VF610 CPUs ---> <*> SoC Audio support for VF610 boards with sgtl5000	Enable ALSA SOC driver, I2C driver and EDMA driver.

Kernel Configure Tree View Options	Description
<*> DMA Engine support ---> <*> Freescale eDMA engine support support	

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_I2C_IMX	y/m/n	y	I2C driver needed for configuring SGTL5000
CONFIG_SOUND	y/m/n	y	Enable sound card support
CONFIG_SND	y/m/n	y	Enable advanced Linux sound architecture supports
CONFIG_SND_PCM_OSS	y/m/n	y	Enable OSS digital audio
CONFIG_SND_PCM_OSS_PLUGINS	y/m/n	y	Support conversion of channels, formats and rates
CONFIG_SND_SUPPORT_OLD_API	y/m/n	y	Enable support old ALSA API
CONFIG_SND_SOC_FSL_SAI	y/m/n	y	Enable SAI module support
CONFIG_SND_VF610_SOC	y/m/n	y	Enable ALSA SOC support
CONFIG_SND_SOC_VF610_SGTL5000	y/m/n	y	Enable ALSA SOC support
CONFIG_SND_SOC_SGTL5000	y/m/n	y	Enable codec sgtl5000 support
CONFIG_FSL_EMDA	y/m/n	y	Enable EDMA engine support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
sound/soc/fsl	ALSA SOC driver source

Verification in Linux

1. The following messages will be shown in the kernel boot process.

```
sgtl5000 1-000a: sgtl5000 revision 0x11
sgtl5000 1-000a: Failed to get supply 'VDDD': -19
1-000a: 1200 mV normal
sgtl5000 1-000a: Using internal LDO instead of VDDD
```

```
......
vf610-sgtl5000 sound.9:  sgtl5000 <-> 2b50000.sai mapping ok
......
ALSA device list:
  #0: FSL-VF610-TWR-BOARD
```

2. If the device nodes don't already exist, create directory /dev/snd/, and create device nodes with the following commands in /dev/snd/ directory.

```
mknod controlC0 c 116 0
mknod pcmC0D0c c 116 24
mknod pcmC0D0p c 116 16
```

3. On LS1021A TWR board, the HEADPHONES interface is J14. The MIC is on chip.
4. Run the following aplay or mplayer commands to test playback. Run the following arecord command to test record.

```
aplay -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo.wav
mplayer 44k-16bit-stereo.wav

arecord -d 10 -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo-10s.wav
aplay -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo-10s.wav
```

5. Use alsamixer to adjust the volume for playing by the options “PCM”. Use alsamixer to choose LINE IN or MIC.

7.1.2 Synchronous Audio Interface (SAI) LS1012A

Description

This document describes how to configure and test SAI audio driver for LS1012A FRDM board. The integrated I2S module is NXP's Synchronous Audio Interface (SAI). The codec is SGTL5000 stereo audio codec.

RCW configuration

Refer to the below table for the RCW for Audio on the LS1012A FRDM board.

Board	RCW
LS1012A FRDM	Bit 364, EC1_EXT_SAI2_TX = 1; Bit 365, EC1_EXT_SAI2_RX =1; Bit 366-367, EC1_BASE = 00

Kernel Configure Options Tree View

Kernel Configure Tree View Options	Description
Device Drivers ---> <*> I2C support ---> [*] Enable compatibility bits for old user-space [*] I2C device interface [*] I2C bus multiplexing support Multiplexer I2C Chip support ---> <*> Philips PCA954x I2C Mux/switches [*] Autoselect pertinent helper modules	Enable ALSA SOC driver, I2C driver and EDMA driver.

Kernel Configure Tree View Options	Description

I2C Hardware Bus support --->
  <*> IMX I2C interface
 	
<*> Voltage and Current Regulator Support --->
  [*] Regulator debug support
  [*] Provide a dummy regulator if regulator lookups fail
  [*] Fixed voltage regulator support
 	
<*> Sound card support
  <*> Advanced Linux Sound Architecture ->
  [*] OSS PCM (digital audio) API
  [*] OSS PCM (digital audio) API - Include plugin system
  [*] Support old ALSA API
  [*] Verbose procfs contents
  ALSA for SoC audio support --->
  SoC Audio for Freescale CPUs --->
  <*> Synchronous Audio Interface (SAI) module support
  CODEC drivers --->
  <*> Freescale SGTL5000 CODEC
  <*> ASoC Simple sound card support
<*> DMA Engine support --->
  <*> Freescale eDMA engine support support	

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_I2C_IMX	y/m/n	y	I2C driver needed for configuring SGTL5000
CONFIG_SOUND	y/m/n	y	Enable sound card support
CONFIG_SND	y/m/n	y	Enable advanced Linux sound architecture supports
CONFIG_SND_PCM_OSS	y/m/n	y	Enable OSS digital audio
CONFIG_SND_PCM_OSS_PLUGINS	y/m/n	y	Support conversion of channels, formats and rates
CONFIG_SND_SUPPORT_OLD_API	y/m/n	y	Enable support old ALSA API
CONFIG_SND_SOC_FSL_SAI	y/m/n	y	Enable SAI module support
CONFIG_SND_SOC_GENERIC_DMAENGINE_PCM	y/m/n	y	Enable generic dma engine for PCM
CONFIG_SND_SIMPLE_CARD	y/m/n	y	Enable generic simple sound card support
CONFIG_SND_SOC_SGTL5000	y/m/n	y	Enable codec sgtl5000 support

Table continues on the next page...

Table continued from the previous page...

CONFIG_FSL_EMDA	y/m/n	y	Enable EDMA engine support

Source Files
The driver source is maintained in the Linux kernel source tree.

Source File	Description
sound/soc/fsl	ALSA SOC driver source

Verification in Linux

1. The following messages will be shown in the kernel boot process:

```
sgtl5000 5-000a: sgtl5000 revision 0x11
sgtl5000 5-000a: Using internal LDO instead of VDDD
......
asoc-simple-card sound:  sgtl5000 <-> 2b60000.sai mapping ok
......
ALSA device list:
  #0: 2b60000.sai-sgtl5000
```

2. If the device nodes don't already exist, create directory /dev/snd/, and create device nodes with the following commands in /dev/snd/ directory.

```
mknod controlC0 c 116 0
mknod pcmC0D0c c 116 24
mknod pcmC0D0p c 116 16
```

3. On LS1012A FRDM board, the LineOut interface is J8 and the LineIn interface is J13
4. Run the following aplay commands to test playback. Run the following arecord command to test record.

```
aplay -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo.wav

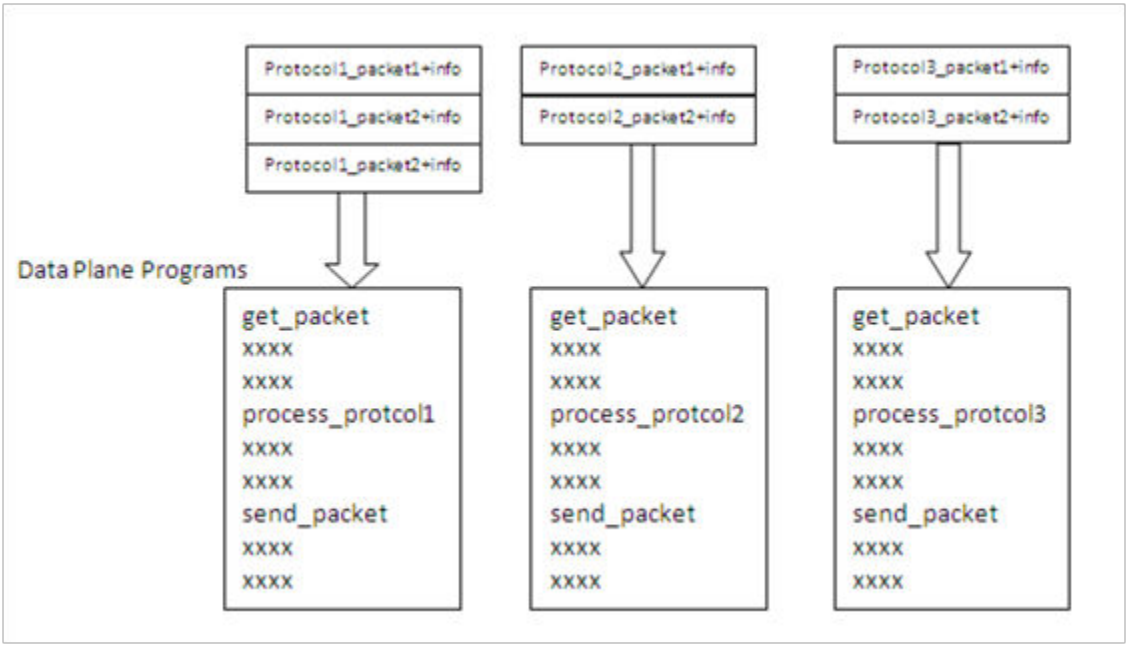
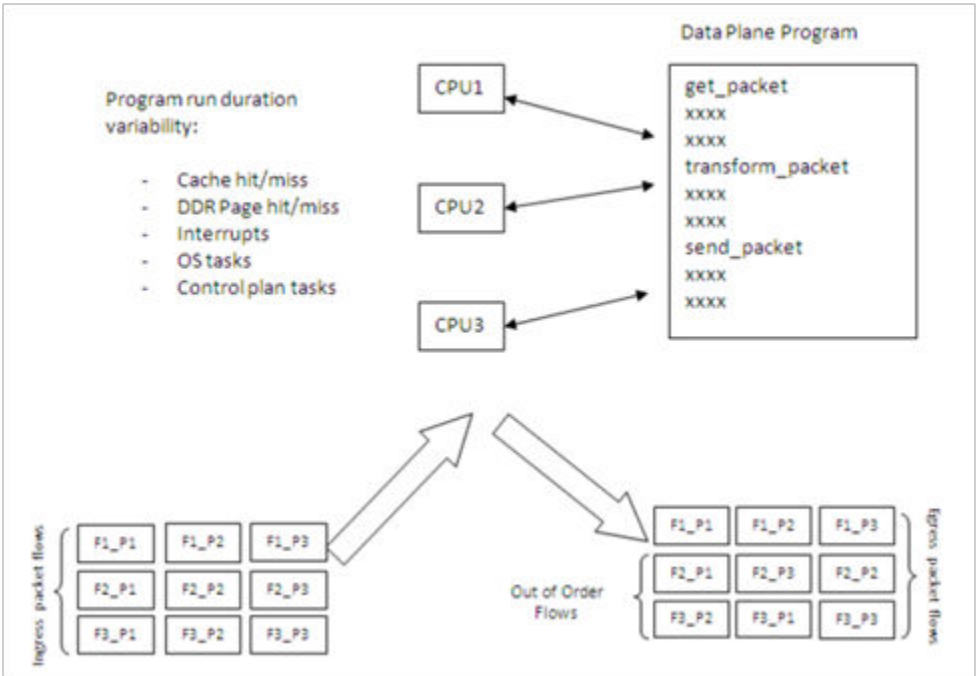
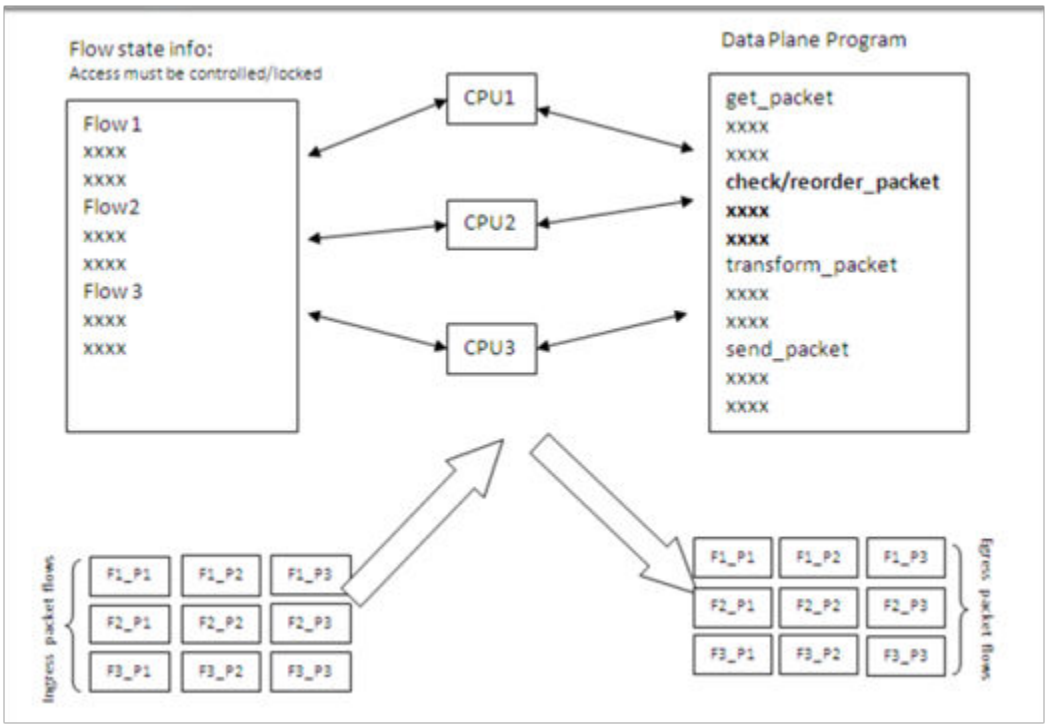
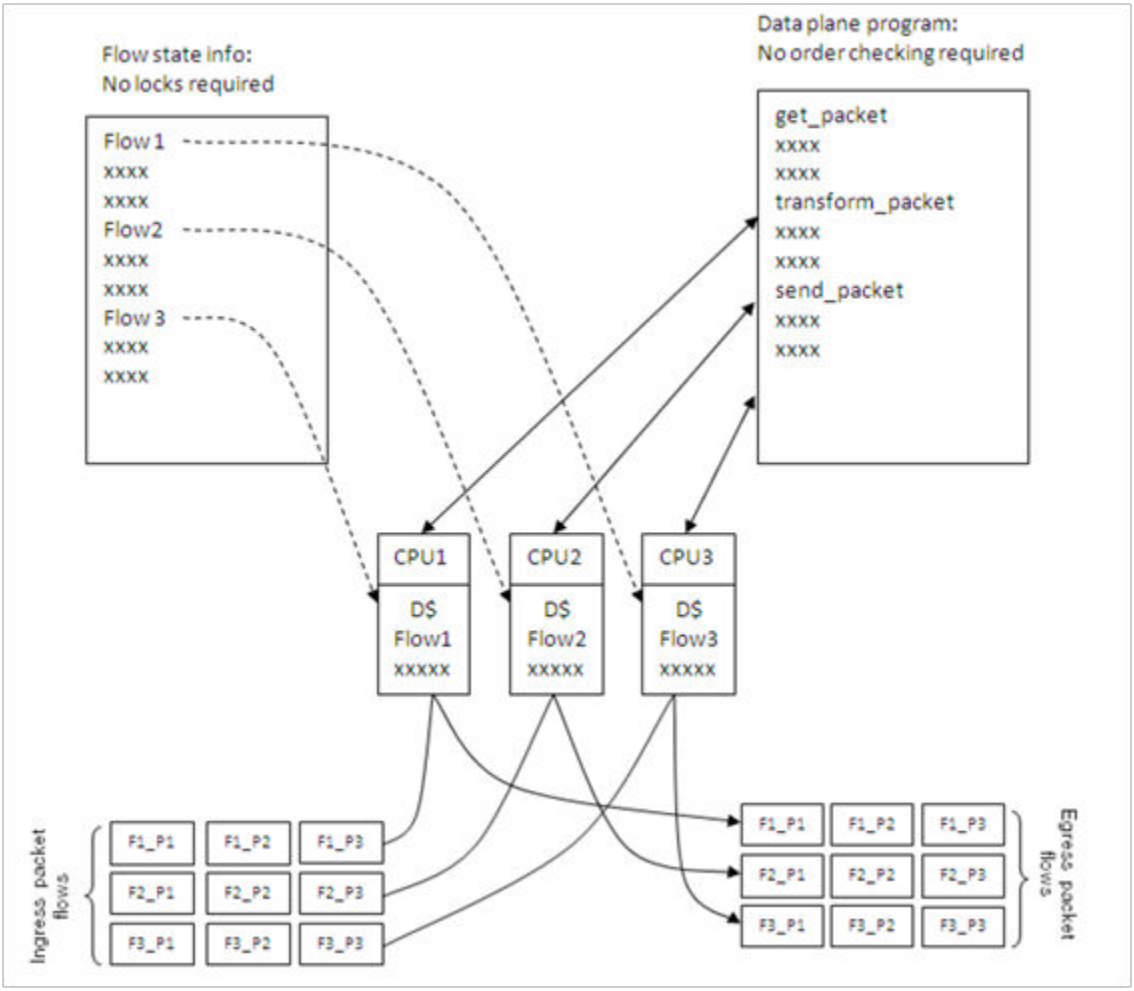
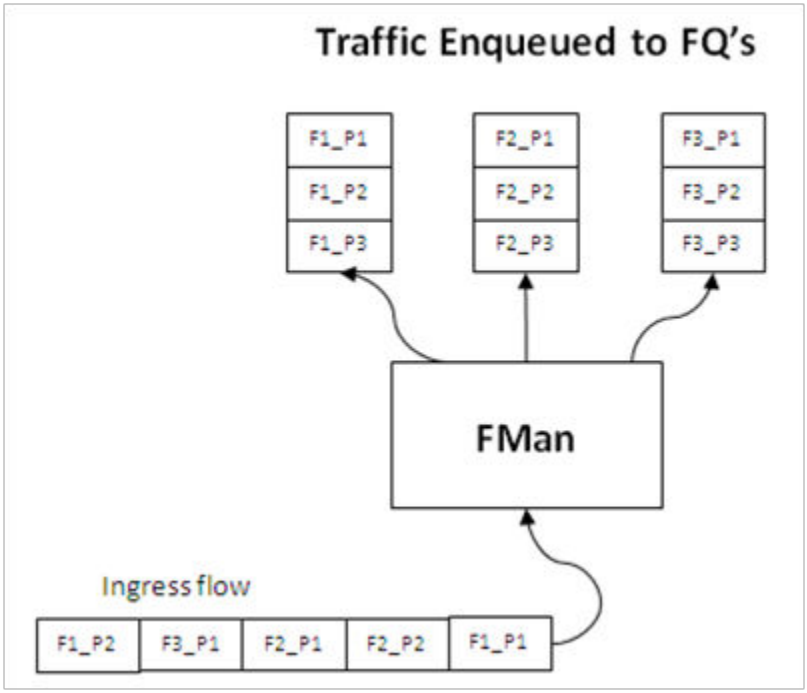
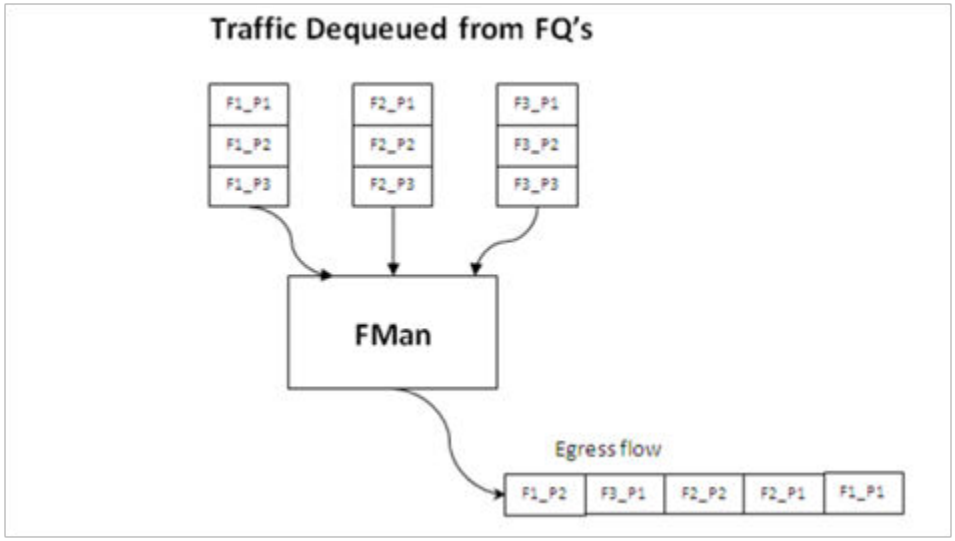
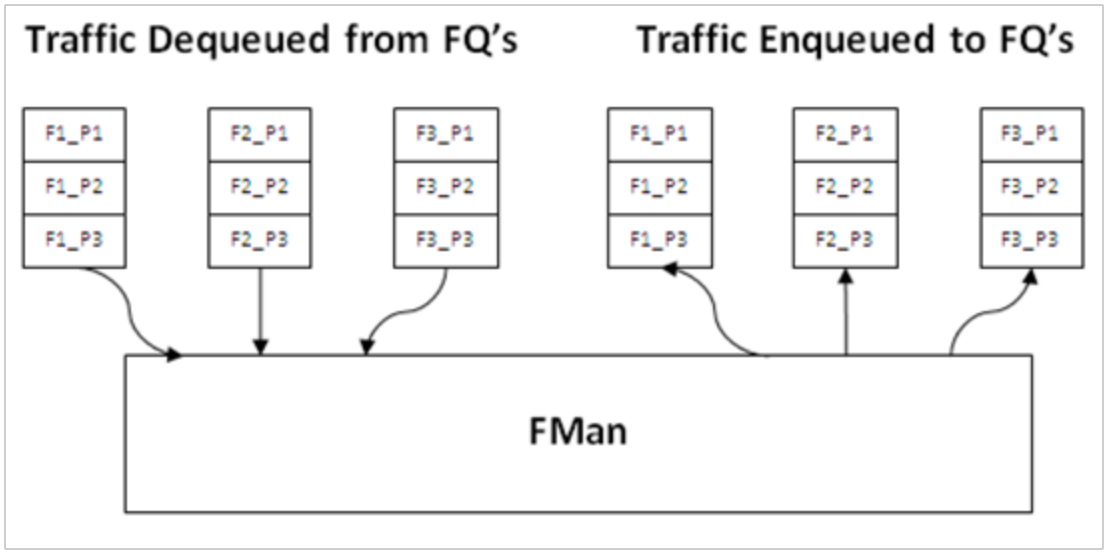
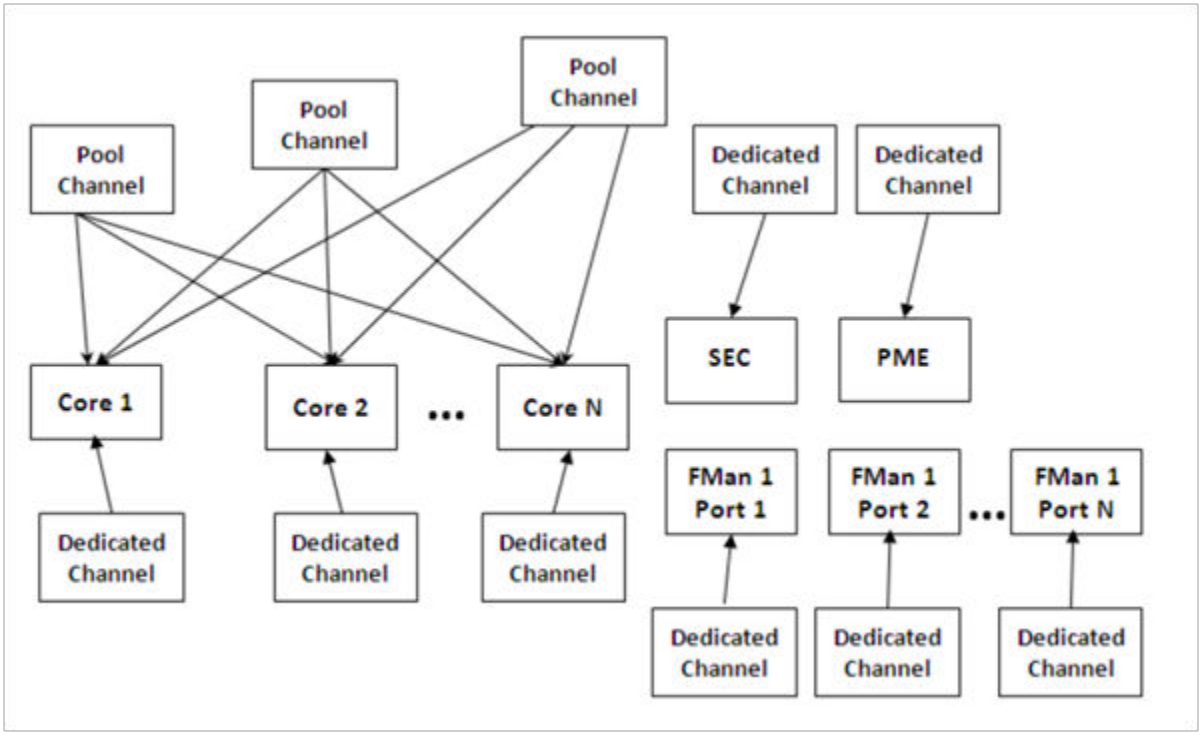
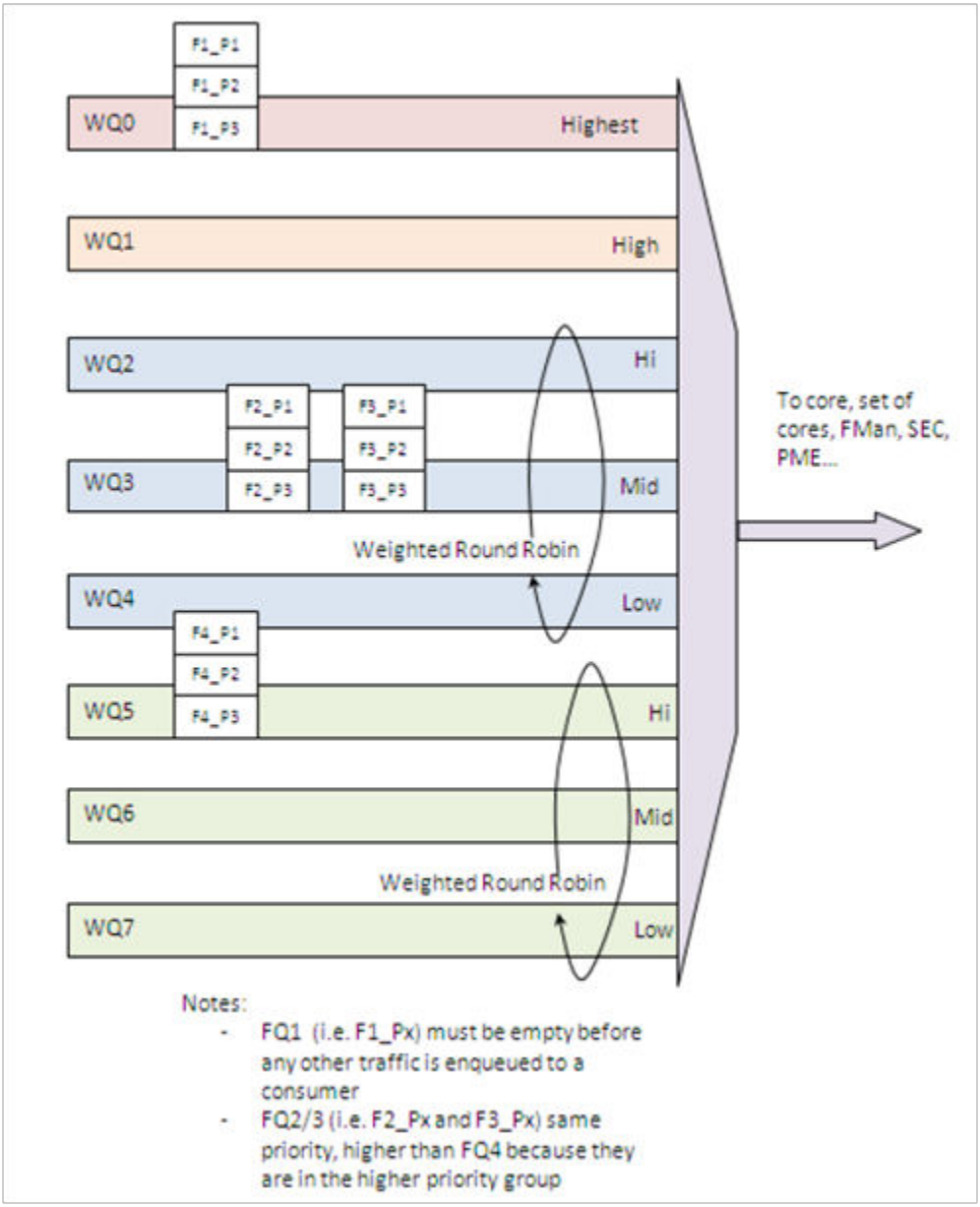
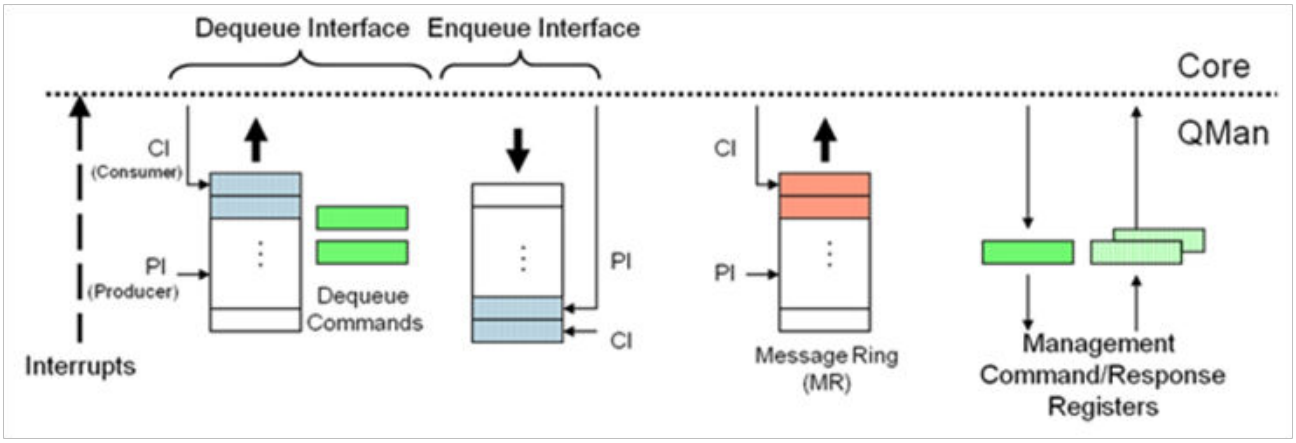
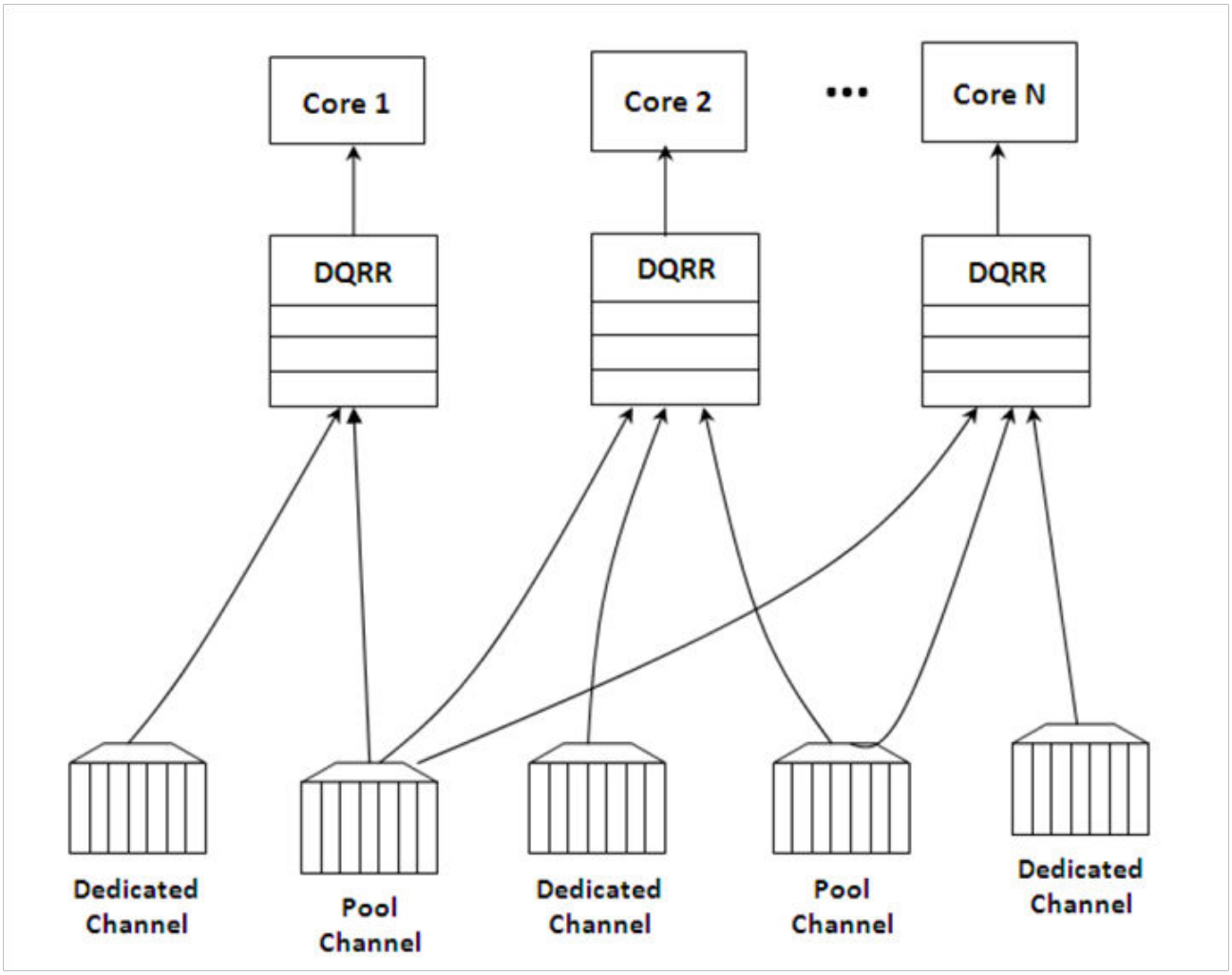
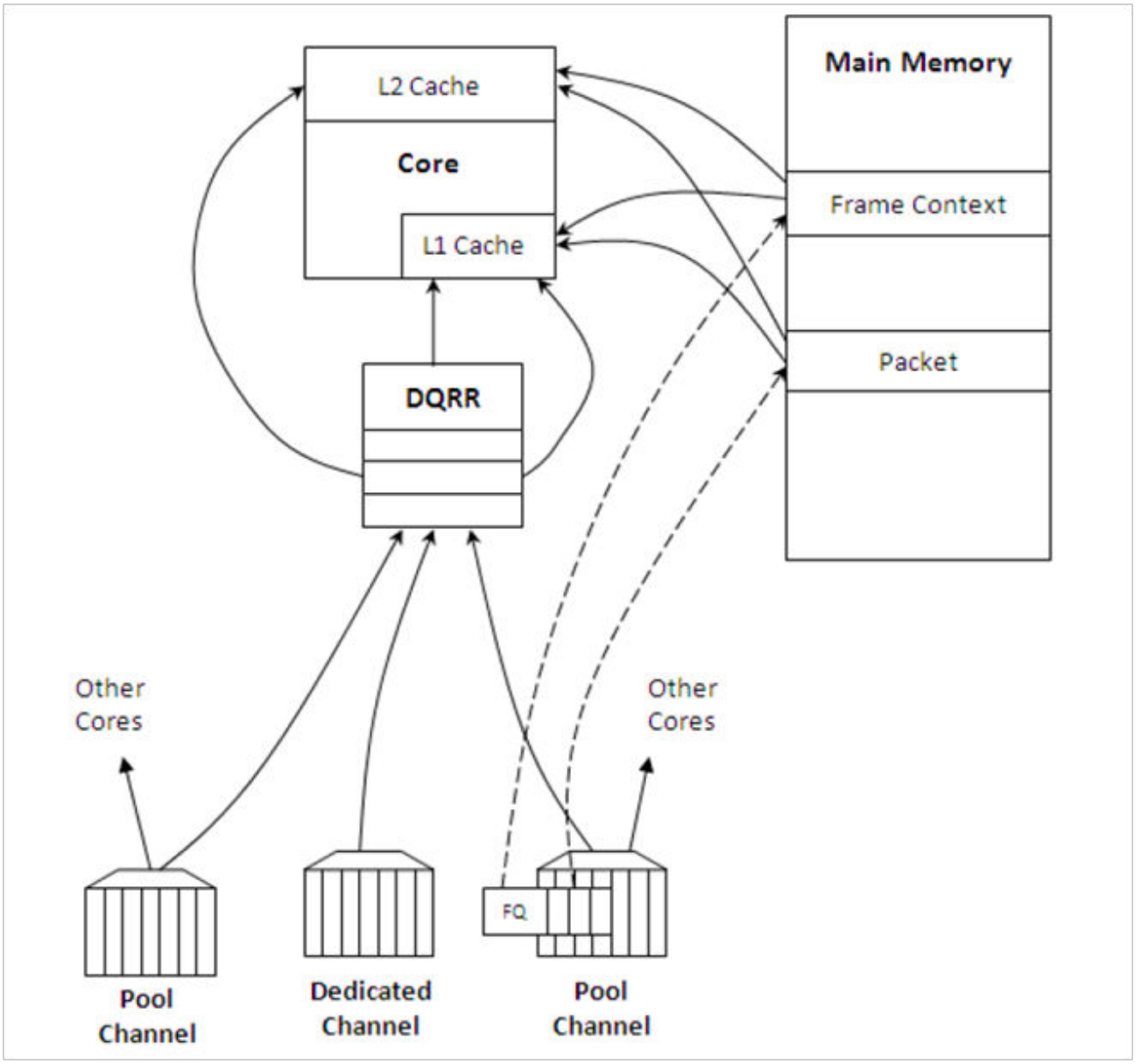
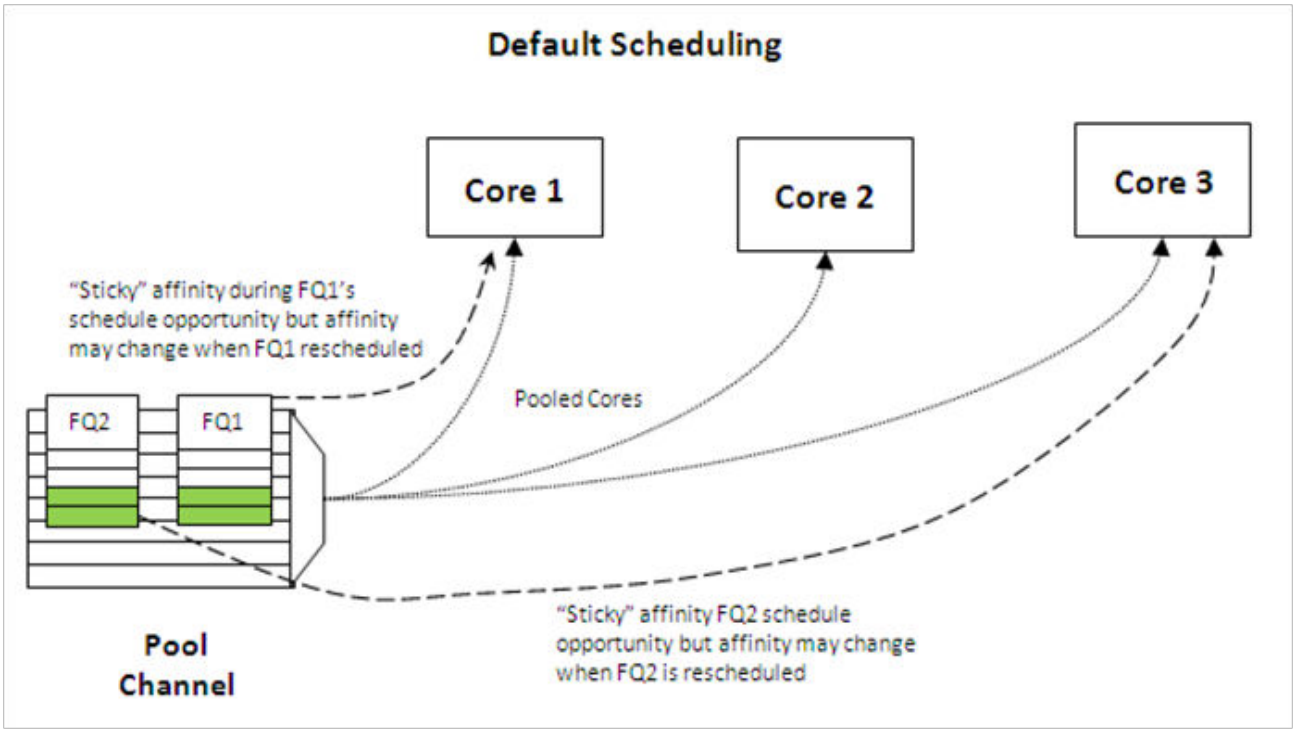
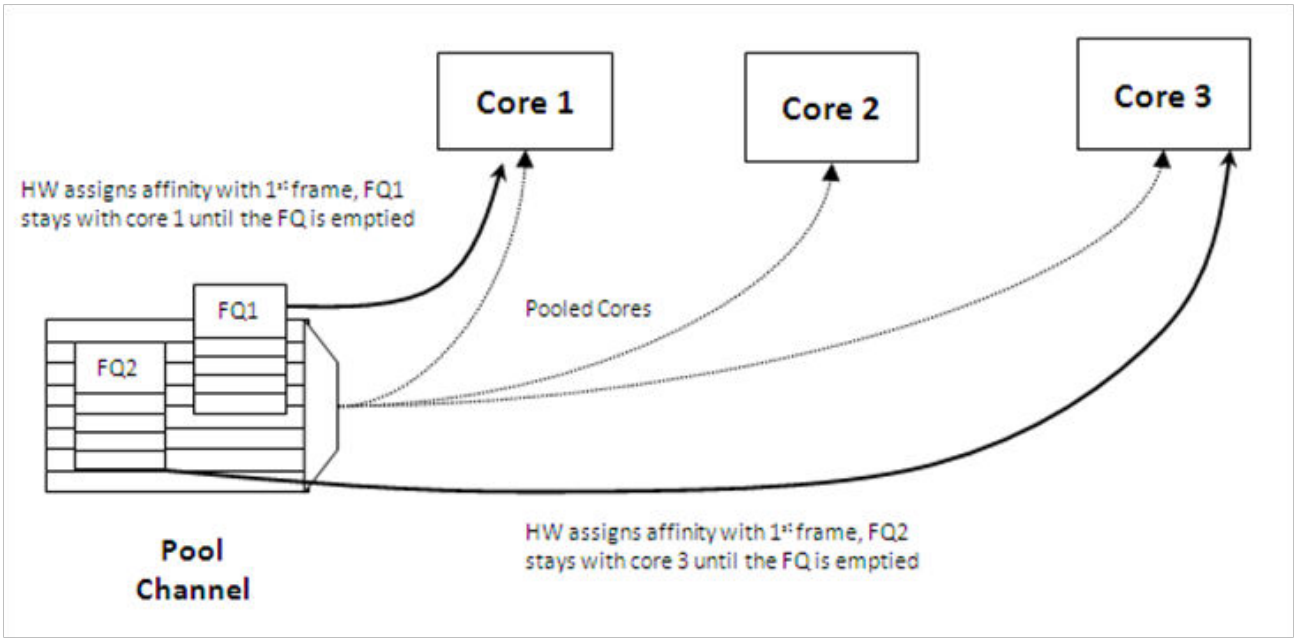
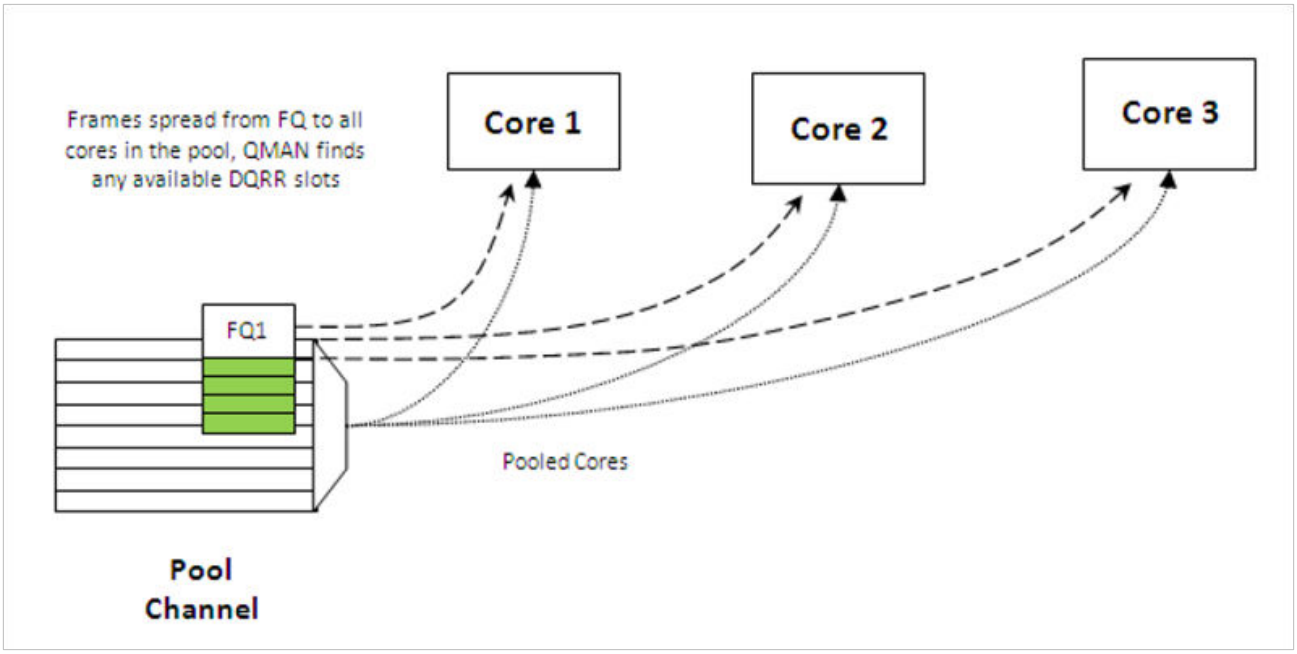
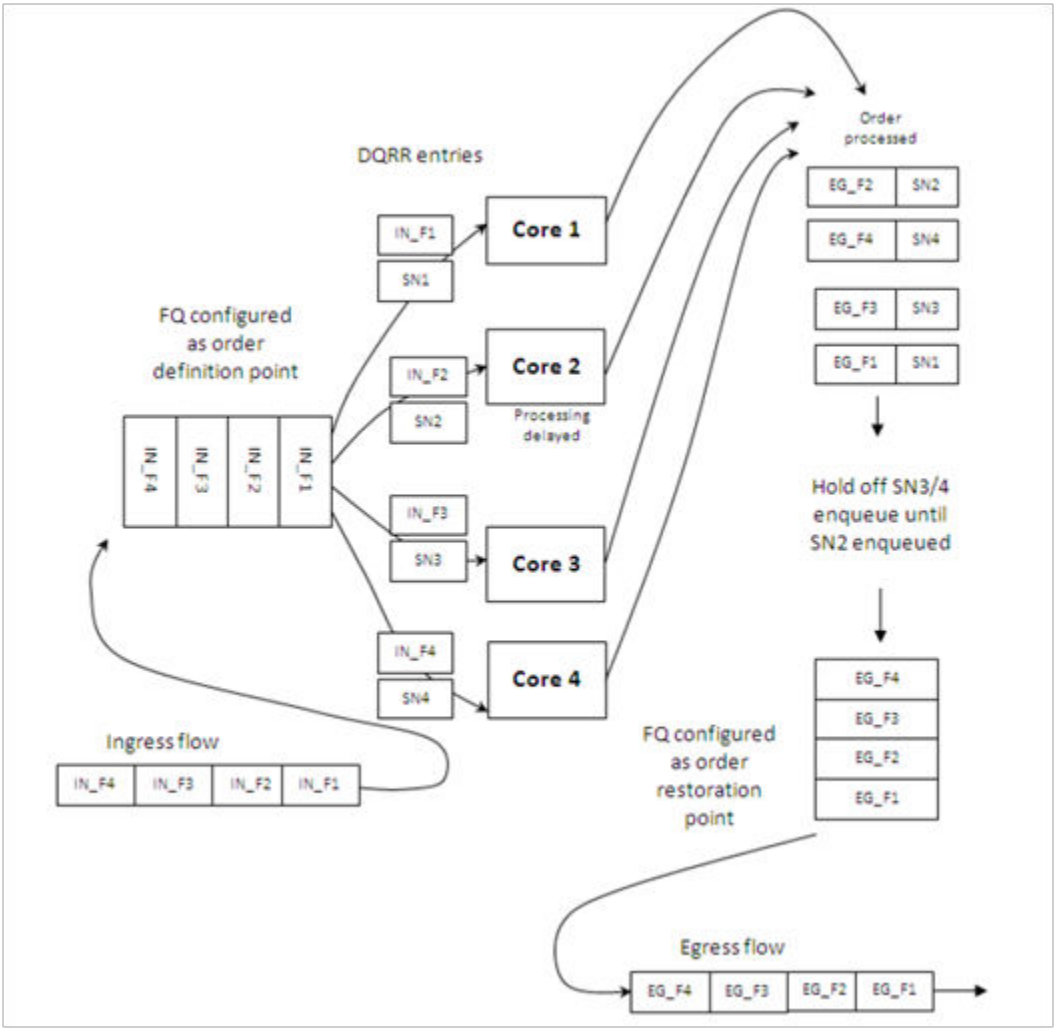
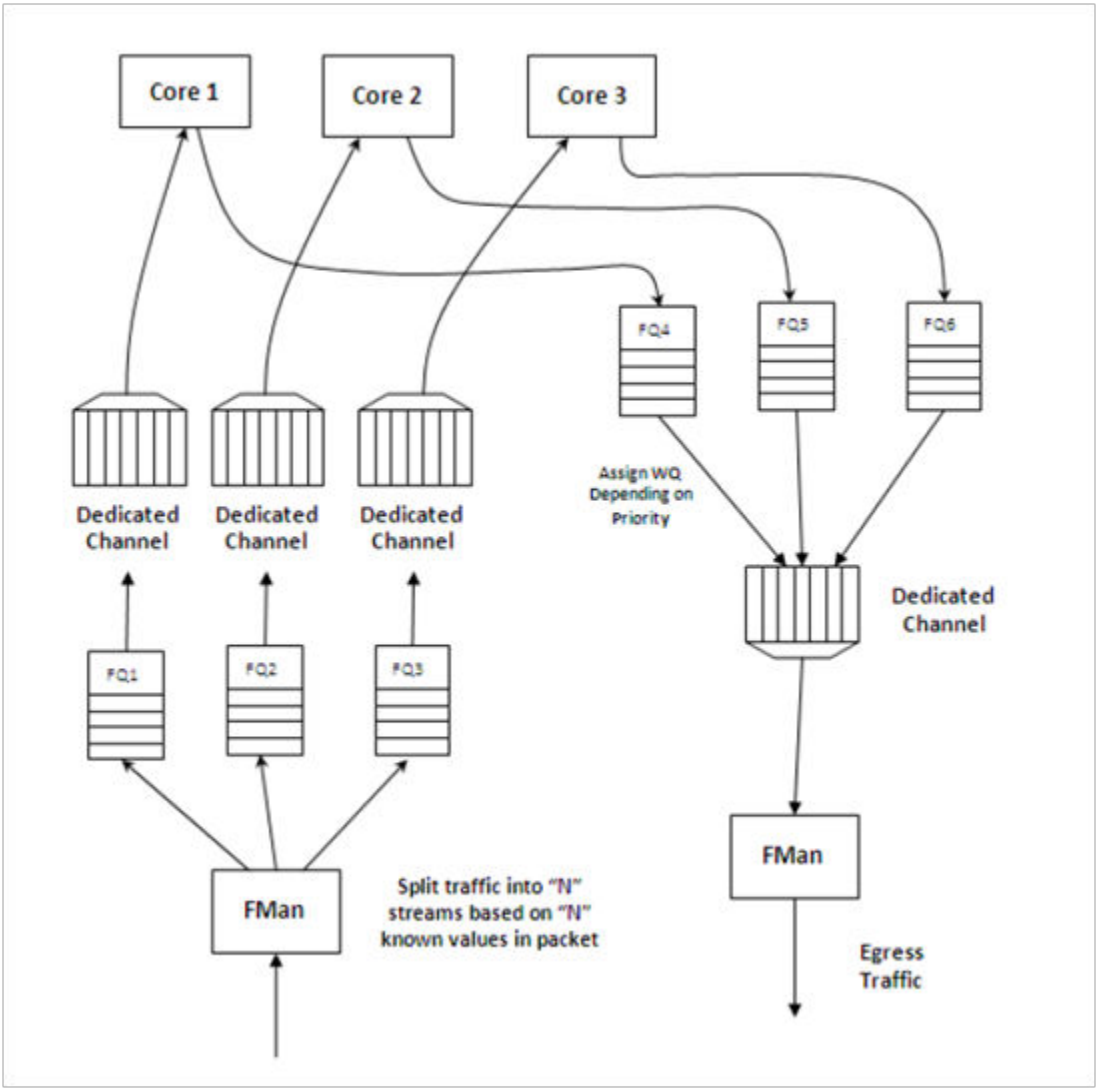
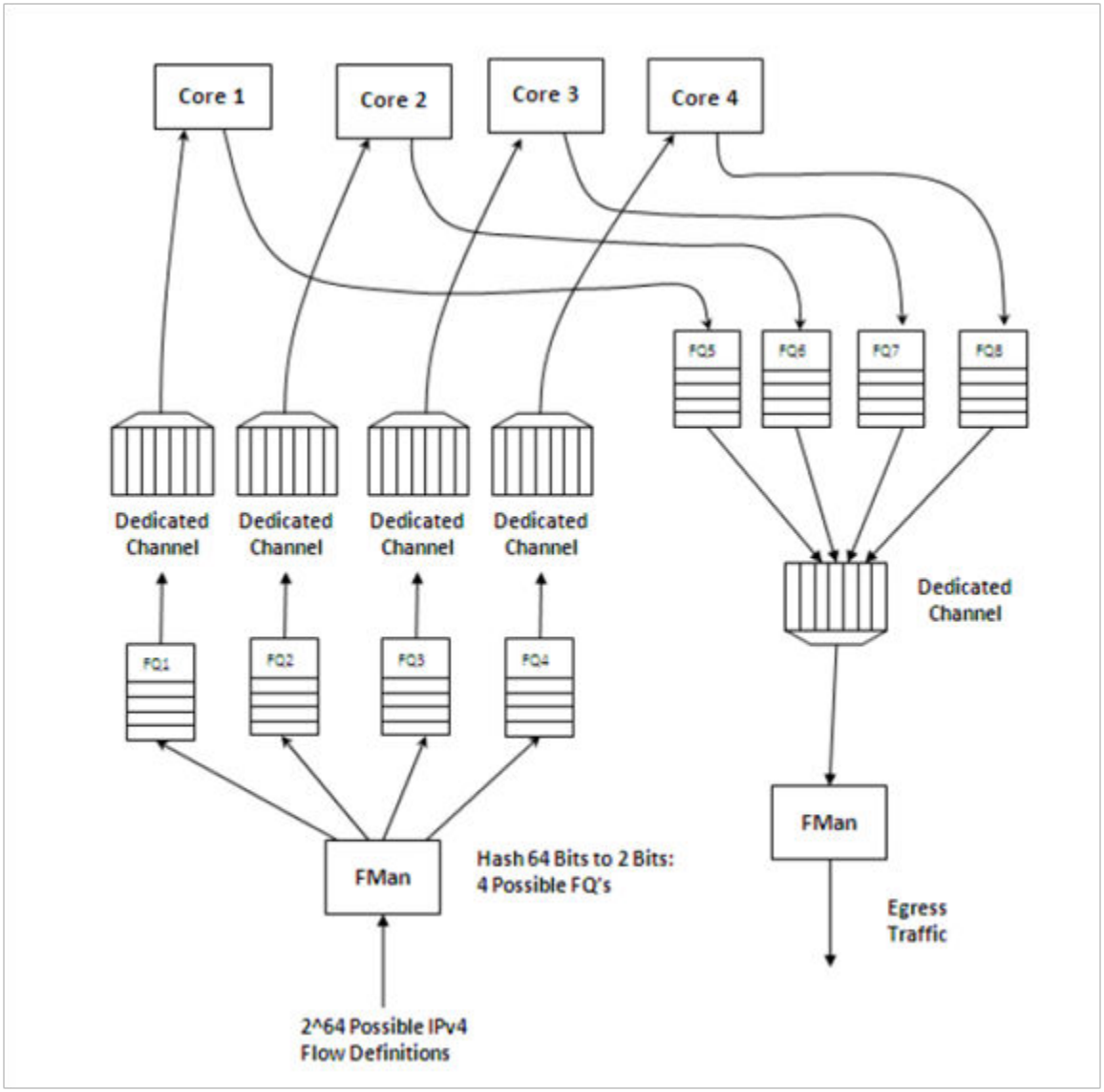
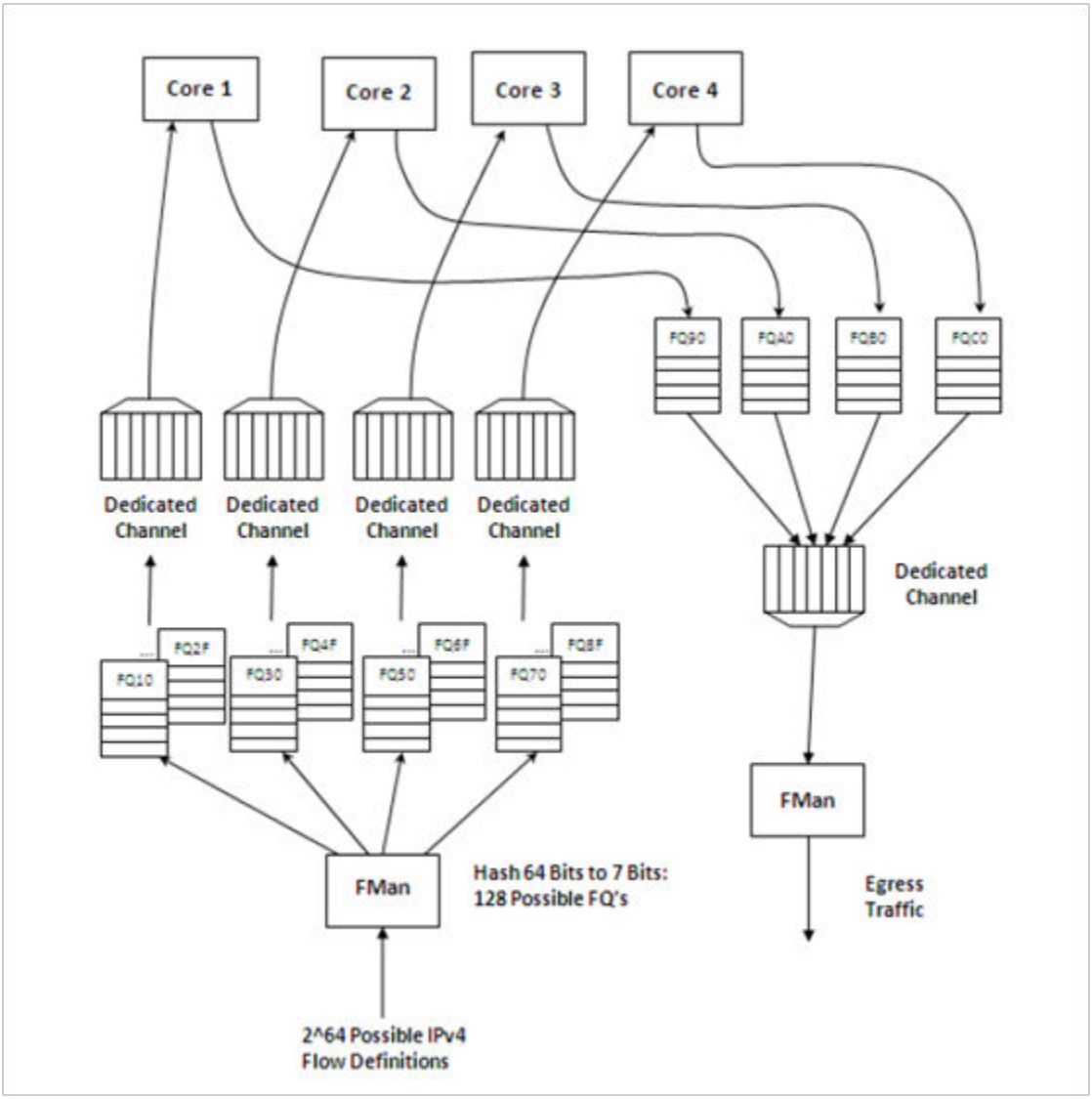
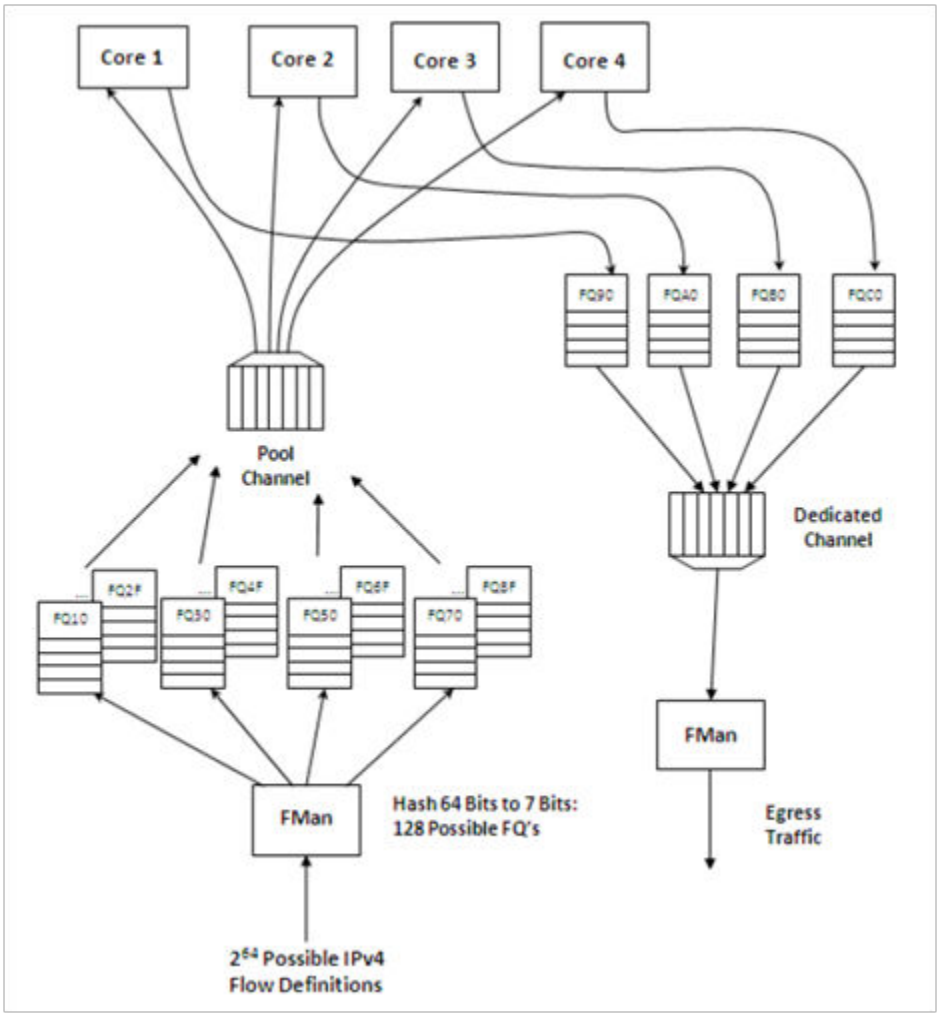
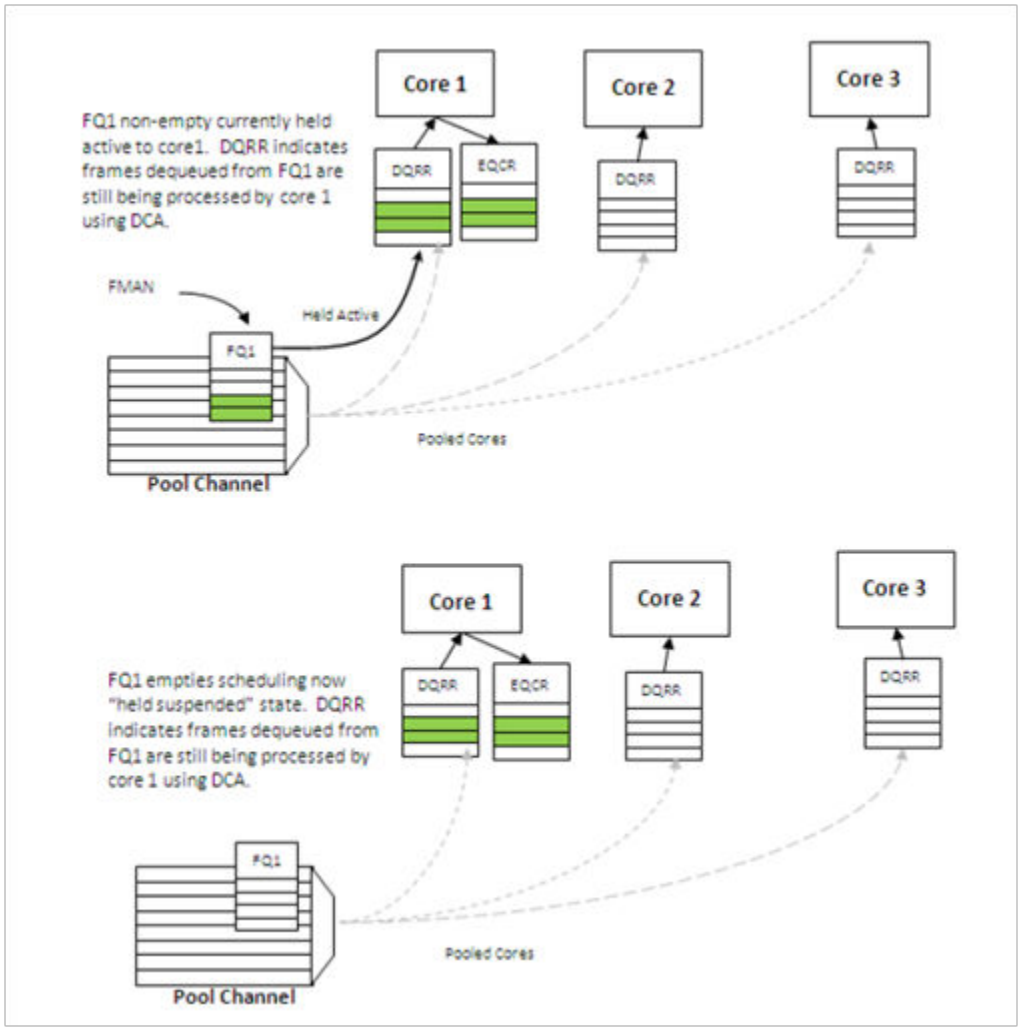
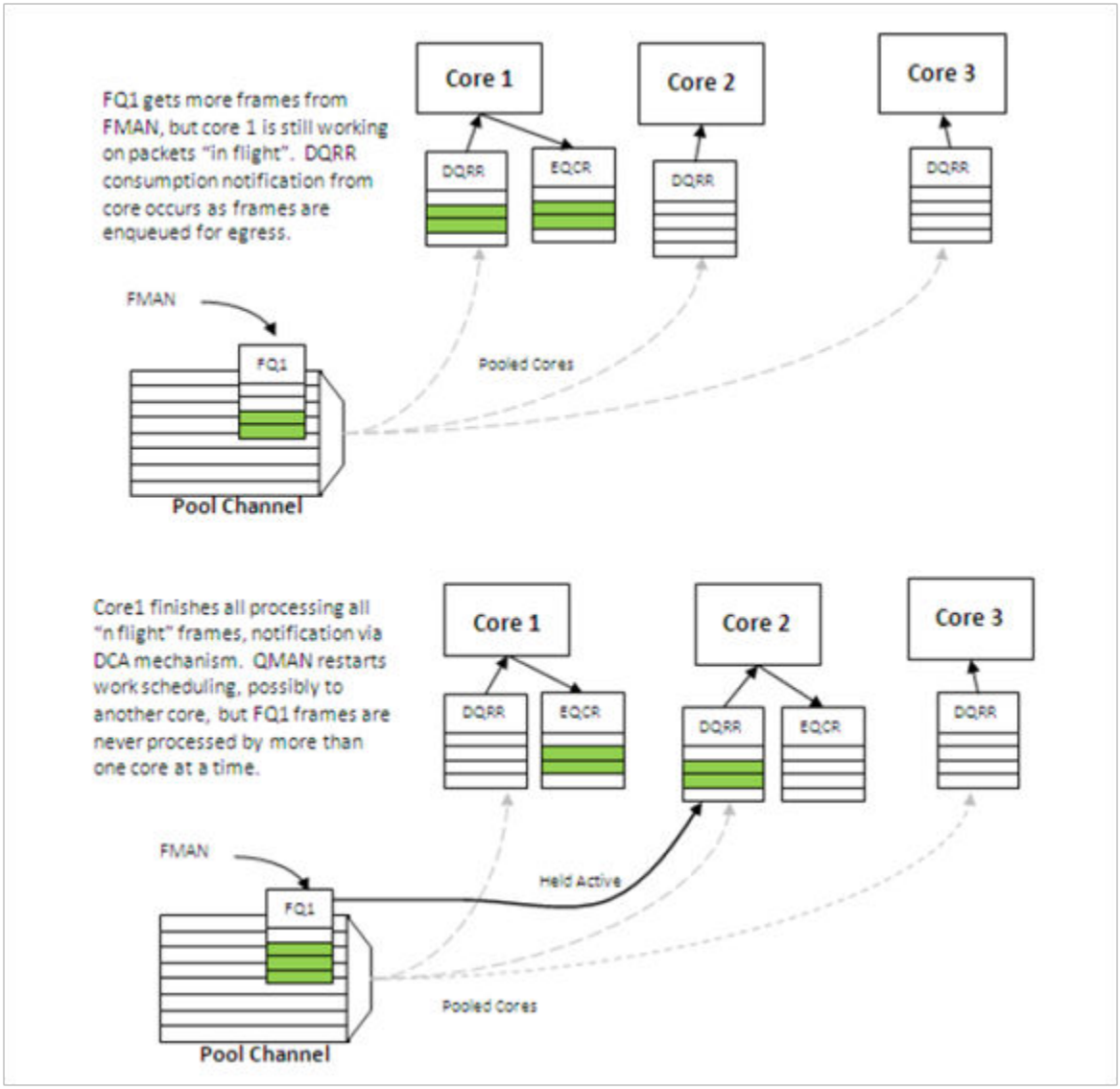
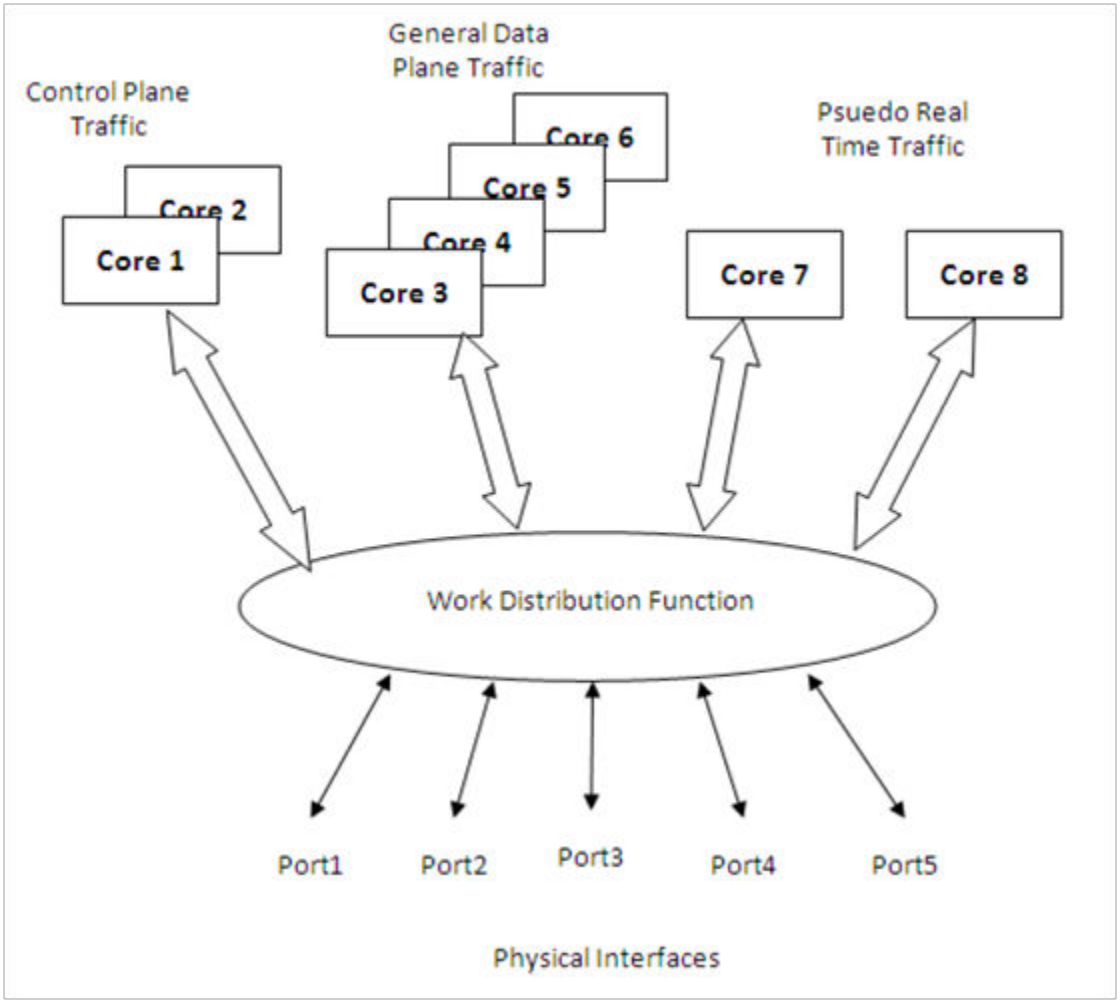
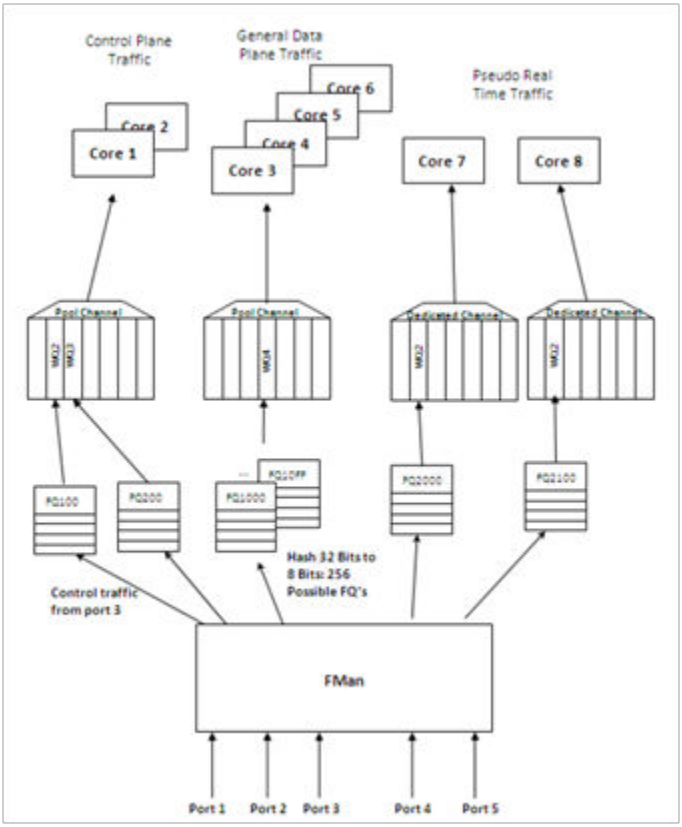
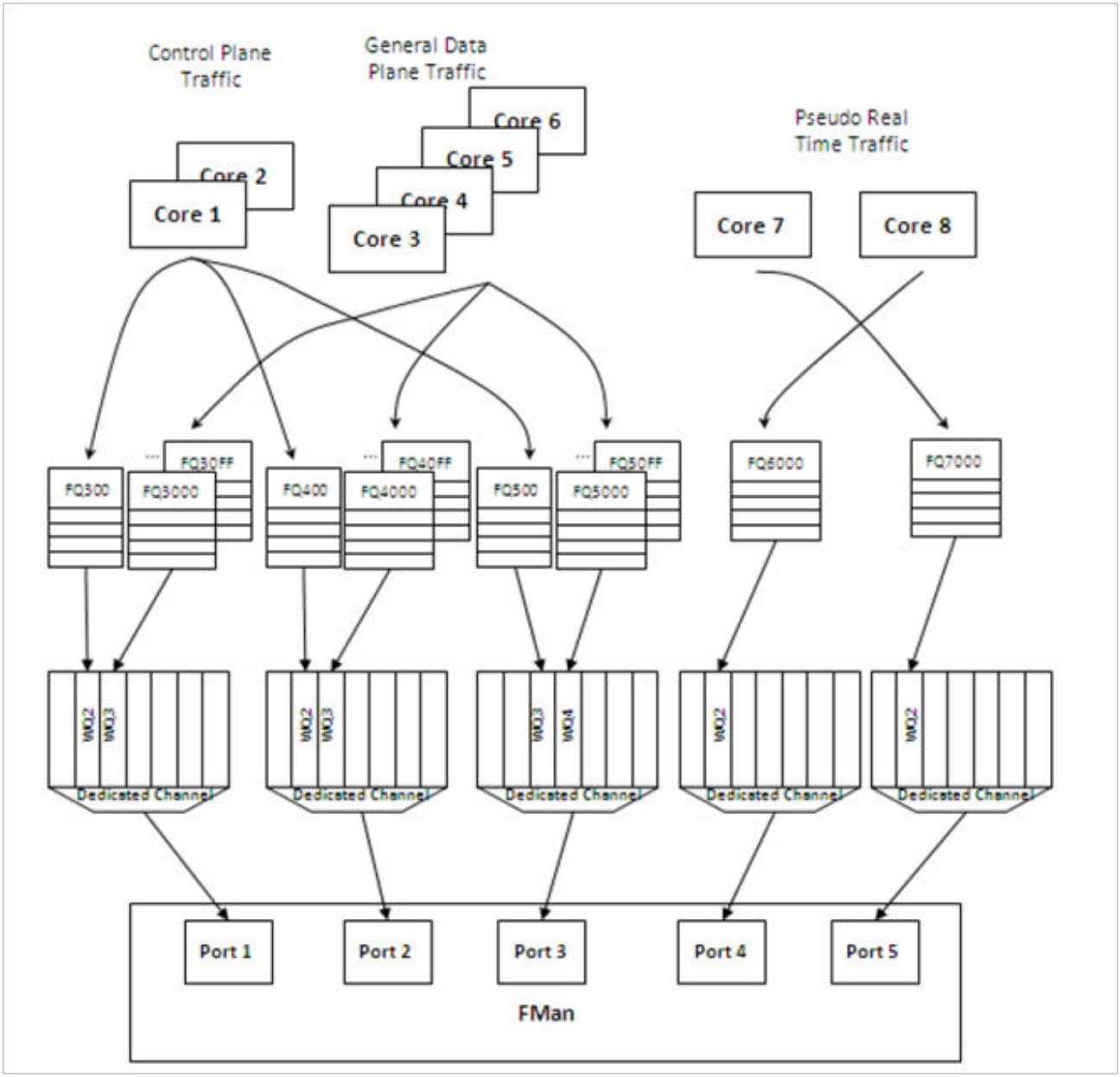
arecord -d 10 -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo-10s.wav
aplay -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo-10s.wav
```

5. Use alsamixer to adjust the volume for playing by the option “PCM” and recording gain by the option "Mic" . Use alsamixer to choose LINE IN or MIC.

7.2 Display Interface Unit (DIU)

7.2.1 DIU Display Device Driver User Manual

Description
This manual describes how to use the DIU.

Module Loading

The DIU device driver supports kernel built-in and module.

U-Boot Environment

In order to enable DIU after building the U-Boot image, define the below environment variable:

```
=>video-mode="fslfb:1024x768-32@60,monitor=dvi"
```

Testing at the U-Boot Level

1. Build U-Boot image with DIU and burn it to the NOR flash.
2. Build the U-Boot image with default option:
3. Make T1042D4RDB_defconfig

```
make
```

4. Setting the DIU output to DVI port: See the "U-Boot Environment" section of the SoC's Software Deployment Guide.
5. Connect to DVI port then power on. The U-Boot will look like the following:

```
U-Boot 2015.01-00582-g14d9d27 (Apr 08 2015 - 10:19:12)
CPU0: T1042E, Version: 1.1, (0x85280211)
Core: e5500, Version: 2.1, (0x80241021)
Clock Configuration:
CPU0:1200 MHz, CPU1:1200 MHz, CPU2:1200 MHz, CPU3:1200 MHz,
CCB:600 MHz,
DDR:800 MHz (1600 MT/s data rate) (Asynchronous), IFC:150 MHz
QE:300 MHz
FMAN1: 600 MHz
QMAN: 300 MHz
PME: 300 MHz
L1: D-cache 32 KiB enabled
I-cache 32 KiB enabled
Reset Configuration Word (RCW):
00000000: 0c18000c 0c000000 00000000 00000000
00000010: 86000002 40000002 ec027000 01000000
00000020: 00000000 00000000 00000000 00030810
00000030: 00000000 03fc5a0f 00000000 00000000
Board: T1042D4RDB
Board rev: 0x01 CPLD ver: 0x01, vBank: 0
                I2C: ready
SPI: ready
DRAM: Initializing....using SPD
Detected UDIMM 18ASF1G72AZ-2G1A1
6 GiB left unmapped
8 GiB (DDR4, 64-bit, CL=11, ECC on)
DDR Chip-Select Interleaving Mode: CS0+CS1
Flash: 256 MiB
L2: 256 KiB enabled
Corenet Platform Cache: 256 KiB enabled
Using SERDES1 Protocol: 134 (0x86)
SEC: RNG instantiated
NAND: 1024 MiB
MMC: FSL_SDHC: 0
QE microcode not found
PCIe1: Root Complex, no link, regs @ 0xfe240000
PCIe1: Bus 00 - 00
```

```
PCIe2: Root Complex, x1 gen1, regs @ 0xfe250000
02:00.0 - 8086:10d3 - Network controller
PCIe2: Bus 01 - 02
PCIe3: Root Complex, no link, regs @ 0xfe260000
PCIe3: Bus 03 - 03
PCIe4: Root Complex, no link, regs @ 0xfe270000
PCIe4: Bus 04 - 04
DIU: Switching to monitor DVI @ 1024x768
```

Kernel

Special DTS

```
display@180000 {
compatible = "fsl,t1040-diu", "fsl,diu";
reg = <0x180000 1000>;
interrupts = <74 2 0 0>;
};
```

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`  Graphics Support --->`
`  Frame buffer Devices --->`
`  < > Permedia3 support`
`  < > Fujitsu carmine frame buffer support`
`  <*> Freescale DIU framebuffer support`
`  < > Freescale DCU framebuffer support`
	FSL DIU display support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
`drivers/video/fbdev/ fsl-diu-fb.c` `include/linux/fsl-diu-fb.h`	FSL DIU framebuffer device driver

Testing at Kernel Level

1. Configure and rebuild the kernel as configuration list above.

```
Make corenet64_fmanv3l_smp_defconfig
Make
```

2. Boot up Linux kernel. The display relative to DIU will look like the following:

```
Freescale Display Interface Unit (DIU) framebuffer driver
graphics fb0: Panel0 registered successfully
graphics fb1: Panel1 AOI0 registered successfully
graphics fb2: Panel1 AOI1 registered successfully
```

```
graphics fb3: Panel2 AOI0 registered successfully
graphics fb4: Panel2 AOI1 registered successfully
```

Known Bugs, Limitations, or Technical Issues

All IPs need to go into idle when the system is going into deep sleep state. If the DIU driver is not enabled in the kernel, but is enabled in U-Boot, deep sleep will fail.

7.2.2 DCU Display Device Driver User Manual

Description

This manual describes how to use the Two Dimensional Animation and Compositing Engine (2D-ACE or DCU) and frame buffer on TWR-LS1021A board.

Module Loading

The DCU device driver supports kernel built-in and module.

U-Boot Configuration

Please use 'ls1021atwr_lpuart_config' to build the uboot.

Runtime options.

Env Variable	Description	Sub Option		Option Description
bootargs	Kernel command line argument passed to kernel	HDMI	console=ttyLP0,115200 hdmi	select LPUART0 as the system console
		LCD	console=ttyLP0,115200	

Kernel Configure Options

Tree View

Below are the Kernel Configure Tree View options need to be set/unset while doing "make menuconfig" for kernel and enable DCU/HDMI drivers and Linux Penguin Logo picture.

```
Device Drivers  --->

   < > Multimedia support  ----
       Graphics support  --->
           <*> Support for frame buffer devices  --->
               <*> Si Image SII9022 DVI/HDMI Interface Chip
               <*> Freescale DCU framebuffer support
                   …
           [ ] Exynos Video driver support  ----
           [ ] Backlight & LCD device support  ----
               Console display driver support  --->
                   <*> Framebuffer Console support
                   [*] Map the console to the primary display device
                   [*] Framebuffer Console Rotation
           [*] Bootup logo  --->
               --- Bootup logo
```

```
               [*] Standard black and white Linux logo
               [*] Standard 16-color Linux logo
               [*] Standard 224-color Linux logo
   < > Sound card support  ----
```

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Special Configure needs to be enabled ("Y") for LS1021A. Please find in below table with default value as "N"

Option	Values	Default Value	Description
CONFIG_ FB_FSL_SII902X	y/m/n	y	Si Image SII9022 DVI/HDMI Interface Chip
CONFIG_FB_FSL_DCU	y/m/n	y	NXP DCU framebuffer supportt
CONFIG_LOGO	y/m/n	y	Bootup logo
CONFIG_LOGO_LINUX_MONO	y/m/n	y	Standard black and white Linux logo
CONFIG_LOGO_LINUX_VGA16	y/m/n	y	Standard 16-color Linux logo
CONFIG_LOGO_LINUX_CLUT224	y/m/n	y	Standard 224-color Linux logo
CONFIG_FRAMEBUFFER_CONSOLE	y/m/n	y	Framebuffer Console support

Device Tree Binding

Special Configure needs to be enabled ("Y") for LS1021A. Please find in below table with default value as "N"

arch/arm/boot/dts/ls1021a.dtsi

```
dcu0: dcu@2ce0000 {
            compatible = "fsl,vf610-dcu";
             reg = <0x0 0x2ce0000 0x0 0x10000>;
             interrupts = <GIC_SPI 172 IRQ_TYPE_LEVEL_HIGH>;
             clocks = <&platform_clk 0>;
             clock-names = "dcu";
             scfg-controller = <&scfg>;
              big-endian;
              status = "disabled";
};
```

arch/arm/boot/dts/ls1021a-twr.dts

```
&dcu0 {
        display = <&display>;
        status = "okay";

        display: display@0 {
                bits-per-pixel = <24>;

                display-timings {
                        native-mode = <&timing0>;
                        timing0: nl4827hc19 {
                                clock-frequency = <10870000>;
                                hactive = <480>;
                                vactive = <272>;
                                hback-porch = <2>;
                                hfront-porch = <2>;
                                vback-porch = <2>;
```

```
                                vfront-porch = <2>;
                                hsync-len = <41>;
                                vsync-len = <4>;
                                hsync-active = <1>;
                                vsync-active = <1>;
                        };
                };
        };
};
```

Ramdisk:

Please use the 'fsl-image-x11-ls1021a(XXXXX)rootfs.ext2.gz.gz' ramdisk from each release images, or you can just use the ramdisk image which has 'x11' label.

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/video/fsl-dcu-fb.c	NXP DCU driver

Testing LCD/DHMI at Uboot Level

1. Display with LCD:

```
=> setenv video-mode  "fslfb:480x272-32@60,monitor=twr_lcd"
=> save
=> setenv stdout vga
```

2. Display with HDMI:

```
=> setenv video-mode  "fslfb:640x480-32@60,monitor=hdmi"
=> save
=> setenv stdout vga
```

Testing LCD at Kernel Level

1. Configure and rebuild the kernel as configuration list above, let the DCU driver built into the Kernel Image.
2. Boot up Linux kernel, upon the kernel has been uncompressed, the TFT Panel will display the Linux Penguin Logo.
3. And then after the root filesystem has been monted, and the Xwindows Desktop will be display.
4. Or also you can start the Xwindow using:

```
root@ls1021atwr:~# killall matchbox-window-manager
root@ls1021atwr:~# xinit /etc/init.d/xserver-nodm restart
```

5. Just plug out and plug in the HDMI to test the hot plug.

Testing HDMI at Kernel Level

1. 1.Configure and rebuild the kernel as configuration list above, let the HDMI and DCU drivers built into the Kernel Image.
2. Boot up Linux kernel, upon the kernel has been uncompressed, the TFT Panel won't display any picture correctly.
3. And then after the root filesystem has been monted, and the Xwindows Desktop will be displayed on the HDMI Monitor.

4. Or also you can start the Xwindow using:

```
root@ls1021atwr:~# killall matchbox-window-manager
root@ls1021atwr:~# xinit /etc/init.d/xserver-nodm restart

Note: Please unplugged the TWR-LDC_RGB daughter board when testing the
HDMI.
```

Known Bugs, Limitations, or Technical Issues

Unplug the SD card before testing the DCU/HDMI, or the system will hang.

7.3 DMA Controller

7.3.1 Direct Memory Access Driver (PowerQUICC, QorIQ)

Description

NXP Platform DMA Controller are integrated with the MPC85xx and QorIQ cpu.

Module Loading

The MPC8xxx DMA support code is compiled into the kernel.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
Device Drivers ---> DMA Engine supportt ---> <*> Freescale Elo and Elo Plus DMA support [*] Network: TCP receive copy offload <*> DMA Test client	NXP Platform DMA support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_FSL_DMA	y/n	y	Enable NXP Platform DMA
CONFIG_DMA_ENGINE	y/n	y	Enable DMA support
CONFIG_NET_DMA	y/n	y	Enable network DMA offload
CONFIG_DMATEST	y/m/n	n	Enable DMA self test

NOTE

CONFIG_DMA_TEST is only for test usage.

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/dma/fsldma.c	NXP plaform DMA code
drivers/dma/dmatest.c	DMA test code

Test Procedure

- Enable CONFIG_DMA_TEST option when build kernel.

This kind of informaiton will appear in the kernel boot up stage

```
...


fsl-elo-dma fe100300.dma: #0 (fsl,eloplus-dma-channel), irq 28
fsl-elo-dma fe100300.dma: #1 (fsl,eloplus-dma-channel), irq 29
fsl-elo-dma fe100300.dma: #2 (fsl,eloplus-dma-channel), irq 30
fsl-elo-dma fe100300.dma: #3 (fsl,eloplus-dma-channel), irq 31
fsl-elo-dma fe101300.dma: Probe the Freescale DMA driver for fsl,eloplus-dma con
troller at 0xfe101300...
fsl-elo-dma fe101300.dma: #0 (fsl,eloplus-dma-channel), irq 32
fsl-elo-dma fe101300.dma: #1 (fsl,eloplus-dma-channel), irq 33
fsl-elo-dma fe101300.dma: #2 (fsl,eloplus-dma-channel), irq 34
fsl-elo-dma fe101300.dma: #3 (fsl,eloplus-dma-channel), irq 35
audit: initializing netlink socket (disabled)
...
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
dmatest: Started 1 threads using dma0chan0
dmatest: Started 1 threads using dma0chan1
dmatest: Started 1 threads using dma0chan2
dmatest: Started 1 threads using dma0chan3
dmatest: Started 1 threads using dma1chan0
dmatest: Started 1 threads using dma1chan1
dmatest: Started 1 threads using dma1chan2
dmatest: Started 1 threads using dma1chan3
RAMDISK: gzip image found at block 0
VFS: Mounted root (ext2 filesystem) readonly on device 1:0.
Freeing unused kernel memory: 204k init
Mounting /proc and /sys
Starting the hotplug events dispatcher ud
...
```

Known Bugs, Limitations, or Technical Issues

- Only support scenenarios without hypervisor

7.3.2 Enhanced Direct Memory Access Driver (ARM)

7.3.2.1 eDMA User Manual

Description

The SoC integrates NXP's Enhanced Direct Memory Access module. Slave device such as I2C or SAI can deploy the DMA functionality to accelerate the transfer and release the CPU from heavy load.

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree View Options	Description
Device Drivers ---> [*] DMA Engine support ---> --->*> <*> Freescale eDMA engine support	DMA engine subsystem driver and eDMA driver support

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_FSL_EDMA	y/m/n	n	eDMA Driver

Device Tree Binding

Device Tree Node

Below is an example device tree node required by this feature. Note that it may has differences among platforms.

```
edma0: edma@2c00000 {
                        #dma-cells = <2>;
                        compatible = "fsl,vf610-edma";
                        reg = <0x0 0x2c00000 0x0 0x10000>,
                                <0x0 0x2c10000 0x0 0x10000>,
                                <0x0 0x2c20000 0x0 0x10000>;
                        interrupts = <GIC_SPI 135 IRQ_TYPE_LEVEL_HIGH>,
                                        <GIC_SPI 135 IRQ_TYPE_LEVEL_HIGH>;
                        interrupt-names = "edma-tx", "edma-err";
                        dma-channels = <32>;
                        big-endian;
                        clock-names = "dmamux0", "dmamux1";
                        clocks = <&platform_clk 1>,
                                <&platform_clk 1>;
                };
```

Device Tree Node Binding for Slave Device

Below is the device tree node binding for a slave device which deploy the eDMA functionality.

```
i2c0: i2c@2180000 {
```

```
                        #address-cells = <1>;
                        #size-cells = <0>;
                        compatible = "fsl,vf610-i2c";
                        reg = <0x0 0x2180000 0x0 0x10000>;
                        interrupts = <GIC_SPI 88 IRQ_TYPE_LEVEL_HIGH>;
                        clock-names = "i2c";
                        clocks = <&platform_clk 1>;
                        dmas = <&edma0 1 39>,
                               <&edma0 1 38>;
                        dma-names = "tx", "rx";
                        status = "disabled";
                };
```

Source Files

The following source files are related the this feature in Linux kernel.

Table 119. Source Files

Source File	Description
drivers/dma/fsl-edma.c	The eDMA driver file

Verification in Linux

1. Use the slave device which deploy the eDMA functionality to verify the eDMA driver, below is a verification with the I2C salve.

```
root@ls1021aqds:~# i2cdetect 0
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0.
I will probe address range 0x03-0x77.
Continue? [Y/n]
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- 69 -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
root@ls1021aqds:~# i2cdump 0 0x69 i
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 05 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff    ??..]U?U???.???.
10: ff e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78    .???....???....x
20: 05 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00    ???..?@??`<??.@.
30: fe 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff    ???)...z........
40: 05 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff    ??..]U?U???.???.
50: ff e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78    .???....???....x
60: 05 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00    ???..?@??`<??.@.
70: fe 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff    ???)...z........
80: 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff ff    ?..]U?U???.???..
90: e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78 00    ???....???....x.
a0: 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00 fe    ??..?@??`<??.@.?
b0: 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff ff    ??)...z.........
c0: 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff ff    ?..]U?U???.???..
d0: e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78 00    ???....???....x.
```

```
e0: 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00 fe    ??..?@??`<??.@.?
f0: 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff ff    ??)...z.........
root@ls1021aqds:~# cat /proc/interrupts
           CPU0       CPU1
 29:          0          0       GIC  29  arch_timer
 30:       5563       5567       GIC  30  arch_timer
112:        260          0       GIC 112  fsl-lpuart
120:         32          0       GIC 120  2180000.i2c
121:          0          0       GIC 121  2190000.i2c
167:          8          0       GIC 167  eDMA
IPI0:         0          1  CPU wakeup interrupts
IPI1:         0          0  Timer broadcast interrupts
IPI2:      1388       1653  Rescheduling interrupts
IPI3:         0          0  Function call interrupts
IPI4:         2          4  Single function call interrupts
IPI5:         0          0  CPU stop interrupts
Err:          0
root@ls1021aqds:~#
```

7.4 DPAA 1.x Devices

7.4.1 DPAA Primer for Software Architecture

7.4.1.1 DPAA Primer

Multicore processing, or multiple execution thread processing, introduces unique considerations to the architecture of networking systems, including processor load balancing/utilization, flow order maintenance, and efficient cache utilization. Herein is a review of the key features, functions, and properties enabled by the QorIQ DPAA (Data Path Acceleration Architecture) hardware and demonstrates how to best architect software to leverage the DPAA hardware.

By exploring how the DPAA is configured and leveraged in a particular application, the user can determine which elements and features to use. This streamlines the software development stage of implementation by allowing programmers to focus their technical understanding on the elements and features that are implemented in the system under development, thereby reducing the overall DPAA learning curve required to implement the application.

Our target audience is familiar with the material in **QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual**.

Benefits of the DPAA

The primary intent of the DPAA is to provide intelligence within the IO portion of the System On Chip (SOC) to route and manage the processing work associated with traffic flows to simplify ordering and load balance concerns associated with multi core processing. The DPAA hardware inspects ingress traffic and extracts user defined flows from the port traffic. It then steers specific flows (or related traffic) to a specific core or set of cores.

Architecting a networking application with a multicore processor presents challenges (such as workload balance and maintaining flow order), which are not present in a single core design. Without hardware assistance, the software must implement techniques that are inefficient and cumbersome, reducing the performance benefit of multiple cores. To address workload balance and flow order challenges, the DPAA determines and separates ingress flows then manages the temporary, permanent, or semi-permanent flow-to-core affinity. The DPAA also provides a work priority scheme, which ensures ingress critical flows are addressed properly and frees software from the need to implement a queuing mechanism on egress. As the hardware determines which core will process which packet, performance is boosted by direct cache warming/stashing as well as by providing biasing for core-to-flow affinity, which ensures that flow-specific data structures can remain in the core's cache.

By understanding how the DPAA is leveraged in a particular design, the system architect can map out the application to meet the performance goals and to utilize the DPAA features to leverage any legacy application software that may exist. Once this application map is defined, the architect can focus on more specific details of the implementation.

7.4.1.1.1 General Architectural Considerations

As the need for processing capability has grown, the only practical way to increase the performance on a single silicon part is to increase the number of general purpose processing cores (CPUs). However, many legacy designs run on a single processor; introducing multiple processors into the system creates special considerations, especially for a networking application.

7.4.1.1.2 Multicore Design

Multicore processing, or multiple execution thread processing, introduces unique considerations. Most networking applications are split between data and control plane tasks. In general, control plane tasks manage the system within the broad network of equipment. While the control plane may process control packets between systems, the control plane process is not involved in the bulk processing of the data traffic. This task is left to the data plane processing task or program.

The general flow of the data plane program is to receive data traffic (in general, packets or frames), process or transform the data in some way and then send the packets to the next hop or device in the network. In many cases, the processing of the traffic depends on the type of traffic. In addition, the traffic usually exists in terms of a flow, a stream of traffic where the packets are related. A simple example could be a connection between two clients that, at the packet level, is defined by the source and destination IP address. Typically, multiple flows are interleaved on a single interface port; the number of flows per port depends on the interface bandwidth as well as on the bandwidth and type of flows involved.

7.4.1.1.3 Parse/classification Software Offload

The DPAA provides intelligence within the IO subsection of the SoC (system-on-chip) to split traffic into user-defined queues. One benefit is that the intelligence used to divide the traffic can be leveraged at a system level.

In addition to sorting and separating the traffic, the DPAA can append useful processing information into the stream; offloading the need for the software to perform these functions (see the following two figures).

Note that the DPAA is not intended to replace significant packet processing or to perform extensive classification tasks. However, some applications may benefit from the streamlining that results from the parse/classify/distribute function within the DPAA. The ability to identify and separate flow traffic is fundamental to how the DPAA solves other multicore application issues.



Figure 63. Hardware-sorted Protocol Flow

7.4.1.1.4 Flow Order Considerations

In most networking applications, individual traffic flows through the system require that the egress packets remain in the order in which they are received. In a single processor core system, this requirement is easy to implement. As long as the data plane software follows a run-to-completion model on a per-packet basis, the egress order will match the ingress packet order. However, if multiple processors are implemented in a true symmetrical multicore processing (SMP) model in the system, the egress packet flow may be out of order with respect to the ingress flow. This may be caused when multiple cores simultaneously process packets from the same flow.

Even if the code flow is identical, factors such as cache hits/misses, DRAM page hits/misses, interrupts, control plane and OS tasks can cause some variability in the processing path, allowing egress packets to “pass” within the same flow, as shown in this figure



Figure 64. Multicore Flow Reordering

For some applications, it is acceptable to reorder the flows from ingress to egress. However, most applications require that order is maintained. When no hardware is available to assist with this ordering, the software must maintain flow order. Typically, this requires additional code to determine the sequence of the packet currently being processed, as well as a reference to a data structure that maintains order information for each flow in the system. Because multiple processors need to access and update this state information, access to this structure must be carefully controlled, typically by using a lock mechanism that can be expensive with regard to program cycles and processing flow (see the next figure). One of goals of the DPAA architecture is to provide the system designer with hardware to assist with packet ordering issues.



Figure 65. Implementing Order in Software

7.4.1.1.5 Managing Flow-to-Core Affinity

Multicore processing, or multiple execution thread processing, introduces unique considerations to the architecture of networking systems, including processor load balancing/utilization, flow order maintenance, and efficient cache utilization. Herein is a review of the key features, functions, and properties enabled by the QorIQ DPAA (Data Path Acceleration Architecture) hardware and demonstrates how to best architect software to leverage the DPAA hardware.

In a multicore networking system, if the hardware configuration always allows a specific core to process a specific flow then the binding of the flow to the core is described as providing flow affinity. If a specific flow is always processed by a specific processor core then for that flow the system acts like a single core system. In this case, flow order is preserved because there is a single thread of operation processing the flow; with a run-to-completion model, there is no opportunity for egress packets to be reordered with respect to ingress packets.

Another benefit of a specific flow being affined to a core is that the cache local to that core (L1 and possibly L2, depending on the specific core type) is less likely to miss when processing the packets because the core's data cache will not fetch flow state information for flows to which it is not affined. Also, because multiple processing entities have no need to access the same individual flow state information, the system need not lock the access to the individual flow state data. The DPAA offers several options to define and manage flow-to-core affinity.



Figure 66. Managing Flow-to-Core Affinity

Many networking applications require intensive, repetitive algorithms to be performed on large portions of the data stream(s). While software in the processor cores could perform these algorithms, specific hardware offload engines often better address specific algorithms. Cryptographic and pattern matching accelerators are examples of this in the QorIQ family. These accelerators act as standalone hardware elements that are fed blocks or streams of data, perform the required processing, and then provide the output in a separate (or perhaps overwritten) data block within the system. The performance boost is significant for tasks that can be done by these hardware accelerators as compared to a software implementation.

In DPAA-equipped SoCs, these offload engines exist as peers to the cores and IO elements, and they use the same queuing mechanism to obtain and transfer data. The details of the specific processing performed by these offload engines is beyond the scope of this document; however, it is important to determine which of these engines will be leveraged in the specific application. The DPAA can then be properly defined to implement the most efficient configuration/definition of the DPAA elements.

7.4.1.2 DPAA Goals

A brief overview of the DPAA elements in order to contextualize the application mapping activities. For more details on the DPAA architecture, see the **QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual**

The primary goals of the DPAA are as follows:

- To provide intelligence within the IO portion of the SoC.
- To route and manage the processing work associated with traffic flows.

- To simplify the ordering and load balance concerns associated with multicore processing.

The DPAA achieves these goals by inspecting and separating ingress traffic into Frame Queues (FQs). In general, the intent is to define a flow or set of flows as the traffic in a particular FQ. The FQs are associated to a specific core or set of cores via a channel. Within the channel definition, the FQs can be prioritized among each other using the Work Queue (WQ) mechanism. The egress flow is similar to the ingress flow. The processors place traffic on a specific FQ, which is associated to a particular physical port via a channel. The same priority scheme using WQs exists on egress, allowing higher priority traffic to pass lower priority traffic on egress without software intervention.

7.4.1.3 FMan Overview

The FMan inspects traffic, splits it into FQs on ingress, and sends traffic from the FQs to the interface on egress.

On ingress traffic, the FMan is configured to determine the traffic split required by the PCD (Parse, Classify, Distribute) function. This allows the user to decide how he wants to define his traffic: typically, by flows or classes of traffic. The PCD can be configured to route all traffic on one port to a single queue or with a higher level of complexity where large numbers of queues are defined and managed. The PCD can identify traffic based on the specific content of the incoming packets (usually within the header) or packet reception rates (policing).

The parse function is used to identify which fields within the data frame determine the traffic split. The fields used may be defined by industry standards, or the user may employ a programmable soft parse feature to accommodate proprietary field (typically header) definition(s). The results of the parse function may be used directly to determine the frame queue; or, the contents of the fields selected by the parse function may be used as a key to select the frame queue. The parse function employs a programmed mask to allow the use of selected fields.

The resultant key from the parse function may be used to assign traffic to a specific queue based on a specific exact match definition of fields in the header. Alternatively, a range of queues can be defined either by using the resultant key directly (if there are a small number of bits in the key) or by performing a hash of the key to use a large number of bits in the flow identifier and create a manageable number of queues.

The FMan also provides a policer function, which is rate-based and allows the user to mark or drop a specific frame that exceeds a traffic threshold. The policing is based on a two-rate, three-color marking algorithm (RFC2698). The sustained and peak rates as well as the burst sizes are user-configurable.



Figure 67. Ingress FMan Flow

The figure above shows the FMan splitting ingress traffic on an external port into a number of queues. However, the FMan works in a similar way on egress: it receives traffic from FQs then transmits the traffic on the designated external port. Alternatively, the FMan can be used to process flows internally via the offline port mechanism: traffic is dequeued (from some other element in the system), processed, then enqueued onto a frame queue processing further within the system.

On ingress traffic, the FMan generates an internal context (IC) data block, which it uses as it performs the PCD function. Optionally, some or all of this information may be added into the frames as they are passed along for further processing. For egress or offline processing, the IC data can be passed with each frame to be processed. This data is mostly the result of the PCD actions and includes the results of the parser, which may be useful for the application software. See the QorIQ DPAA Reference Manual for details on the contents of the IC data block.



Figure 68. FMan Egress Flow



Figure 69. FMan Offline Flow

7.4.1.4 QMan Overview

The QMan links the FQs to producers and consumers (of data traffic) within the SoC. The producers/consumers are either FMan, acceleration blocks, or CPU cores.

All the producers/consumers have one channel, each of which is referred to as a dedicated channel. Additionally, there are a number of pool channels available to allow multiple cores (not FMan or accelerators) to service the same channel. Note that there are channels for each external FMan port, the number of which depends on the SoC, as well as the internal offline ports.



Figure 70. DPAA Channel Types

Each channel provides for eight levels of priority, each of which has its own work queue (WQ). The two highest level WQs are strict priority: traffic from WQ0 must be drained before any other traffic flows; then traffic from WQ1 and then traffic from the other six WQs is allowed to pass. The last six WQs are grouped together in two groups of three, which are configurable in a weighted round robin fashion. Most applications do not need to use all priority levels. When multiple FQs are assigned to the same WQ, QMan implements a credit-based scheme to determine which FQ is scheduled (providing frames to be processed) and how many frames (actually the credit is defined by the number of bytes in the frames) it can dequeue before QMan switches the scheduling to the next FQ on the WQ. If a higher priority WQ becomes active (that is, one of the FQs in the higher priority WQ receives a frame to become non-empty) then the dequeue from the lower priority FQ is suspended until the higher priority frames are dequeued. After the higher priority FQ is serviced, when the lower priority FQ restarts servicing, it does so with the remaining credit it had before being pre-empted by the higher priority FQ.

When the DPAA elements of the SoC are initialized, the FQs are associated with WQs, allowing the traffic to be steered to the desired core (dedicated connect channel), set of cores (pool channel), FMan, or accelerator, using a defined priority.



Figure 71. Prioritizing Work

QMan: Portals

A single portal exists for each non-core DPAA producer/consumer (FMan, SEC, and PME). This is a data structure internal to the SoC that passes data directly to/from the consumer's direct connect channel.

Software portals are associated with the processor cores and are, effectively, data structures that the cores use to pass (enqueue) packets to or receive (dequeue) packets from the channels associated with that portal (core). Each SoC has at least as many software portals as there are cores. Software portals are the interface through which the DPAA provides the data processing workload for a single thread of execution.

The portal structure consists of the following:

- The Dequeue Response Ring (DQRR) determines the next packet to be processed.
- The Enqueue Command Ring (EQCR) sends packets from the core to the other elements.
- The Message Ring (MR) notifies the core of the action (for example, attempted dequeue rejected, and so on).

- The Management command and response control registers.



Figure 72. Processor Core Portal

On ingress, the DQRR acts as a small buffer of incoming packets to a particular core. When a section of software performs a get packet type operation, it gets the packet from a pointer provided as an entry in the DQRR for the specific core running that software. Note that the DQRR consolidates all potential channels that may be feeding frames to a particular core. There are up to 16 entries in each DQRR. Each DQRR entry contains:

- a pointer to the packet to be processed,
- an identifier of the frame queue from which the packet originated,
- a sequence number (when configured),
- and additional FMan-determined data (when configured).

When configured for push mode, QMan attempts to fill the DQRR from all the potential incoming channels. When configured in pull mode, QMan only adds one DQRR entry when it is told to by the requesting core. Pull mode may be useful in cases where the traffic flows must be very tightly controlled; however, push mode is normally considered the preferred mode for most applications.



Figure 73. Ingress Channel to Portal Options

The DQRRs are tightly coupled to a processor core. The DPAA implements a feature that allows the DQRR mechanism to pre-allocate, or stash, the L1 and/or L2 cache with data related to the packet to be processed by that core. The intent is to have the data required for packet processing in the cache before the processor runs the “get packet” routine, thereby reducing the overall time spent processing a particular packet.

The following is data that may be warmed into the caches:

- The DQRR entry
- The packet or portion of the packet for a single buffer packet
- The scatter gather list for a multi-buffer packet
- Additional data added by FMan
- FQ context (A and B)

The FQ context is a user-defined space in memory that contains data associated with the FQ (per flow) to be processed. The intent is to place in this data area the state information required when processing a packet for this flow. The FQ context is part of the FQ definition, which is performed when the FQ is initialized.

The cache warming feature is enabled by configuring the capability and some definition of the FQs and QMan at system initialization time. This can provide a significant performance boost and requires little to no change in the processing flow.

When defining the system architecture, it is highly recommended that the user enable this feature and consider how to maximize its impact.



Figure 74. Cache Warming Options

In addition to getting packet information from the DQRR, the software also manages the DQRR by indicating which DQRR entry it will consume next. This is how the QMan determines when the DQRR (portal) is ready to process more frames. Two basic options are provided. In the first option, the software can update the ring pointer after one or several entries have been consumed. By waiting to indicate the consumption of multiple frames, the performance impact of the write doing this is minimized. The second option is to use the discrete consumption acknowledgment (DCA) mode. This mode allows the consumption indication to be directly associated with a frame enqueue operation from the portal (that is, after the frame has been processed and is on the way to the egress queue). This tracking of the DQRR Ring Pointer CI (Consumer Index) helps implement frame ordering by ensuring that QMan does not dequeue a frame from the same FQ (or flow) to a different core until the processing is completed.

7.4.1.5 QMan Scheduling

The QMan links the FQs to producers and consumers (of data traffic) within the SoC.

QMan: Queue schedule options

The primary communication path between QMan and the processor cores is the portal memory structure. QMan uses this interface to schedule the frames to be processed on a per-core basis. For a dedicated channel, the process is straightforward: the QMan places an entry in the DQRR for the portal (processor) of the dedicated channel and dequeues the frame from an FQ to the portal. To do this, QMan determines, based on the priority scheme (previously described) for the channel, which frame should be processed next and then adds an entry to the DQRR for the portal associated with the channel.

When configured for push mode, once the portal requests QMan to provide frames for processing, QMan provides frames until halted. When the DQRR is full and more frames are destined for the portal, QMan waits for an empty slot to become available in the DQRR and then adds more entries (frames to be processed) as slots become available.

When configured for pull mode, the QMan only adds entries to the DQRR at the direct request of the portal (software request). The command to the QMan that determines if a push or pull mode is implemented and tells QMan to provide either one or from one to three (up to three if there are that many frames to be dequeued) frames at a time. This is a tradeoff of smaller granularity (for one frame only) versus memory access consolidation (if the up to three frames option is selected).

When the system is configured to use pool channels, a portal may get frames from more than one channel and a channel may provide frames (work) to more than one portal (core). QMan dequeues frames using the same mechanism described above (updating DQRR) and QMan also provides for some specific scheduling options to account for the pool channel case in which multiple cores may process the same channel.

QMan: Default Scheduling

The default scheduling is to have an FQ send frames to the same core for as long as that FQ is active. An FQ is active until it uses up its allocated credit or becomes empty. After an FQ uses its credit, it is rescheduled again, until it is empty. For its schedule opportunity, the FQ is active and all frames dequeued during the opportunity go to the same core. After the credit is consumed, QMan reactivates that FQ but may assign the flow processing to a different core. This provides for a sticky affinity during the period of the schedule opportunity. The schedule opportunity is managed by the amount of credit assigned to the FQ.

NOTE

A larger credit assigned to an FQ provides for a stickier core affinity, but his makes the processing work granularity larger and may affect load balancing.



Figure 75. Default Scheduling

QMan: Hold Active Scheduling

With the hold active option, when the QMan assigns an FQ to a particular core, that Q is affined to that core until it is empty. Even after the FQ's credit is consumed, hen it is rescheduled with the next schedule opportunity, the frames go to the same core for processing. This effectively makes the flow-to-core affinity stickier than the default case, ensuring the same flow is processed by the same core for as long as there are frames queued up for processing. Because the flow-to-core affinity is not hard-wired as in the dedicated channel case, the software may still need to account for potential order issues. However, because of flow-to-core biasing, the flow state data is more likely to remain in L1 or L2 cache, increasing hit rates and thus improving processing performance. Because of the specific QMan implementation, only four FQs may be in held active state at a given time.



Figure 76. Hold Active Scheduling

QMan: Avoid blocking scheduling

Avoid blocking scheduling QMan can also be scheduled in the avoid blocking mode, which is mutually exclusive to hold active. In this mode, QMan schedules frames for an FQ to any available core in the pool channel. For example, if the credit allows for three frames to be dequeued, the first frame may go to core 1. But, when that dequeue fills core 1's DQRR, QMan finds the next available DQRR entry in any core in the pool. With avoid blocking mode there is no biasing of the flow to core affinity. This mode is useful if a particular flow either has no specific order requirements or the anticipated processing required for a single flow is expected to consume more than one core's worth of processing capability.

Alternatively, software can bypass QMan scheduling and directly control the dequeue of frame descriptors from the FQ. This mode is implemented by placing the FQ in parked state. This allows software to determine precisely which flow will be processed (by the core running the software). However, it requires software to manage the scheduling, which can add overhead and impact performance.



Figure 77. Avoid Blocking Scheduling

QMan: Order Definition/ Restoration

The QMan provides a mechanism to strictly enforce ordering. Each FQ may be defined to participate in the process of an order definition point and/or an order restoration point. On ingress, an order definition point provides for a 14 bit sequence number assigned to each frame (incremented per frame) in a FQ in the order in which they were received on the interface. The sequence number is placed in the DQRR entry for the frame when it is dequeued to a portal. This allows software to efficiently determine which packet it is currently processing in the sequence without the need to access a shared (between cores) data structure. On egress, an order restoration point delays placing a frame onto the FQ until the expected next sequence number is encountered. From the software standpoint, once it has determined the relative sequence of a packet, it can enqueue it and resume other processing in a fire-and-forget manner.

NOTE

The order definition points and order restoration points are not dependent on each other; it is possible to have one without the other depending on application requirements. To effectively use these mechanisms, the packet software must be aware of the sequence tagging.



Figure 78. Order Definition/Restoration

As processors enqueue packets for egress, it is possible that they may skip a sequence number because of the nature of the protocol being processed. To handle this situation, each FQ that participates in the order restoration service maintains its own Next Expected Sequence Number (NESN). When the difference between the sequence number of the next expected and the most recently received sequence number exceeds the configurable ORP threshold, QMan gives up on the missing frame(s) and autonomously advances the NESN to bring the skew within threshold. This causes any deferred enqueus that are currently held in the ORP link list to become unblocked and immediately enqueue them to their destination FQ. If the “skipped” frame arrives after this, the ORP can be configured to reject or immediately enqueu the late arriving frame.

7.4.1.6 BMan

The BMan block manages the data buffers in memory. Processing cores, FMan, SEC and PME all may get a buffer directly from the BMan without additional software intervention. These elements are also responsible for releasing the buffers back to the pool when the buffer is no longer in use.

Typically, the FMan directly acquires a buffer from the BMan on ingress. When the traffic is terminated in the system, the core generally releases the buffer. When the traffic is received, processed, and then transmitted, the same buffer may be used throughout the process. In this case, the FMan may be configured to release the buffer automatically, when the transmit completes.

The BMan also supports single or multi-buffer frames. Single buffer frames generally require the adequately defined (or allocated) buffer size to contain the largest data frame and minimize system overhead. Multi-buffer frames potentially allow

better memory utilization, but the entity passed between the producers/consumers is a scatter-gather table (that then points to the buffers within the frame) rather than the pointer to the entire frame, which adds an extra processing requirement to the processing element.

The software defines pools of buffers when the system is initialized. The BMan unit itself manages the pointers to the buffers provided by the oftware and can be configured to interrupt the software when it reaches a condition where the number of free buffers is depleted (so that software may provide more buffers as needed).

7.4.1.7 Order Handling

The DPAA helps address packet order issues that may occur as a result of running an application in a multiple processor environment. And there are several ways to leverage the DPAA to handle flow order in a system. The order preservation technique maps flows such that a specific flow always executes on a specific processor core.

For the case that DPAA handles flow order, the individual flow will not have multiple execution threads and the system will run much like a single core system. This option generally requires less impact to legacy, single-core software but may not effectively utilize all the processing cores in the system because it requires using a dedicated channel to the processors. The FMan PCD can be configured to either directly match a flow to a core or to use the hashing to provide traffic spreading that offers a permanent flow-to-core affinity.

If the application must use pool channels to balance the processing load then the software must be more involved in the ordering. The software can make use of the order restoration point function in QMan, which requires the software to manage a sequence number for frames enqueued on egress. Alternatively, the software can be implemented to maintain order by biasing the stickiness of flow affinity with default or hold active scheduling; lock contention and cache misses can be biased to increase performance.

If there are no order requirements then load balancing can be achieved by associating the non-ordered traffic to a pool of cores.

NOTE

All of these techniques may be implemented simultaneously on the same SoC; as long as the flow definition is precise enough to split the traffic types, it is simply a matter of proper defining the FQs and associating them to the proper channels in the system.

Using the exact match flow definition to preserve order

The simplest technique for preserving order is to route the ingress traffic of an individual flow to a particular core. For the particular flow in question, the system appears as a legacy, single-core programming model and, therefore, has minimal impact on the structure of the software. In this case, the flow definition determines the core affinity of a flow.



Figure 79. Direct Flow-to-Core Mapping (Order Preserved)

This technique is completely deterministic: the DPAA forces specific flows to a specific processor, so it may be easier to determine the performance assuming the ingress flows are completely understood and well defined. Notice that a particular processor core may become overloaded with traffic while another sits idle for increasingly random flow traffic rates.

To implement this sort of scheme, the FMan must be configured to exactly match fields in the traffic stream. This approach can only be used for a limited number of total flows before the FMan's internal resources are consumed.

In general, this sort of hard-wired approach should be reserved for either critical out-of-band traffic or for systems with a small number of flows that can benefit from the highly deterministic nature of the processing.

Using hashing to distribute flows across cores

The FMan can be configured to extract data from a field or fields within the data frame, build a key from that, and then hash the resultant key into a smaller number. This is a useful technique to handle a larger number of flows while ensuring that a particular flow is always associated with a particular core. An example is to define a flow as an IPv4 source + IPv4 destination address. Both fields together constitute 64 bits, so there are 264 possible combinations for the flow in that definition. The FMan then uses a hash algorithm to compress this into a manageable number of bits. Note that, because the hash algorithm is consistent, packets from a particular flow always go to the same FQ. By utilizing this technique, the flows can be spread in a pseudo-random, consistent (per flow) manner to a smaller number of FQs. For example, hashing the 64 bits down to 2

bits spreads the flows among four queues. Then these queues can be assigned to four separate cores by using a dedicated channel; effectively, this appears as a single-core implementation to any specific flow.

This spreading technique works best with a large number of possible flows to allow the hash algorithm to evenly spread the traffic between the FQs. In the example below, when the system is only expected to have eight flows at a given time, there is a good chance the hash will not assign exactly two flows per FQ to evenly distribute the flows between the four cores shown. However, when the number of flows handled is in the hundreds, the odds are good that the hash will evenly spread the flows for processing.



Figure 80. Simple flow distribution via hash (order preserved)

To optimize cache warming, the total number of hash buckets can be increased with flow-to-core affinity maintained. When the number of hash values is larger than the number of expected flows at a given time, it is likely though not guaranteed that each FQ will contain a single flow. For most applications, the penalty of a hash collision is two or more flows within a single FQ. In the case of multiple flows within a single FQ, the cache warming and temporary core affinity benefits are reduced unless the flow order is maintained per flow.

Note that there are 24 bits architected for the FQ ID, so there may be as many as 16 million FQs in the system. Although this total may be impractical, this does allow for the user to define more FQs than expected flows in order to reduce the likelihood of a hash collision; it also allows flexibility in assigning FQID's in some meaningful manner. It is also possible to hash some

fields in the data frame and concatenate other parse results, possibly allowing a defined one-to-one flow to FQ implementation without hash collisions.



Figure 81. . Using hash to assign one flow per FQ (order preserved and cache stashing effective)

7.4.1.8 Pool Channels

A user may employ a pool channel approach where multiple cores may pool together to service a specific set of flows. This alternative approach allows potentially better processing balance, but increases the likelihood that packets may be processed out of order allowing egress packets to pass ingress packets.

So far, the techniques discussed in this white paper have involved assigning specific flows to the same core to ensure that the same core always processes the same flow or set of flows, thereby preserving flow order. However, depending on the nature of the flows being processed (that is, variable frame sizes, difficulty efficiently spreading due to the nature of the flow contents, and so on), this may not effectively balance the processing load among the cores. Alternatively, a user may employ a pool channel approach where multiple cores may pool together to service a specific set of flows. This alternative approach allows potentially better processing balance, but increases the likelihood that packets may be processed out of order allowing egress packets to pass ingress packets. When the application does not require flows to be processed in order, the pool channel approach allows the easiest method for balancing the processing load. When a pool channel is used and order is required, the software must maintain order. The hardware order preservation may be used by the software to implement order

without requiring locked access to shared state information. When the system uses a software lock to handle order then the default scheduling and hold active scheduling tends to minimize lock contention.



Figure 82. Using pool channel to balance processing

Order preservation using hold active scheduling and DCA mode

As shown in the examples above, order is preserved as long as two or more cores never process frames from the same flow at the same time. This can also be accomplished by using hold active scheduling along with discrete consumption acknowledgment (DCA) mode associated with the DQRR. Although flow affinity may change for an FQ with hold active scheduling when the FQ is emptied, if the new work (from frames received after the FQ is emptied) is held off until all previous work completes, then the flow will not be processed by multiple cores simultaneously, thereby preserving order.

When the FQ is emptied, QMan places the FQ in hold suspended state, which means that no further work for that FQ is enqueued to any core until all previously enqueued work is completed. Because DCA mode effectively holds off the consumption notification (from the core to QMan) until the resultant processed frame is enqueued for egress, this implies processing is completely finished for any frames in flight to the core. After all the in-flight frames have been processed, QMan reschedules the FQ to the appropriate core.

NOTE

After the FQ is empty and when in hold active mode, the affinity is not likely to change. This is because the indication of “completeness” from the core currently processing the flow frees up some DQRR slots that could be used by QMan when it restarts enqueuing work for the flow. The possibility of the flow-to-core affinity changing when the FQ empties is only discussed as a worst case possibility with regards to order preservation.



Figure 83. Hold active to held suspended mode



Figure 84. Held suspended to hold active mode

Congestion management

From an overall system perspective, there are multiple potential overflow conditions to consider. The maximum number of frames active in the system (the number of frames in flight) is determined by the amount of memory allocated to the Packed Frame Queue Descriptors (PQFD's). Each PQFD is 64 bytes and can identify up to three frames, so the total number of frames that can be identified by the PQFDs is equal to the amount of memory allocated for PQFD space divided by 64 bytes (per entry) multiplied by three (frames per entry).

A pool of buffers may deplete in BMan. This depends on how many buffers have been assigned by software for BMan. BMan may raise an interrupt to request more buffers when in a depleted state for a given pool; the software can manage the congestion state of the buffer pools in this manner.

In addition to these high-level system mechanisms, congestion management may also be identified specific to the FQs. A number of FQs can be grouped together to form a congestion group (up to 256 congestion groups per system for most DPAA SoCs). These FQs need not be on the same channel. The system may be configured to indicate congestion either by consider the aggregate number of bytes within the FQ's in the congestion group or by the aggregate number of frames within the congestion group. The frame count option is useful when attempting to manage the number of buffers in a buffer pool as they

are used by a particular core or group of cores. The byte count is useful to manage the amount of system memory used by a particular core or group of cores.

When the total number of frames/bytes within the frames in the congestion group exceeds the set threshold, subsequent enqueues to any of the FQs in the group are rejected; in general, the frame is dropped. For the congestion group mechanism, the decision to reject is defined by a programmed weighted random early discard (WRED) algorithm programmed when the congestion group is defined.

In addition, a specific FQ can be set to a particular maximum allowable depth (in bytes); after the threshold is reached enqueue attempts will be rejected. This is a maximum threshold: there is no WRED algorithm for this mechanism. Note that, when the FQ threshold is not set, a specific FQ may fill until some other mechanism (because it's part of a congestion group or system PQFD depletion or BMAN depletion) prevents the FQ from getting frames. Typically, FQs within a congestion group are expected to have a maximum threshold set for each FQ in the group to ensure a single queue does not get stuck and unfairly consume the congestion group. Note that, when an FQ does not have a queue depth set and/or is not a part of a congestion group, the FQ has no maximum depth. It would be possible for a single queue to have all the frames in the system, until the PQFD space or the buffer pool is exhausted.

7.4.1.9 Application Mapping

The first step in application mapping is to determine how much processing capability is required for tasks that may be partitioned separately.

Processor core assignment

Consider a typical networking application with a set of distinct control and data plane functionality. Assigning two cores to perform control plane tasks and six cores to perform data plane tasks may be a reasonable partition in an eight-core device. When initially identifying the SoC required for the application, along with the number of cores and frequencies required, the designer makes some performance assumptions based on previous designs and/or applicable benchmark data.

Define flows

Next, define what flows will be in the system. Key considerations for flow definition include the following:

- Total number of flows expected at a given time within the system
- Desired flow-to-core affinity, ingress flow destination
- Processor load balancing
- Frame sizes (may be fixed or variable)
- Order preservation requirement
- Traffic priority relative to the flows
- Expected bandwidth requirement of specific flows or class of flows
- Desired congestion handling



Figure 85. Example Application with Three Classes

In the figure above, two cores are dedicated to processing control plane traffic, four cores are assigned to process general data traffic and special time critical traffic is split between two other cores. In this case, assume the traffic characteristics in the following table. With this system-level definition, the designer can determine which flows are in the system and how to define the FQs needed.

Table 120. Traffic characteristics

Characteristic	Definition
Control plane traffic	• Terminated in the system and any particular packet sent has no dependency on previous or subsequent packets (no order requirement). • May occur on ports 1, 2 or 3. • Ingress control plane traffic on port three is higher priority than the other ports. • Any ICMP packet on ports 1, 2 or 3 is considered control plane traffic. • Control plane traffic makes up a small portion of the overall port bandwidth.

Table continues on the next page...

Table 120. Traffic characteristics (continued)

Characteristic	Definition
General data plane traffic	• May occur on ports 1, 2 or 3 and is expected to comprise the bulk of the traffic on these ports. • The function performed is done on flows and egress packets must match the order of ingress packets. • A flow is identified by the IP source address. • The system can expect up to 50 flows at a time. • All flows have the same priority and a lower priority than any control plane traffic. • It is expected that software will not always be able to keep up with this traffic and the system should drop packets after some amount of packets are within the system.
Pseudo real-time traffic	• A high amount of determinism is required by the function. • This traffic only occurs on port 4 and port 5 and is identified by a proprietary field in the header; any traffic on these ports without the proper value in this field is dropped. • All valid ingress traffic on port 4 is to be processed by core 7, ingress traffic on port 5 processed by core 8. • There are only two flows, one from port 4 to port 5 and one from port 5 to port 4, egress order must match ingress order. • The traffic on these flows are the highest priority.

Identify ingress and egress frame queues (FQs)

For many applications, because the ingress flow has more implications for processing, it is easier to consider ingress flows first. In the example above, the control plane and pseudo real-time traffic FQ definitions are fairly straightforward. For the control plane ingress, one FQ for lower priority traffic on ports 1 and 2 and one for the higher priority traffic would work. Note that two ports can share the same queue on ingress when it does not matter for which core the traffic is destined. For ingress pseudo real-time traffic, there is one FQ on port 4 and one FQ on port 5.

The general data plane ingress traffic is more complicated. Multiple options exist which maintain the required ordering for this traffic. While this traffic would certainly benefit from some of the control features of the QMan (cache warming, and so on), it is best to have one FQ per flow. Per the example, the flow is identified by the IP source (32-bits), which consists of too many bits to directly use as the FQID. The hash algorithm can be used to reduce the 32-bits to a smaller number; in this case, six bits would generate 64 queues, which is more than the anticipated maximum flows at a given time. However, this is not significantly more than maximum flow expected, so more FQs can be defined to reduce hash collisions. Note that, in this case, a hash collision implies that two flows are assigned to the same FQ. As the ingress FQs fill directly from the port, the packet order is still maintained when there is a collision (two flows into one FQ). However, having two flows in the same FQ tends to minimize the impact of cache warming. There may be other possibilities to refine the definition of flows to ensure a one-to-one mapping of flows to FQs (for example, concatenating other fields in the frame) but for this example assume that an 8 bit hash (256 FQs) minimizes the likelihood of two flows in the FQ to an acceptable level.

Consider the case in which, on ingress, there is traffic that does not match any of the intended flow definitions. The design can handle these by placing unidentifiable packets into a separate garbage FQ or by simply having the FMan discard the packets.

On egress control traffic, because the traffic may go out on three different ports, three FQs are required. For the egress pseudo real-time traffic, there is one queue for each port as well.

For the egress data plane traffic, there are multiple options. When the flows are pinned to a specific core, it might be possible to simply have one queue per port. In this case, the cores would effectively be maintaining order. However, for this example, assume that the system uses the order definition/order restoration mechanism previously described. In this case, the system needs to define an FQ for each egress flow. Note that, since software is managing this, there is no need for any sort of hash algorithm to spread the traffic; the cores will enqueue to the FQ associated with the flow. When there are no more than 50 flows in the system at one time, and number of egress flows per port is unknown, the system could define 50 FQs for each port when the DPAA is initialized.

Define PCD configuration for ingress FQs

This step involves defining how the FMan splits the incoming port traffic into the FQs. In general, this is accomplished using the PCD (Parse, Classify, Distribute) function and results in specific traffic assigned to a specific FQID. Fields in the incoming packet may be used to identify and split the traffic as required. For this key optimization case, the user must determine the correct field. The example is as follows:

- For the ingress control traffic, the ICMP protocol identifier is the selector or key. If the traffic is from ports 1 or 2 then that traffic goes to one FQID. If it is from port 3, the traffic goes to a different FQID because this needs to be separated and given a higher priority than the other two ports.
- For the ingress data plane traffic, the IP source field is used to determine the FQID. The PCD is then configured to hash the IP source to 8 bits, which will generate 256 possible FQs. Note that this is the same, regardless of whether the packet came from ports 1, 2, or 3.
- For the ingress pseudo real-time traffic, the PCD is configured to check for the proprietary identifier. If there is a match then the traffic goes to an FQID based on the ingress port. If there is no match then the incoming packet is discarded. Also, the soft parser needs to be configured/programmed to locate the proprietary identifier.

Note that the FQID number itself can be anything (within the 24 bits to define the FQ). To maintain meaning, use a numbering scheme to help identify the type of traffic. For the example, define the following ingress FQIDs:

- High priority control: FQID `0x100`
- Low priority control: FQID `0x200`
- General data plane: FQID `0x1000 – 0x10FF`
- Pseudo real-time traffic: FQID `0x2000` (port 4), FQID `0x2100` (port 5)

The specifics for configuring the PCDs are described in the **DPAA Reference Manual (link)** and in the Software Developer Kit (SDK) used to develop the software.

7.4.1.10 FQ/WQ/Channel

For each class of traffic in the system, the FQs must be defined together with both the channel and the WQ to which they are associated. The channel association affines to a specific processor core while the WQ determines priority.

Consider the following by class of traffic:

- The control traffic goes to a pool of two cores with priority given to traffic on port 3.
- The general data plane traffic goes to a pool of 4 cores.
- The pseudo real-time traffic goes to two separate cores as a dedicated channel.

Note that, when the FQ is defined, in addition to the channel association, other parameters may be configured. In the application example, the FQs from 1000 to 10FF are all assigned to the same congestion group; this is done when the FQ is initialized. Also, for these FQs it is desirable to limit the individual FQ length; this would also be configured when the FQ is initialized.

Because the example application is going to use order definition/order restoration mode, this setting needs to be configured for each FQ in the general data plane traffic (FQID 0x1000-0x10FF). Note that order is not required for the control plane traffic and that order is preserved in the pseudo real-time traffic because the ingress traffic flows are mapped to specific cores.

QMan configuration considerations include the congestion management and pool channel scheduling. A congestion group must be defined as part of QMan initialization. (Note that the FQ initialization is where the FQ is bound to a congestion group.) This is where the total number of frames and the discard policy of the congestion group are defined. Also, consider the QMan scheduling for pool channels. In this case, the default of temporarily attaching an FQ to a core until the FQ is empty will likely work best. This tends to keep the caches current, especially for the general data plane traffic on cores 3-6.



Figure 86. Ingress application map

Define egress FQ/WQ/channel configuration

For egress, the packets still flow through the system using the DPAA, but the considerations are somewhat different. Note that each external port has its own dedicated channel; therefore, to send traffic out of a specific port, the cores enqueue a frame to an FQ associated with the dedicated channel for that port. Depending on the priority level required, the FQ is associated with a specific work queue.

For the example, the egress configuration is as follows:

- For control plane traffic, there needs to be separate queues for each port this traffic may use. These FQs must be assigned to a WQ that is higher in priority then the WQ used for the data plane traffic. The example shown includes a strict priority (over the data plane traffic) for ports 1 and 2 with the possibility of WRED with the data plane traffic on port 3.
- Because the example assumes that the order restoration facility in the FQs will be utilized, there must be one egress FQ for each flow. The initial system assumptions are for up to 50 flows of this type; however, the division by port is unknown, the FQs can be assigned so that there are at least 50 for each port. Note that FQs can be added when the flow is discovered or they can be defined at system initialization time.
- For the pseudo real-time traffic, per the initial assumptions, core 7 sends traffic out of port 4 and core 8 sends traffic out of port 5. As the flows are per core, the order is preserved because of this mapping. These are assigned to WQ2, which allows definition for even higher priority traffic (to WQ1) or lower priority traffic for future definition on these ports.

As stated before, the FQIDs can be whatever the user desires and should be selected to help keep track of what type of traffic the FQ's are associated. For this example:

- Control traffic for ports 1, 2, 3 are FQID 300, 400, 500 respectively.
- Data plane traffic for ports 1, 2, 3 are FQID 3000-303F, 4000-403F, and 5000-503F respectively, this provides for 64 FQ's per port on egress.
- The pseudo real-time traffic uses FQID 6000 for port 4 and 7000 for port 5.

Because this application makes use of the order restoration feature, an order restoration point must be defined for each data plane traffic flow. Also, congestion management on the FQs may be desirable. Consider that the data plane traffic may come in on multiple ports but may potentially be consolidated such that is egresses out a single port. In this case, more traffic may be attempted to be enqueued to a port than the port interface rate may allow, which may cause congestion. To manage this possibility, three congestion groups can be defined each containing all the FQs on each of the three ports that may have the control plus data plane traffic. As previously discussed, it may be desirable to set the length of the individual FQs to further manage this potential congestion.



Figure 87. Egress application map

End of Document

7.4.2 Queue Manager (QMan) and Buffer Manager (BMan)

7.4.2.1 QMan/BMan Drivers Release Notes

Description

This document describes Linux and USDPAA drivers for the QMan and BMan hardware blocks underlying the QorIQ data path. QMan and BMan have independent drivers but their implementation and interfaces are very much analogous due to the similar CCSR and Corenet programming interfaces for each. As such, we will describe here "the driver", when in fact the description applies to both the QMan and BMan drivers equally and independently.

The driver targets the Linux and USDPAA environments. The majority of the code is shared between the environments. Environmental differences are dealt with by including a compatibility layer in the USDPAA code. This code redefines Linux-specific functionality for use in the other environments (for example `irqs` and `spinlocks`).

The driver has two parts to it, "config" and "portal", corresponding to the two complimentary programming interfaces exposed by the device itself - these are described below. Additionally there is a self-test module for each driver that uses the portal interface to perform some basic tests provided one or more portals are made available to the OS via its device-tree.

CCSR, or "global config"

The CCSR map and associated registers allows the device to be configured and controlled in a global/un-partitioned manner. This includes such basic notions as configuring the device's private memory region(s), configuring the hardware interfaces that are exposed by QMan/BMan to the dependent hardware blocks (CAAM, PME, Fman), managing global device error interrupts, etc. Only one "control" operating system should map to this CCSR register space in the case that a hypervisor is managing multiple guests. Other operating systems like secondary Linux instances or USDPAA applications do not have access to CCSR registers.

Corenet portals

Use of QMan/BMan is via a number of independent portals located within sub-regions of a corenet memory map. Each portal has its own copy of the device interfaces that allow independent and parallel use of QMan/BMan functionality, possibly by different operating systems or by different processors (or threads) within a single operating system. When a hypervisor is managing multiple guests, each guest device tree indicates to the guest the portal it has access to. The device tree specifies a one to one mapping between cores and QMan/BMan portals. This mapping is assumed by the high-level QMan/BMan APIs. This should not be changed unless only low-level QMan/BMan APIs are to be used.

Functionality

Configuration

The QMan device is configured via device-tree nodes and by some compile-time options controlled via Linux's Kconfig system. See the “QMan and BMan Kernel Configure Options” section for more info.

API

For the Linux kernel, the C interface of the QMan and BMan drivers provides access to portal-based functionality for arbitrary higher-layer code, hiding all the mux/demux/locking details required for shared use by multiple driver layers (networking, pattern matching, encryption, IPC, etc.) The driver makes 1-to-1 associations between cpus and portals to improve cache locality and reduce locking requirements. The QMan API permits users to work with Frame Queues and callbacks, independently of other users and associated portal details. The BMan API permits users to work with Buffer Pools in a similar manner.

For USDPAA, the driver associates portals with threads (in the *pthreads* sense), so the above comments about “shared use by multiple driver layers” only applies with respect to code executed within the thread owning a portal. To benefit from cache locality, and particularly from portal stashing, USDPAA-enabled threads are generally expected to be configured to execute

on the same core that the portal is assigned to. Indeed, the USDPAA API for threads to call to initialise a portal takes the core as a function parameter. Please see the USDPAA User Guide for more information (as well as the “Queue Manager, Buffer Manager API Reference Manual”).

DPAA allocator

The DPAA allocator is a purely software-based range-allocator, but this must be explicitly seeded with a hard-coded range of values and is not shared between operating systems. The DPAA allocator is used to allocate all QMan and BMan resource, i.e bman-bpid, qman-fqid, qman-pool, qman-cgrid, ceetm-sp, ceetm-lni, ceetm-lfqid, ceetm-ccgrid.

Sysfs Interface

QMan and BMan have a sysfs interface. Refer to the Queue Manager, Buffer Manager API reference Manual for details

Debugfs Interface

Both the QMan and BMan have a debugfs interface available to assist in device debugging. The code can be built either as a loadable module or statically.

Module Loading

The drivers are statically linked into the kernel. Driver self-tests and the debugfs interface may be built as dynamically loadable modules.

QMan and BMan Kernel Configure Options

Common Kernel Configure Options	Description
CONFIG_STAGING	Required in order to make “staging” drivers such as QMan/BMan available.
CONFIG_FSL_DPA	Required to build either QMan and/or BMan drivers.
CONFIG_FSL_DPA_CHECKING	Compiles in additional sanity-checks, at the expense of minor performance degradation. Recommended for debugging, but not for benchmarking.
CONFIG_FSL_DPA_CAN_WAIT	Compiles in support for interfaces and functionality that allow callers to optionally be put to “sleep” waiting for temporarily blocked resources to become available rather than returning errors. Eg. enqueuing when an enqueue ring is full. This is enabled unconditionally on linux.
CONFIG_FSL_DPA_CAN_WAIT_SYNC	Similar to “_CAN_WAIT”, but supports additional API flags for waiting for asynchronous operatoins to complete. Eg. after starting a volatile dequeue, wait for all dequeues to complete. This is enabled unconditionally on linux.
CONFIG_FSL_DPA_PIRQ_FAST	If set, causes portals to initialise with fast-path interrupt sources enabled. (Otherwise, polling APIs must be called to perform fast-path processing.) This is enabled unconditionally on linux.
CONFIG_FSL_DPA_PIRQ_SLOW	If set, causes portals to initialise with slow-path interrupt sources enabled. (Otherwise, polling APIs must be called to perform slow-path processing.) This is enabled unconditionally on linux.
CONFIG_FSL_DPA_PORTAL_SHARE	Compiles in support for sharing one CPU's portal with all online CPUs that do not have their own. Useful when assigning most portals to USDPAA applications and leaving only a minimum for kernel requirements, in which case Tx events on all CPUs can be handled by the network driver. This is enabled by default, as the microscopic performance overhead of checking this option is not noticable in the kernel environment.

QMan Kernel Configure Options	Description
CONFIG_FSL_QMAN	Required to build the QMan driver
CONFIG_FSL_QMAN_CONFIG	Handles config/CCSR nodes in the device-tree and initialises the corresponding devices
CONFIG_FSL_QMAN_TEST	Builds a self-test kernel module (static or dynamic) that will, if QMan portal nodes are available in the device-tree, exercise one of the portals and panic() the kernel if any errors are detected.
CONFIG_FSL_QMAN_TEST_STASH_POTATO	This requires the presence of multiple unused cpu-affine portals, and performs a "hot potato" style test enqueuing/dequeuing a frame across a series of FQs scheduled to different portals (and cpus). The intention is to test stashing. The "potato" will visit each "spoon" (portal/cpu pair) during the test. Each "potato" frame has a single cacheline of data that is read-modify-written by each cpu/portal before passing it to the next.
CONFIG_FSL_QMAN_TEST_HIGH	This requires the presence of cpu-affine portals, and performs high-level API testing with them (whichever portal(s) are affine to the cpu(s) the test executes on).
CONFIG_FSL_QMAN_TEST_ERRATA	This requires the presence of cpu-affine portals, and performs testing that handling for known hardware-errata is correct.
CONFIG_FSL_QMAN_DEBUGFS	This option enables files in the debugfs filesystem.

BMan Kernel Configure Options	Description
CONFIG_FSL_BMAN	Required to build the BMan driver
CONFIG_FSL_BMAN_CONFIG	Handles config/CCSR nodes in the device-tree and initialises the corresponding devices
CONFIG_FSL_BMAN_TEST	Builds a self-test kernel module (static or dynamic) that will, if BMan portal nodes are available in the device-tree, exercise one of the portals and panic() the kernel if any errors are detected.
CONFIG_FSL_BMAN_TEST_HIGH	Performs high-level API testing.
CONFIG_FSL_BMAN_TEST_THRESH	Multi-threaded testing of BMan pool depletion handling.
CONFIG_FSL_BMAN_DEBUGFS	This option enables files in the debugfs filesystem.

Device-tree nodes

Device tree nodes are used to describe QMan/BMan resources to the driver, some of which are specific to control-plane s/w (i.e. depending on CCSR access) and some of which relate to portal usage for control and data plane s/w.

CCSR, or "global config"

The "fsl,qman" and "fsl,bman" nodes (i.e. these "compatible" property types) indicate the presence and location of the 4Kb "Configuration, Control, and Status Register" (CCSR) space, for use by a single control-plane driver instance to initialise and manage the device. The device-tree usually groups all such CCSR maps as sub-nodes under a parent node that represents the SoCs entire CCSR map, usually named "soc" or "ccsr". For example;

```
soc {
      #address-cells = <1>;
      #size-cells = <1>;
      device_type = "soc";
```

```
        compatible = "simple-bus";

        ddr1: memory-controller@8000{
            [...]
        };
        i2c@118000 {
            [...]
        };
        mpic: pic@40000 {
            [...]
        };

        qman: qman@318000 {
            compatible = "fsl,qman";
            reg = <0x318000 0x1000>;
            interrupts = <16 2 1 3>;
            /* Commented out, use default allocation */
            /* fsl,qman-fqd = <0x0 0x20000000 0x0 0x01000000>; */
            /* fsl,qman-pfdr = <0x0 0x21000000 0x0 0x01000000>; */
        };
        bman: bman@31a000 {
             compatible = "fsl,bman";
             reg = <0x31a000 0x1000>;
             interrupts = <16 2 1 3>;
             /* Same as fsl,qman-*, use default allocation */
             /* fsl,bman-fbpr = <0x0 0x22000000 0x0 0x01000000>; */
        };
        [...]
    };
```

Contiguous memory

The `fsl,qman-fqd`, `fsl,qman-pfdr`, and `fsl,bman-fbpr` properties can be used to specify which contiguous sub-regions of memory should be used for the various memory requirements of QMan/BMan. The properties use 64-bit values, so 4 cells express the address/size 2-tuple to use. In the above example, if uncommented, the QMan/BMan resources would be allocated in the range `0x2000000-0x221fffff`, with 16MB each for QMan FQD and PFDR memory and BMan FBPR memory. If these properties are not specified (or they are commented out) in the device tree, then default values hard-coded within the QMan and BMan drivers are used instead. The linux kernel will reserve these memory ranges early on boot-up. Note that in the case of a hypervisor scenario, these memory ranges are relative to the partition memory space of the control-plane guest OS.

QMan and BMan Corenet portals

The QMan Corenet portal interface in QorIQ P4080/P3041/P5020 provides up to 10 distinct memory-mapped interfaces for use by software to interact efficiently with QMan functionality. The number may be different for other SoCs - the QMan driver determines the available corenet portals from the device tree. The portal nodes are at the physical address scope (unlike the device-tree node for the BMan device itself, which is within the “soc” physical address node that represents CCSR). These nodes indicate the physical address ranges of the cache-enabled and cache-inhibited sub-regions of the portal (respectively), and look something like the following;

```
    qman-portals@ffe4200000 {
        #address-cells = <0x1>;
        #size-cells = <0x1>;
        compatible = "simple-bus";
        qportal0: qman-portal@0 {
            [...]
        };
        [...]
```

```
        qportal3: qman-portal@c000 {
            cell-index = <0x3>;
            compatible = "fsl,qman-portal";
            reg = <0xc000 0x4000 0x103000 0x1000>;
            interrupts = <110 0x2 0 0>;
            fsl,qman-channel-id = <0x3>;
        };
        [...]
    };
```

QMan FQID-range allocation

The "fsl,fqid-range" node (i.e. these "compatible" property types) indicates a range of FQIDs to use for FQID allocation by the QMan driver. The range within the node is specified using a property of the same name, and whose two cells are the starting FQID value and the count. Multiple nodes can be provided to seed the allocator with a discontiguous set of FQIDs.

Eg. to specify that the allocator use FQIDs between 256 and 512 inclusive;

```
    qman-fqids@0 {
        compatible = "fsl,fqid-range";
        fsl,fqid-range = <256 256>;
    };
```

BMan BPID-range allocation

The "fsl,bpool-range" node (i.e. these "compatible" property types) indicates a range of BPIDs to use for BPID allocation by the BMan driver. The range within the node is specified using a property of the same name, and whose two cells are the starting BPID value and the count. Multiple nodes can be provided to seed the allocator with a discontiguous set of BPIDs.

Eg. to specify that the allocator use BPIDs between 32 and 64 inclusive;

```
    bman-bpids@0 {
        compatible = "fsl,bpid-range";
        fsl,bpid-range = <32 32>;
    };
```

Compile-time Configuration Options

The "Kernel Configure Options" above describe the compile-time configuration options for the kernel. The device tree entries are also "compile-time", and are described above.

Source Files

As mentioned earlier, the QMan/BMan drivers support Linux and USDPAA environments. Many of the files have the same contents between the different environments, though the files are located at different paths to satisfy the different build systems for each.

For DPAA QBMan drivers, all the files are located in `drivers/staging/fsl_qbman/ directory`

For the device trees, all files are located in `arch/powerpc/boot/dts/fsl/ and arch/arm64/boot/dts/freescale/`

USDPAA

Source Files	Description
include/usdpaa/fsl_qman.h	The QMan driver APIs

Table continues on the next page...

Table continued from the previous page...

Source Files	Description
include/usdpaa/fsl_bman.h	The BMan driver APIs
include/usdpaa/fsl_usd.h	The USDPAA-specific APIs for QMan/BMan (eg. Binding portals to threads, support for UIO-based interrupt handling, etc.)
include/usdpaa/compat.h	The QMan/BMan driver compatibility shims
include/usdpaa/compat_list.h	The QMan/BMan driver compatibility shims, linked-list support.
src/qbman/qman_*.*	The QMan driver
src/qbman/bman_*.*	The BMan driver
src/qbman/dpa_sys.h	USDPAA-specific definitions shared by the QMan/BMan drivers.
src/qbman/dpa_alloc.c	USDPAA support for dpa allocator.
src/qbman/06-usdpaa-uio.rules	Udev rules to create appropriately-named /dev entries when the kernel registers portals as UIO devices.

Build Procedure

The procedure is a standard SDK build, which includes Linux kernel and USDPAA drivers by default.

Test Procedure

The QMan/BMan drivers are used by all Linux kernel software that communicates with datapath functionality such as CAAM, PME, and/or Fman. (The exception is that kernel cryptographic acceleration presently bypasses QMan/BMan interfaces by using the device's own “job queue” interface.) Use of such datapath-based functionality provides test-coverage of user-facing features of the QMan/BMan drivers in the Linux environment. This complements the QMan/BMan unit tests that are run during development but are not part of the release. For USDPAA, all applications and tests use QMan and BMan interfaces in a fundamental way, so all imply a degree of test-coverage.

Additionally, for Linux, the QMan and BMan self-tests target QMan and BMan directly without involving other datapath blocks. If these are built statically into the kernel and the device-tree makes one or more QMan and/or BMan portals available, then the self-tests will run during the kernel boots and log output to the boot console. The output of both QMan and BMan tests resembles the following excerpts;

Detecting the CCSR and portal device-tree nodes;

```
[...]
Qman ver:0a01,01,02
[...]
Bman ver:0a02,01,00
[...]
BMan err interrupt handler present

BMan portal initialised, cpu 0

BMan portal initialised, cpu 1

BMan portal initialised, cpu 2

BMan portal initialised, cpu 3

BMan portal initialised, cpu 4
```

```
BMan portal initialised, cpu 5

BMan portal initialised, cpu 6

BMan portal initialised, cpu 7

BMan portals initialised

BMan: BPID allocator includes range 32:32

QMan err interrupt handler present

QMan portal initialised, cpu 0

QMan portal initialised, cpu 1

QMan portal initialised, cpu 2

QMan portal initialised, cpu 3

QMan portal initialised, cpu 4

QMan portal initialised, cpu 5

QMan portal initialised, cpu 6

QMan portal initialised, cpu 7

QMan portals initialised

QMan: FQID allocator includes range 256:256

QMan: FQID allocator includes range 32768:32768

QMan: CGRID allocator includes range 0:256

QMan: pool channel allocator includes range 33:15

[...]
```

Running the QMan and BMan self-tests;

```
[...]
BMAN: --- starting high-level test ---
BMAN: --- finished high-level test ---
[...]
qman_test_high starting
VDQCR (till-empty);
VDQCR (4 of 10);
VDQCR (6 of 10);
scheduled dequeue (till-empty)
Retirement message received
qman_test_high finished
[...]
```

Running the BMan threshold test;

```
[...]
bman_test_thresh: start
```

```
bman_test_thresh: buffers are in
thread 0: starting
thread 1: starting
thread 2: starting
thread 3: starting
thread 4: starting
thread 5: starting
thread 6: starting
thread 7: starting
thread 0: draining...
cb_depletion: bpid=62, depleted=2, cpu=0
cb_depletion: bpid=62, depleted=2, cpu=1
cb_depletion: bpid=62, depleted=2, cpu=2
cb_depletion: bpid=62, depleted=2, cpu=3
cb_depletion: bpid=62, depleted=2, cpu=4
cb_depletion: bpid=62, depleted=2, cpu=5
cb_depletion: bpid=62, depleted=2, cpu=6
cb_depletion: bpid=62, depleted=2, cpu=7
thread 0: draining done.
thread 0: exiting
thread 1: exiting
thread 2: exiting
thread 3: exiting
thread 4: exiting
thread 5: exiting
thread 6: exiting
thread 7: exiting
bman_test_thresh: done
[...]
```

Running the QMan hot potato test;

```
[...]
qman_test_hotpotato starting
Creating 2 handlers per cpu...
Number of cpus: 8, total of 16 handlers
Sending first frame
Received final (8th) frame
qman_test_hotpotato finished
[...]
```

If the self-tests detect any errors, they will `panic()` the kernel immediately, so if the kernel gets beyond the QMan/BMan self-tests then the tests passed.

Changes since SDKv1.7

- Add new CEETM APIs to allow the user to set the LNI and channel shaping rate in bps format directly.

  See API reference manual for details.
- Add new CEETM API to check the LNI/channel shaping enablment.
- Add new CEETM API to set the CQ's weight in the ratio format. The ratio to weight code conversion is done inside this new API
- Add new CEETM API to drain the Class Queue till empty, and call this API inside the function to release CQ so that there will be no frames left in CQ when it is released

Known Bugs, Limitations, or Technical Issues

- QMan and BMan portals are core affine, with each core assigned one QMan and one BMan portal. This assignment also assumes that interrupts associated with these portals are directed to the same cores. This is a fundamental assumption for QMan/BMan drivers. Users should not change the core affinity of portal interrupts for any reason as this would cause the portal to become non-functional and possibly cause a kernel crash.
- QMan enqueue command ring (EQCR) stashing is only supported on QMan rev >= 3.0 on T4240 and B4860, not on other QorIQ Pxxxx silicon. Therefore, run-to-completion software that is attempting an enqueue operation (ie. it is not providing any WAIT flag) should implement its own “back off” after an enqueue returns an EBUSY return code (a 1000-cycle back off is a recommended guide-line). Failure to do so can consume excessive memory bandwidth and reduce overall throughput supported by the system.

7.4.2.2 QMan BMan API Reference

7.4.2.2.1 About this document

This document describes drivers for the Queue Manager and Buffer Manager hardware blocks underlying the datapath architecture within the NXP QorIQ multicore SoC's (P4080, P3041, P5020). There are also explanations given to various aspects of the QMan and BMan hardware itself, insofar as this is essential knowledge in using these devices effectively. The driver descriptions cover some driver implementation details, usage details (loading and configuring), but the main emphasis is on the programming interfaces (APIs) exposed by these drivers.

7.4.2.2.1.1 Suggested Reading

1. *QorIQ* P4080, P3041, P5020 Reference Manuals (P4080RM, P3041RM, P5020RM)
2. Power.org Standard for Embedded Power Architecture Platform Requirements (ePAPR). power.org, 2008.

7.4.2.2.2 Introduction to the Queue Manager and the Buffer Manager

The Queue Manager (QMan) and Buffer Manager (BMan) devices each expose two interfaces to software control. One interface is the Configuration and Control Status Register map (CCSR), which provides global configuration of the device, registers related to global device errors, performance, statistics, debugging, etc. The other interface is the CoreNet interface, which provides a memory map with multiple "portals" located in separable sub-regions for independent/parallel run-time use of the devices.

7.4.2.2.2.1 O/S specifics

The software described in this document is targeted to the Linux kernel and Linux user-space (USDPAA) system targets. However, only Linux supports operating as the controller for the devices, so all interfaces related to CCSR access are Linux-only. Also, remember platform-specific considerations when working with the interfaces described here. See Operating system specifics on page 691for more details.

7.4.2.2.3 Buffer Manager

7.4.2.2.3.1 BMan Overview

7.4.2.2.3.1.1 Buffer Manager's Function

The QorIQ Buffer Manager (BMan) SoC block manages pools of buffers for use by software and hardware in the “Datapath” architecture. On the P4080, BMan maintains state for 64 “Buffer Pools”, which are typically used by hardware blocks for constructing output data for returning to software, where software can not (or does not wish to) pre-allocate an output descriptor. *For non-P4080 specifications, refer to the appropriate QorIQ SoC Reference Manual.*

In particular;

1. provides an efficient use of buffer resources because the output will only occupy as many buffers as required (whereas pre-allocation must provide for the worst-case scenario each time if it wishes to avoid truncation and information-loss),
2. software does not need to provision resources for every queued operation nor handle the complications of recycling unused output buffers, etc.,

3. the footprint for buffer resources for a variety of different flows (and even different guest operating systems) can be "pooled".

With respect to "buffers", BMan really acts as an allocator of any 48-bit tokens the user wishes - BMan does not interpret these tokens at all, it is only the software and hardware blocks that use BMan that may assume these to be memory addresses. In many cases, the BMan acquire and release interfaces are likely to be more efficient than software-managed allocators due to the proximity of BMan's corenet-based interfaces to each CPU and its on-board caching and pre-fetching of pool data. Possible examples include; a BMan-oriented page-allocator for operating system memory-management, a "frame queue" allocator to manage unused QMan frame queue descriptors (FQD), etc. In particular, the frame queue example provides a simple mechanism for sharing a range of frame-queue IDs across different partitions/operating systems in a virtualized environment without needing inter-partition communications in software.

7.4.2.2.3.1.2 BMan's interfaces

The BMan block has a CCSR register space and interrupt line associated with the block for global configuration and management, specifically;

- the private system memory range (invisible to software) needed by BMan,
- software and hardware depletion interrupt thresholds for each pool,
- device error handling uses the global interrupt line and the CCSR register space contains error-capture and error-status registers.

The BMan block also exposes a Corenet memory space for low-latency interaction by the multiple SoC cores, and this corenet region is divided into a geometry of "portals" to allow independent access to BMan functionality in a partitioned (and/or virtualized) environment. Each portal consists of one 16KB cache-enabled and one 4KB cache-inhibited sub-range of the Corenet region, as well as a per-portal interrupt line. There are a variety of possible reasons for using distinct portals;

- for partitioning between distinct guest operating systems,
- to dedicate a portal for each CPU to reduce locking and improve cache-affinity,
- to make distinct portal configurations available,
- to give certain applications their own portal rather than enforcing a mux/demux layer to share a portal between applications,
- [etc.]

Each portal presents the following BMan functionality;

- a "release command ring" (RCR), a pipelined mechanism for software to hardware commands that release buffers to BMan-managed buffer pools,
- a "management command" interface (MC), a low-latency command/response interface for acquiring buffers from buffer pools, and querying the status of all buffer pools,
- an interrupt line and associated status, disable, enable, and inhibit registers.

These portal interfaces will be described in more detail in their respective sections.

7.4.2.2.3.2 BMan configuration interface

The BMan configuration interface is an encapsulation of the BMan CCSR register space and the global/error interrupt line. Whereas BMan portals provide independent channels for accessing BMan functionality, the configuration interface represents the BMan device itself. The BMan configuration interface is presently limited to the device-tree node that represents it, with one exception: an API exists to set per-buffer-pool depletion thresholds. This API is only available in the linux control-plane - that is, a kernel compiled with BMan control support that has the BMan CCSR device-tree node present. In a hypervisor scenario, this implies that only the control-plane linux guest OS can set buffer pool depletion thresholds.

7.4.2.2.3.2.1 BMan Device-Tree Node

The BMan device tree node represents the BMan device and its CCSR configuration space. When a linux kernel has BMan control support compiled in, it will react to this device tree node by configuring and managing the BMan device. The device-tree node sits within the CCSR node ("soc") and is of the following form;

```
soc@fe000000 {
        [...]
        bman: bman@31a000 {
                compatible = "fsl,bman";
                reg = <0x31a000 0x1000>;
                fsl,liodn = <0x20>;
        };
        [...]
  };
```

'compatible' and 'reg' are standard ePAPR properties.

7.4.2.2.3.2.1.1 Free Buffer Proxy Records

As previously mentioned, BMan buffer pools needn't be used only for managing memory buffers, but in fact can manage pools of arbitrary 48-bit token values, whatever those tokens might represent. This is possible because BMan never uses those token values as memory locations - all management of buffer pools is maintained in memory that is private to the BMan block. Specifically, BMan uses some internal memory together with a private range of contiguous system memory for backing store. The internal units of the backing store memory are called "free buffer proxy records" (FBPRs), each of which occupies a 64-byte cacheline of memory, and can hold 8 tokens.

The current driver implementation allows this memory resource to be specified via the 'fsl,bman-fbpr' device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. The 'fsl,bman-fbpr' property specifies a 2-tuple of address and size, specifying the physical address range to assign to BMan. The example given configures 16MB for FBPR memory **(262,144 FBPR entries or 2,097,152 buffer tokens)**. These elements are expressed as 64-bit values, so take two cells each:

```
fsl,fbpr = <0x0 0x20000000 0x0 0x01000000>;
```

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the linux control-plane OS, it will attempt to reserve the range against use by the OS.

NOTE

For all BMan and QMan private memory resources, the alignment of the memory region must match its size.

7.4.2.2.3.2.1.2 Logical I/O Device Number (BMan)

Reads and writes to BMan's FBPR memory are subject to processing by the PAMU IO-MMU configuration of the SoC. In particular, BMan has an LIODN (Logical I/O Device Number) register setting that will be used by PAMU authorise and possibly translate memory accesses. The bootloader (u-boot) will program BMan's LIODN register and it will add this value as the "fsl,liodn" property before passing it along to the booted software.

```
fsl,liodn = <0x20>;
```

This property is only used by the hypervisor, in order to ensure that any translation between guest physical and real physical memory for the linux guest OS is similarly applied to BMan transactions. If linux is booted natively (no hypervisor), then the PAMU is either left in bypass mode or it is configured without translation. In any case the LIODN is of little practical importance to the configuration or use of BMan driver software.

7.4.2.2.3.2.2 Buffer Pool Node

The BMan buffer pool device tree node represents one of a BMan device's buffer pools and its associated configuration. When a linux kernel has BMan control support compiled in, it will react to this device tree node by configuring and managing the BMan buffer pool, in particular the pool will be marked as reserved by the driver so that it is not available for dynamic assignment. The device-tree nodes usually sit within a BMan portals parent node ("bman-portals") and is of the following form;

```
                bman-portals@f4000000 {

                        [...]

                        buffer-pool@0 {

                                compatible = "fsl,bpool";

                               fsl,bpid = <0x0>;

                              fsl,bpool-cfg = <0x0 0x100 0x0 0x1 0x0 0x100>;

                                fsl,bpool-thresholds = <0x8 0x20 0x0 0x0>;

                        };

                        [...]

         };
```

7.4.2.2.3.2.2.1 Buffer Pool ID

The BMan device in QorIQ P4080 supports 64 hardware managed buffer pools, so valid IDs range from 0 to 63. For non-P4080 specifications, refer to the appropriate QorIQ SoC Reference Manual. The above example configures buffer pool 0, which is used by the QMan driver as an inter-partition allocator of unused QMan Frame Queue IDs;

```
        fsl,bpid = <0x0>;
```

Buffer pool nodes in the device-tree indicate that the corresponding buffer pool IDs are reserved, ie. that they are not to be used for ad-hoc allocation of unused pools.

7.4.2.2.3.2.2.2 Seeding Buffer Pools

It is also possible to have the control plane linux BMan driver seed the buffer pool with an arbitrary arithmetic sequence of values, using the "fsl,bpool-cfg" property. This property is a 3-tuple of 64-bit values (each taking 2 cells) defining the arithmetic sequence; the count, the increment, and the base.

```
        fsl,bpool-cfg = <0x0 0x100 0x0 0x1 0x0 0x100>;
```

In this example, the QMan FQ allocator implemented using BMan buffer pool ID 0 is seeded with 256 FQIDs in the range [256...511].

7.4.2.2.3.2.2.3 Depletion Thresholds

Each of the 64 buffer pools has CCSR registers related to depletion-handling. A pool is considered "depleted" once the number of buffers in that pool crosses a "depletion-entry" threshold from above, and this ends when the number of buffers subsequently crosses a "depletion-exit" threshold from below (the depletion-exit threshold should be higher than the depletion-entry threshold).

Each pool maintains two independent depletion states - one for software use and another for hardware blocks. Hardware blocks (like CAAM, FMan, PME) use the hardware depletion state primarily for the purpose of implementing push back (e.g. by stalling input-processing, issuing "pause frames", etc). There is a depletion-entry and -exit threshold for each buffer pool related to this hardware depletion state. The software depletion state serves two possible purposes - one is to allow software to implement push back too. The other use of software depletion thresholds is to allow software to manage "replenishment" of buffer pools. It is software that seeds buffer pools with blocks of memory initially and if desired, it can also use this mechanism to selectively provide additional blocks at run-time during depletion.

```
                        fsl,bpool-thresholds = <0x8 0x20 0x0 0x0>;
```

Here, software depletion thresholds have been set for the buffer pool used for the FQ allocator, but hardware depletion thresholds are disabled (the pool is for software use only). The pool will enter depletion when it drops below 8 "buffers" (in this case, FQIDs), and exit depletion when it rises above 32.

7.4.2.2.3.2.3 BMan Portal Device-Tree Node

The BMan Corenet portal interface in QorIQ P4080 provides up to 10 distinct memory-mapped interfaces for use by software to interact efficiently with BMan functionality. Specifically, each portal provides the following sub-interfaces; RCR (Release Command Ring), MC (Management Command), and ISR (Interrupt Status Register). For non-P4080 specifications, refer to the appropriate QorIQ SoC Reference Manual.

The BMan driver determines the available corenet portals from the device tree. The portal nodes are at the physical address scope (unlike the device-tree node for the BMan device itself, which is within the “soc” physical address node that represents CCSR). These nodes indicate the physical address ranges of the cache-enabled and cache-inhibited sub-regions of the portal (respectively), and look something like the following;

```
        bman-portal@0 {
                compatible = "fsl,bman-portal";
                reg = <0xe4000000 0x4000 0xe4100000 0x1000>;
                interrupts = <0x69 2>;
                interrupt-parent = <&mpic>;
                cell-index = <0x0>;
                cpu-handle = <&cpu3>;
        };
```

The most note-worthy property is "cpu-handle", which is used to express an affinity/association between the given BMan portal and the CPU represented by the referenced device-tree node.

7.4.2.2.3.2.3.1 Portal Initialization (BMan)

The driver is informed of the BMan portals that are available to it via the device-tree passed to the system from the boot process. For those portals that aren’t reserved for USDPAA usage via the “fsl,usdpaa-portal” property, it will automatically create TLB entries to map the BMan portal corenet sub-regions as cpu-addressable and cache-inhibited or cache-enabled as appropriate.

The BMan driver will automatically associate initialised BMan portals with the CPU to which they are configured, only a one-per-CPU basis (if multiple portals are configured for the same CPU, only one is used). The purpose of this is to provide a canonical portal that software can use for whichever CPU it is running on, with the advantages of a cpu-affine interface being improved cache-locality and reduced locking. This requires that each CPU have at least one portal device-tree node dedicated to it using the “cpu-handle” property.

7.4.2.2.3.2.3.2 Portal sharing

If there are CPUs that have no affine portal associated with them (for example if most portals have been reserved for USDPAA use), then the driver will select the highest-index portal to be configured for “sharing” with the CPUs that have no affine portal, otherwise called “slave CPUs” in this document. In this mode of operation, a coarser locking scheme is used for the portal in order to properly synchronise use by more than one CPU.

One key point to understand with portal sharing is that hardware-instigated portal events will continue to be processed only by the CPU to which the portal is affine, they are not shared. One consequence of this is that slave CPUs can not use *_irqsource_*() APIs to alter the interrupt-vs-polling state of the portal, nor can they call *_poll_*() APIs to perform run-to-completion servicing of the portal. The sharing of the portal is only to allow software-instigated portal functionality to be available to slave CPUs, such as creating and manipulating objects, performing commands, etc.

7.4.2.2.4 BMan CoreNet portal APIs

The following sections describe interfaces provided by the BMan driver for manipulating portals, as defined in BMan Portal Device-Tree Node on page 626.

7.4.2.2.4.1 BMan High-Level Portal Interface

7.4.2.2.4.1.1 Overview (BMan)

The high-level portal interface provides management and encapsulation of a portal hardware interface. The operations performed on the portal are co-ordinated internally, hiding the user from the I/O semantics, and allowing multiple users/contexts to share portals without collaboration between them. This interface also provides an object representation for buffer pools, with optional assists for cases where the user wishes to track depletion entry and exit events.

This interface provides locking and arbitration of portal operations from multiple software contexts and/or threads (ie. the portal is shared). In cases where a resource is busy, the interface also gives callers the option of blocking/sleeping until the resouce is available. In any case where sleeping is an option, the caller can also specify whether the sleep should be interruptible.

NOTE

Support for blocking/sleeping is limited to linux, it is not available on run-to-completion systems such as USDPAA.

7.4.2.2.4.1.2 Portal management (BMan)

The portal management API provides bman_affine_cpus(), which returns a mask that indicates which CPUs have auto-initialized portals associated with them. See BMan Portal Device-Tree Node on page 626. All other BMan API functions must be executed on CPUs contained within this mask, and any interactions they require with h/w will be performed on the corresponding portals.

```
/**
 * bman_affine_cpus - return a mask of cpus that have portal access
 */
const cpumask_t *bman_affine_cpus(void);
```

7.4.2.2.4.1.2.1 Modifying interrupt-driven portal duties (BMan)

Portals have various servicing duties they must perform in reaction to hardware events. The portal management API allows applications to control which of these duties/events are triggered by interrupt-handling versus those which are performed at the application’s explicit request via bman_poll(). If portal-sharing is in effect (see Portal sharing on page 626), these APIs won’t succeed when called from a slave CPU.

```
#define BM_PIRQ_RCRI    0x00000002      /* RCR Ring (below threshold) */
#define BM_PIRQ_BSCN    0x00000001      /* Buffer depletion State Change */
/**
 * bman_irqsource_get - return the portal work that is interrupt-driven
 *
 * Returns a bitmask of BM_PIRQ_**I processing sources that are currently
 * enabled for interrupt handling on the current cpu's affine portal. These
```

```
 * sources will trigger the portal interrupt and the interrupt handler (or a
 * tasklet/bottom-half it defers to) will perform the corresponding processing
 * work. The bman_poll_***() functions will only process sources that are not in
 * this bitmask. If the current CPU is sharing a portal hosted on another CPU,
 * this always returns zero.
 */
u32 bman_irqsource_get(void);
/**
 * bman_irqsource_add - add processing sources to be interrupt-driven
 * @bits: bitmask of BM_PIRQ_**I processing sources

 * Adds processing sources that should be interrupt-driven, (rather than
* processed via bman_poll_***() functions). Returns zero for success, or
* -EINVAL if the current CPU is sharing a portal hosted on another CPU.
*/
int bman_irqsource_add(u32 bits);
/**
* bman_irqsource_remove - remove processing sources from being interrupt-driven
* @bits: bitmask of BM_PIRQ_**I processing sources
*
* Removes processing sources from being interrupt-driven, so that they will
* instead be processed via bman_poll_***() functions. Returns zero for success,
* or -EINVAL if the current CPU is sharing a portal hosted on another CPU. */
int bman_irqsource_remove(u32 bits);
```

7.4.2.2.4.1.2.2 Processing non-interrupt-driven portal duties (BMan)

If portal-sharing is in effect (see Portal sharing on page 626), these APIs won't succeed when called from a slave CPU.

```
/**
 * bman_poll_slow - process anything that isn't interrupt-driven.
 *
 * This function does any portal processing that isn't interrupt-driven. NB,
 * unlike the legacy wrapper bman_poll(), this function will deterministically
 * check for the presence of portal processing work and do it, which implies
 * some latency even if there's nothing to do. The bman_poll() wrapper on the
 * other hand (like the qman_poll() wrapper) attenuates this by checking for
 * (and doing) portal processing infrequently. Ie. such that qman_poll() and
 * bmna_poll() can be called from core-processing loops. Use bman_poll_slow()
 * when you yourself are deciding when to incur the overhead of processing. If
* the current CPU is sharing a portal hosted on another CPU, this function will
* return -EINVAL, otherwise returns zero for success.
*/
int bman_poll_slow(void);
/**
 * bman_poll - process anything that isn't interrupt-driven.
 *
 * Dispatcher logic on a cpu can use this to trigger any maintenance of the
 * affine portal. This function does whatever processing is not triggered by
 * interrupts. This is a legacy wrapper that can be used in core-processing
 * loops but mitigates the performance overhead of portal processing by
 * adaptively bypassing true portal processing most of the time. (Processing is
 * done once every 10 calls if the previous processing revealed that work needed
 * to be done, or once very 1000 calls if the previous processing revealed no
 * work needed doing.) If you wish to control this yourself, call
 * bman_poll_slow() instead, which always checks for portal processing work.
 */
void bman_poll(void);
```

7.4.2.2.4.1.2.3 Recovery support (BMan)

Note that the following functions require the BMan portal to have been initialized in "recovery mode", which is not possible with the current release. As such, these functions are for future use only (and documented here only because they're declared in the API header).

```
/**
 * bman_recovery_cleanup_bpid  - in recovery mode, cleanup a buffer pool
 */
int bman_recovery_cleanup_bpid(u32 bpid);
/**
 * bman_recovery_exit - leave recovery mode
 */
int bman_recovery_exit(void);
```

7.4.2.2.4.1.2.4 Determining if the release ring is empty

```
/**
 * bman_rcr_is_empty - Determine if portal's RCR is empty
 *
 * For use in situations where a cpu-affine caller needs to determine when all
 * releases for the local portal have been processed by BMan but can't use the
 * BMAN_RELEASE_FLAG_WAIT_SYNC flag to do this from the final bman_release().
 * The function forces tracking of RCR consumption (which normally doesn't
 * happen until release processing needs to find space to put new release
 * commands), and returns zero if the ring still has unprocessed entries,
 * non-zero if it is empty.
 */
int bman_rcr_is_empty(void);
```

7.4.2.2.4.1.3 Pool Management

To work with BMan buffer pools, a pool object must be created. As explained in Depletion State on page 632, the pool may be created with the BMAN_POOL_FLAG_DEPLETION flag and corresponding depletion-entry/exit callbacks if the owner wishes to be notified of changes in the pool's depletion state. Creation of the pool object can also modify the pool's depletion entry and exit thresholds with the BMAN_POOL_FLAG_THRESH flag, so long as the BMAN_POOL_FLAG_DYNAMIC_BPID flag is specified (which will allocate an unreserved BPID) and when running in the control-plane (where reserved BPIDs are tracked). Depletion thresholds for reserved BPIDs can be set in the device-tree within the nodes that reserve them, so support for setting them in the API is not provided. The pool object can also maintain an internal buffer stockpile to optimize releases and acquires of buffers by specifying the BMAN_POOL_FLAG_STOCKPILE flag - actual releases to and acquires from h/w will only occur when the stockpile needs flushing or replenishing, ensuring that the interactions with hardware occur less often and are always optimized to release/acquire the maximum number of buffers at once. If a pool object is being freed and it has been configured to use stockpiling, a flush operation must be performed on the pool object. This will ensure that all buffers in the stockpile are flushed to h/w. The pool object can then be freed. The stockpiling option is recommended wherever possible. One implementation note is that applications will sometimes want to create multiple pool objects for the same BPID in order to have one for each CPU (for performance reasons) - this means that each pool object will have its own stockpile. As a consequence, to drain a buffer pool empty would require that all pool objects for that BPID be drained independently (whereas without stockpiling enabled, only one pool object needs to be drained).

```
struct bman_pool;
/* This callback type is used when handling pool depletion entry/exit. The
 * 'cb_ctx' value is the opaque value associated with the pool object in
 * bman_new_pool(). 'depleted' is non-zero on depletion-entry, and zero on
 * depletion-exit. */
typedef void (*bman_cb_depletion)(struct bman_portal *bm,
                          struct bman_pool *pool, void *cb_ctx, int depleted);
/* Flags to bman_new_pool() */
```

```
#define BMAN_POOL_FLAG_NO_RELEASE    0x00000001 /* can't release to pool */
#define BMAN_POOL_FLAG_ONLY_RELEASE  0x00000002 /* can only release to pool */
#define BMAN_POOL_FLAG_DEPLETION     0x00000004 /* track depletion entry/exit */
#define BMAN_POOL_FLAG_DYNAMIC_BPID  0x00000008 /* (de)allocate bpid */
#define BMAN_POOL_FLAG_THRESH        0x00000010 /* set depletion thresholds */
#define BMAN_POOL_FLAG_STOCKPILE     0x00000020 /* stockpile to reduce hw ops */
/* This struct specifies parameters for a bman_pool object. */
struct bman_pool_params {
        /* index of the buffer pool to encapsulate (0-63), ignored if
         * BMAN_POOL_FLAG_DYNAMIC_BPID is set. */
        u32 bpid;
        /* bit-mask of BMAN_POOL_FLAG_*** options */
        u32 flags;
        /* depletion-entry/exit callback, if BMAN_POOL_FLAG_DEPLETION is set */
        bman_cb_depletion cb;
        /* opaque user value passed as a parameter to 'cb' */
        void *cb_ctx;
        /* depletion-entry/exit thresholds, if BMAN_POOL_FLAG_THRESH is set. NB:
         * this is only allowed if BMAN_POOL_FLAG_DYNAMIC_BPID is used *and*
         * when run in the control plane (which controls BMan CCSR). This array
         * matches the definition of bm_pool_set(). */
        u32 thresholds[4];
};
/**
 * bman_new_pool - Allocates a Buffer Pool object
 * @params: parameters specifying the buffer pool behavior
 *
 * Creates a pool object for the given @params. A portal and the depletion
 * callback field of @params are only used if the BMAN_POOL_FLAG_DEPLETION flag
 * is set. NB, the fields from @params are copied into the new pool object, so
 * the structure provided by the caller can be released or reused after the
 * function returns.
 */
struct bman_pool *bman_new_pool(const struct bman_pool_params *params);
/**
 * bman_free_pool - Deallocates a Buffer Pool object
 * @pool: the pool object to release
 */
void bman_free_pool(struct bman_pool *pool);
/**
 * bman_flush_stockpile - Flush stockpile buffer(s) to the buffer pool
 * @pool: the buffer pool object the stockpile belongs
 * @flags: bit-mask of BMAN_RELEASE_FLAG_*** options
 *
 * Adds stockpile buffers to RCR entries until the stockpile is empty.
 * The return value will be a negative error code if a h/w error occured.
 * If BMAN_RELEASE_FLAG_NOW flag is passed and RCR ring is full,
 * -EAGAIN will be returned.
 */
int bman_flush_stockpile(struct bman_pool *pool, u32 flags);
/**
 * bman_get_params - Returns a pool object's parameters.
 * @pool: the pool object
 *
 * The returned pointer refers to state within the pool object so must not be
 * modified and can no longer be read once the pool object is destroyed.
 */
const struct bman_pool_params *bman_get_params(const struct bman_pool *pool);
/**
 * bman_query_free_buffers - Query how many free buffers are in buffer pool
```

```
 * @pool: the buffer pool object to query
 *
 * Return the number of the free buffers
 */
u32 bman_query_free_buffers(struct bman_pool *pool);
/**
 * bman_update_pool_thresholds - Change the buffer pool's depletion thresholds
 * @pool: the buffer pool object to which the thresholds will be set
 * @thresholds: the new thresholds
 */
int bman_update_pool_thresholds(struct bman_pool *pool, const u32 *thresholds);
```

7.4.2.2.4.1.4 Releasing and Acquiring Buffers

The following API functions allow applications to release buffers to a pool and acquire buffers from a pool. Note that the various "WAIT" flags for bman_release() are only available on linux.

```
/* Flags to bman_release() */
#define BMAN_RELEASE_FLAG_WAIT       0x00000001 /* wait if RCR is full */
#define BMAN_RELEASE_FLAG_WAIT_INT   0x00000002 /* if we wait, interruptible? */
#define BMAN_RELEASE_FLAG_WAIT_SYNC  0x00000004 /* if wait, until consumed? */
/**
 * bman_release - Release buffer(s) to the buffer pool
 * @pool: the buffer pool object to release to
 * @bufs: an array of buffers to release
 * @num: the number of buffers in @bufs (1-8)
 * @flags: bit-mask of BMAN_RELEASE_FLAG_*** options
 *
 * Releases the specified buffers to the buffer pool. If stockpiling is
 * enabled, this may not require a release command to be issued via the RCR
 * ring, otherwise it certainly will. If the RCR ring is full, the function
 * will return -EBUSY unless BMAN_RELEASE_FLAG_WAIT is selected, in which case
 * it will sleep waiting for space to become available in RCR. If
 * BMAN_RELEASE_FLAG_WAIT_SYNC is also specified then it will sleep until
 * hardware has processed the command from the RCR (otherwise the same
 * information can be obtained by polling bman_rcr_is_empty() until it returns
 * TRUE). If the BMAN_RELEASE_FLAG_WAIT_INT is set), then any sleeps will be
 * interruptible. If it is interrupted before producing the release command, it
 * returns -EINTR. Otherwise, it will return zero to indicate the release was
 * successfully issued. (In the case of interruptible sleeps and WAIT_SYNC,
 * check signal_pending() upon return to determine whether the wait was
 * interrupted.)
 */
int bman_release(struct bman_pool *pool, const struct bm_buffer *bufs,
                                u8 num, u32 flags);
/**
 * bman_acquire - Acquire buffer(s) from a buffer pool
 * @pool: the buffer pool object to acquire from
 * @bufs: array for storing the acquired buffers
 * @num: the number of buffers desired (@bufs is at least this big)
 *
 * Acquires buffers from the buffer pool. If stockpiling is enabled, this may
 * not require an acquire command to be issed via the MC interface, otherwise
 * it certainly will. The return value will be the number of buffers obtained
 * from the pool, or a negative error code if a h/w error or pool starvation
 * was encountered.
 */
```

```
int bman_acquire(struct bman_pool *pool, struct bm_buffer *bufs, u8 num,
                        u32 flags);
```

7.4.2.2.4.1.5 Depletion State

It is possible for portals to track depletion state changes to any of the 64 buffer pools supported in BMan. As described in Pool Management on page 629, a pool object can invoke callbacks to convey depletion-entry and depletion-exit events if created with the BMAN_POOL_FLAG_DEPLETION flag.

Conversely, software can issue a portal management command to obtain a snapshot of the depletion and availability status of all BMan 64 pools at once, which is what the following interface does. Here "availability" implies that the pool is not completely empty. Depletion on the other hand is relative to the pools depletion-entry and exit-thresholds. The state of all 64 buffer pools is represented by the following structure types, accessor macros, and bman_query_pools() API;

```
struct bm_pool_state {
          [...]
};
/**
 * bman_query_pools - Query all buffer pool states
 * @state: storage for the queried availability and depletion states
 */
int bman_query_pools(struct bm_pool_state *state);
/* Determine the "availability state" of BPID 'p' from a query result 'r' */
#define BM_MCR_QUERY_AVAILABILITY(r,p) [...]
/* Determine the "depletion state" of BPID 'p' from a query result 'r' */
#define BM_MCR_QUERY_DEPLETION(r,p) [...]
```

7.4.2.2.5 Queue Manager

7.4.2.2.5.1 QMan Overview

7.4.2.2.5.1.1 Queue Manager's Function

The QorIQ Queue Manager (QMan) SoC block manages the movement of data (“frames”) along uni-directional flows (“frame queues”) between different software and hardware end-points (“portals”). This allows software instances to communicate with other software instances and/or datapath hardware blocks (CAAM, PME, FMan) using a hardware-managed queueing mechanism. QMan provides a variety of features in the way this data movement can be managed, including tail-drop or weighted-red congestion/flow-control, congestion group depletion notification, order restoration, and order preservation.

It is beyond the scope of this document to fully explain all the QMan-related notions that are essential to using datapath functionality effectively. But unlike the BMan reference, we will cover at least some of the basic elements here that are fundamental to the software interface, because QMan is more complicated than BMan and some simplistic definitions can be helpful as a place to start. For any more information about what QMan does and how it behaves, please consult the appropriate QorIQ SoC Reference Manual.

7.4.2.2.5.1.2 Frame Descriptors

Frames are represented by "frame descriptors" (or "FD"s) which are 16-byte structures consisting of fields to describe;

- contiguous or scatter-gather data,
- a 32-bit per-frame-descriptor token value (called "cmd/status" because of its common usage in processing data to/from hardware blocks),
- trace-debugging bits,
- a partition ID, used for virtualizing memory access to frame data by datapath hardware blocks (CAAM, PME, FMan),
- a BMan buffer pool ID, used to identify frames whose buffers are sourced from (or are to be recycled to) a BMan buffer pool.

A third ("nested") mode of the scatter-gather representation allows a frame-descriptor to reference more than one frame - this is referred to as a *compound frame*, and is a mechanism for creating an indissociable binding of more than one data descriptor, eg. this is used when sending an input descriptor to PME or CAAM and providing an output descriptor to go with it.

Frame descriptors that are under QMan's control reside in QMan-private resources, comprised of dedicated on-board cache as well as system memory assigned to QMan on initialization. When frames are enqueued to (and dequeued from) frame queues by QMan on behalf of software portals or hardware blocks, the frame descriptor fields are copied in to (and out of) these QMan-private resources.

As with BMan not caring whether the 48-bit tokens it manages are real buffer addresses or not, the same is mostly true for QMan with respect to the frame descriptors it manages. QMan ignores the memory addresses present in the frame descriptor, unless it is dequeued via a portal configured for data stashing and is dequeued from a frame queue that is configured for frame data (or annotation) stashing. However QMan always pays attention to the length field of frame descriptors. In general, the only field that can be safely used as a "pass-through" value without any QMan consequences is the 32-bit cmd/status field.

7.4.2.2.5.1.3 Frame Queue Descriptors (QMan)

Frame queues are uni-directional queues of frames, where frames are enqueued to the tail of the frame queue and dequeued from the head. A frame queue is represented in QMan by a "frame queue descriptor" (or "FQD"), and these reside in a private system memory resource configured for QMan on initialization. A frame queue is referred to by a "frame queue identifier" (or "FQID"), which is literally the index of that FQD within QMan's memory resource. As such, FQIDs form a global name-space, even in an otherwise virtualized environment, so two entities of software can not simultaneously use the same FQID for different purposes.

7.4.2.2.5.1.4 Work Queues

Work queues (or "WQ"s) are uni-directional queues of "scheduled" frame queues. We will see shortly what is meant here by a "scheduled" frame queue, but suffice it to say that QMan supports a fixed collection of work queues, to which QMan appends frame queues when they are due to be serviced. To summarize, multiple FDs can be linked to a single FQ, and multiple FQs can be linked to a single WQ.

7.4.2.2.5.1.5 Channels

A channel is a fixed, hardware-defined association of 8 work queues, also thought of as "priority work queues". This grouping is convenient in that QMan provides sophisticated prioritization support for dequeueing from entire channels rather than specific work queues. Specifically, the 8 work queues within a channel are divided into 3 tiers according to QMan's "class scheduler" logic - work queues 0 and 1 form the high-priority tier and are treated with a strict priority semantic, work queues 2, 3, and 4 form the medium-priority tier and are treated with a weighted interleaved round-robin semantic, and work queues 5, 6, and 7 form the low-priority tier and are also treated with a weighted interleaved round-robin semantic. Apart from the top-tier, the weighting within and between the other two tiers is programmable.

7.4.2.2.5.1.6 Portals

A QMan portal is similar in nature to a BMan portal. There are hardware portals (also called "direct connect portals", or "DCP"s) that allow QMan to be used by other hardware blocks, and there are software portals that allow QMan to be used by logically separated units of software. A software portal consists of two sub-regions of QMan's corenet region, in precisely the same way as with BMan.

7.4.2.2.5.1.7 Dedicated Portal Channels

Each software portal has its own dedicated channel (of 8 work queues), that only it may dequeue from. As a shorthand, one sometimes says that a frame queue is "scheduled to a portal", when what is really meant is that the frame queue is scheduled to a work queue within that portal's *dedicated channel*. Hardware portals also have their own dedicated channels, though sometimes more than one (FMan blocks have multiple dedicated channels).

7.4.2.2.5.1.8 Pool Channels

There are also 15 "pool channels" from which any software portal can dequeue - this is typically used for load-balancing or load-spreading.

7.4.2.2.5.1.9 Portal Sub-Interfaces

Each portal exposes cache-inhibited and cache-enabled registers that can be read and/or written by software to achieve various ends. With some necessary exceptions, the software interface hides most of these details. However an important conceptual point regarding portals is that they have essentially four decoupled sub-interfaces;

- EQCR (EnQueue Command Ring), this is an 8-cacheline ring containing commands from software to QMan. These commands perform enqueues of frame descriptors to frame queues.
- DQRR (DeQueue Response Ring), this is a 16-cacheline ring containing dequeue processing results from QMan to software. These entries usually contain a frame descriptor (except when the dequeue action produced no valid frame descriptor) as well as status information about the dequeue action, the frame queue being dequeued from, and other context for software's use. This ring is unique in that QMan can be configured to stash new ring entries to processor cache, rather than relying on software to (pre)fetch ring entries into cache explicitly.
- MR (Message Ring), this is an 8-cacheline ring containing messages from QMan to software, most notably for enqueue rejection messages and asynchronous retirement processing events. Unlike DQRR, this ring does not support stashing.
- Management commands, consisting of a Command Register (CR) and two Response Register locations (RR0 and RR1), used for issuing a variety of other commands to QMan. EQCR and DQRR (and to a lesser extent, MR) are intended to provide the communications with QMan that represent the fast-path of data processing logic, and the management command interface is where "everything else happens".

7.4.2.2.5.1.10 Frame queue dequeuing

Enqueuing a frame to a frame queue is an unambiguous mechanism; an enqueue command in the EQCR specifies a frame descriptor and a frame queue ID, and the intention is clear. Dequeuing is more subtle, and falls into two general classes depending on *what* one is dequeuing *from* - these are "scheduled" or "unscheduled" dequeues.

7.4.2.2.5.1.10.1 Unscheduled Dequeues

One can dequeue from a specific frame queue, but that frame queue must necessarily be "idle" - or in QMan terminology, "unscheduled". It is an illegal action to attempt to dequeue directly from a frame queue that is in a "scheduled" state. Specifically, unscheduled dequeues require the frame queue to be in the "Parked" or "Retired" state (described in Frame Queue States on page 636).

7.4.2.2.5.1.10.2 Scheduled Dequeues

Conversely, if a frame queue is "scheduled" then, by definition, management of the frame queue is (until further notice) under QMan's control and may at any point change state according to events within QMan or via actions on other software or hardware portals. So a "scheduled dequeue" does not target a specific FQ, but either a specific WQ or collection of channels. QMan processes scheduled dequeue commands within a portal by selecting from among the non-empty WQs, dequeueing a FQ from that selected WQ, and then dequeuing a FD from that FQ.

QMan portals implement two dequeue command modes, "push" and "pull";

7.4.2.2.5.1.10.3 Pull Mode

The "pull" mode is the less conventional of the two, as it is driven by software writing a dequeue command to a single cache-inhibited register that will, in response, perform a single instance of that command and publish its result to DQRR. This "pull" command (PDQCR - Pull DeQueue Command Register) could generate anywhere between 1 and 3 DQRR entries, and software must ensure that it does not write a new command to PDQCR until it knows at least one of these DQRR entries has been published (otherwise writing a new command could clobber the previous command before QMan has prepared its execution). The PDQCR command register can perform scheduled and unscheduled dequeues.

7.4.2.2.5.1.10.4 Push Mode

The "push" mode is the mode that gives software a familiar "DMA-style" interface, ie. where hardware performs work and fills in a kind of "Rx ring" autonomously. In the case of the QMan portal's DQRR sub-interface, this push mode is driven by two dequeue command registers, one for scheduled dequeues (SDQCR - Static DeQueue Command Register), and one for unscheduled dequeues (VDQCR - Volatile DeQueue Command Register). The reason for the static/volatile terminology (rather than scheduled/unscheduled), as well as the presence of two command registers instead of one, relates to how QMan schedules execution of the dequeue commands.

Unlike "pull" mode, QMan is not prodded by a write to the command register each time a dequeue command should occur, it must autonomously execute commands when appropriate. So it is clear that scheduled dequeues can only be performed when the targetted work queue or channels have Truly Scheduled frame queues available to dequeue from. Note that this is not an issue with "pull" mode, as a scheduled dequeue command can be issued when there are no available frame queues and QMan will simply publish a DQRR entry containing no frame descriptor to mark completion of the command - for "push" mode, this semantic cannot work. When in "push" mode, the QMan portal has a (possibly NULL) scheduled dequeue command for dequeuing from a selection of available channels. QMan executes this command only when there is matching scheduled dequeue work available on one of of the channels - ie. the scheduled dequeue command (for channels) is *static*. If software writes SDQCR with a command to dequeue from a specific WQ, the command is executed only once (like the pull command), at which point it reverts to the static dequeue command for channels.

For unscheduled dequeues, a single Parked or Retired frame queue is identified for dequeuing, and as QMan does not manipulate the state of such frame queues in reaction to enqueue or dequeue activity (ie. there is no "scheduling"), there is no mechanism for QMan to "know" when this frame queue becomes non-empty some time in the future. So like "pull" mode, unscheduled dequeues must be done when explicitly demanded by software, and as such they must also (a) expire after a configurable number of frame descriptors are dequeued from frame queue or once it is empty, and (b) even if the frame queue is already empty, a DQRR entry with no frame descriptor should be used to notify software that the unscheduled dequeue command has expired. Ie. the unscheduled command "goes live" when written and becomes inactive once completed - it is *volatile*. Unlike "pull" mode however, the volatile command can perform more than a single dequeue action, and it can even block or flow-control while active, however it always runs to completion and then stops.

As "push" mode supports two dequeue commands (in fact one of them, SDQCR, encompasses two commands in its own right - it has a persistent channel-dequeue command, and an optional one-shot workqueue-dequeue command can be issued without clobbering it), it is worth pointing out that it can service both at once. The VDQCR command register contains a precedence option that QMan uses to determine whether SDQCR or VDQCR work be favoured in the situation where both are active.

7.4.2.2.5.1.10.5 Stashing to Processor Cache

When dequeueing frame queues and publishing entries in DQRR, QMan provides stashing features that involve prepositioning data in processor cache. The main benefit of hardware-instigated stashing is that the data will already be in cache when the processor needs it, avoiding the need to explicitly prefetch it in advance or stalling the processor to fetch it on-demand. As we will see, there is another benefit in the specific case of DQRR stashing.

Each portal supports two types of stashing, for which distinct PAMU entries are configured.

DLIODN

The DLIODN setting configures PAMU authorization and/or translation of transactions to stash DQRR ring entries as they are produced by QMan. The stashing of DQRR entries is not just a performance tweak, it changes the way driver software operates the portal. Rather than needing to invalidate and prefetch the DQRR cachelines to see (or poll for) new DQRR entries, software can simply reread the cached version until it "magically changes". The stashing transaction is then the only implied traffic across the corenet bus (reducing bandwidth) and it is initiated by hardware at the first instant at which a software-initiated prefetch could have seen anything new (minimum possible latency).

Note that if the driver does not enable DQRR stashing, then it is a requirement to manipulate the processor cache directly, so its run time mode of operation must match device configuration. Note also that if DQRR stashing is used, software can not trust the DQRI interrupt source nor read PI index registers to determine that a new DQRR entry is available, as they may race against the stash transaction. On the other hand, software may use the interrupt source to avoid polling for DQRR production unnecessarily, but it does not guarantee that the first read would show the new DQRR entry.

NOTE

P1023 supports DQRR stashing but since it doesn't have Corenet and PAMU, the DLIODN is not applicable to P1023.

FLIODN

QMan can also stash per-frame-descriptor information, specifically;

1. Frame data, pointed to by the frame descriptor
2. Frame annotations, which is anything prior to the data due to a non-zero offset
3. Frame queue context (for the frame queue from which the frame descriptor was dequeued).

In all cases, the FLIODN setting is used by PAMU to authorize/translate these stashing transactions.

7.4.2.2.5.1.11 Frame Queue States

Frame queues are managed by QMan via state-transitions, and some of these states are of interest to software. From software's perspective, a simplification of the frame queue states is to group them as follows;

- **Out of service:** the frame queue is not in use and must be initialized. Neither enqueues nor dequeues are permitted.
- **Parked:** the frame queue is initialized and in an idle state. Enqueues are permitted, as are unscheduled dequeues, neither of which change the frame queue's state. Scheduled dequeues will not result in dequeues from parked frame queues, as a parked frame queue is never linked to a work queue.
- **Scheduled:** the frame queue has been scheduled, implying that hardware will modify its state as/when relevant events occur. Enqueues are permitted, but unscheduled dequeus are not. This is not a real state, but actually a set of states that a frame queue moves between - as hardware performs these moves internally, it's useful to treat them as one, because changes between them are asynchronous to software. The real states are;
  - **Tentatively Scheduled:** the frame queue is not linked to a work queue (yet), the frame queue must therefore be empty and no retirement or force-eligible command has been issued against the frame queue.
  - **Truly Scheduled:** the frame queue is linked to a work queue, either because it has become non-empty or a force-eligible command has occured.
  - **Active:** the frame queue has been selected by a portal for scheduled dequeue and so is removed from the work queue.
  - **Held Active:** the frame queue is still held by the portal after scheduled dequeuing has been performed, it may yet be dequeued from again, depending on scheduling configuration, priorities, etc.
  - **Held Suspended:** the frame queue is still held by the portal after scheduled dequeuing has been performed but another frame queue has been selected "active" and so no further dequeuing will occur on this frame queue.
- **Retired:** the frame queue is being "closed". A frame queue can be put into the retired state as a means of (a) getting it back under software's control (not under QMan's control nor the control of another hardware block), eg. for closing down "Tx" frame queues, and (b) blocking further enqueues to the frame queue so that it can be drained to empty in a deterministic manner. Enqueues are therefore not permitted in this state. Unscheduled dequeues are permitted, and are the only way to dequeue frames from a frame queue in this state.

See the appropriate QorIQ SoC Reference Manual for more detailed information.

7.4.2.2.5.1.12 Hold active

The QMan portal sub-interfaces are generally decoupled or asynchronous in their operation. For example: The processing of software-produced enqueue commands in EQCR is asynchronous to the processing of dequeue commands into DQRR, and both of these are asynchronous to the production of messages into MR and the processing of management commands.

There is however a specific coupling mechanism between EQCR and DQRR to address a certain class of requirements for datapath processing. Consider first that it is possible for multiple portals to dequeue independently from the same data source,

eg. for the purposes of load-balancing, or perhaps idle-time processing of low-priority work. This could occur because multiple portals issue unscheduled dequeue commands from the same Parked (or Retired) frame queue, or because they issue scheduled dequeue commands that target the same pool channels (or the same specific work queue within a pool channel). So we describe here the "hold active" mechanisms that help maintain some synchronicity of hardware dequeue processing (and optionally software *post*-processing) on multiple portals/CPUs.

The unscheduled dequeue case is not covered by the mechanisms described here - QMan will correctly handle multiple unscheduled dequues from the same frame queue, but the "hold active" mechanisms have no effect in this case. For scheduled dequeues however, there are two levels of "hold active" functionality that can be used for software to synchronise multiple portals dequeuing from the same source.

7.4.2.2.5.1.12.1 Dequeue Atomicity

As described in the previous section ("Frame queue states"), the Active, Held Active, and Held Suspended states are for frame queues that have been selected by a portal for *scheduled* dequeuing. These states imply that the frame queue has been detached from the work queue that it was previously "scheduled" to, but not yet moved to the Parked state nor rescheduled to the Tentatively Scheduled or Truly Scheduled state after the completion of dequeuing.

Normally, a frame queue is rescheduled by QMan as soon as it is done dequeuing, potentially even before the resulting DQRR entries are visible to software. However, if the frame queue has been configured for "Held active" behavior, then this will not happen - the frame queue will remain in the Held Active or Held Suspended state once QMan has finished dequeuing from it. QMan will only reschedule or park the frame queue once software consumes all DQRR entries that correspond to that frame queue - the default behavior is to reschedule, but this "held" state of the frame queue allows software an opportunity to request that the final action for the frame queue be to park it instead.

A consequence of this mechanism is that if a DQRR entry is seen that corresponds to a frame queue configured for "held active" behavior, software implicitly knows that there can be no other (unconsumed) DQRR entry on any other portal for that same frame queue. (Proof: if there was, the frame queue would be currently "held" in that portal and not in this one.) For an SMP system where each core has its own portal, this would obviate the need to (spin)lock software context related to a frame queue when handling incoming frames - the "lock" is implicitly obtained when the DQRR entry is seen, and it is implicitly released when the DQRR entries are consumed. This is what is meant by "dequeue atomicity".

7.4.2.2.5.1.12.2 Parking Scheduled FQs

As noted above in Dequeue Atomicity on page 637, if a FQ is currently "held active" in the portal, software can request that it be move to the Parked state once its final DQRR entry is consumed, rather than rescheduled which is the normal behavior. As we will also see in Force Eligible on page 638, this is not necessarily limited to FQs that are configured for "hold active" behavior, but can also be applied to regular FQs by issuing a Force Eligible command on them.

7.4.2.2.5.1.12.3 Order Preservation & Discrete Consumption Acknowledgement

In addition to the dequeue atomicity feature, it is possible to obtain a stronger property from QMan to aid with datapath situations that "spread" incoming data over multiple portals. Specifically, if incoming frames are to be forwarded via subsequent enqueues, then dequeue atomicity does not prevent the forwarded frames from getting out of order. Ie. multiple CPUs (using multiple portals) may be using dequeue atomicity in order to write enqueue commands to their EQCR rings before consuming the DQRR entries, and thus ensuring that EQCR entries are *published* in the same order as the incoming frames. But as there are multiple portals, this does not ensure that QMan will necessarily *process* those EQCR entries in the same order. Indeed if the portals' EQCR rings have significantly varied fill-levels, then there is a reasonable chance that two enqueue commands published in quick succession via different portals could get processed in the opposite order by QMan.

Instead, software can elect to only consume DQRR entries when no forwarding is to be performed on the corresponding frames (eg. when dropping a packet), and for the others, it can encode the EQCR enqueue commands to perform an implicit "Discrete Consumption Acknowledgement" (or "DCA") - the result of which is that QMan will consume the corresponding DQRR entry on software's behalf *once it has finished processing the enqueue command*. This provides a cross-portal, order preservation semantic from end-to-end (from dequeue to enqueue) using hardware assists.

Note, QMan has other functionality called Order Restoration that is completely unrelated to the above - Order Restoration is a mechanism to restore frames into their intended order once they been allowed to get out of order, using sequence numbers

and "reassembly windows" within QMan, see Order Restoration on page 638. The above "hold active" mechanisms are to prevent frames from getting out of order in the first place.

7.4.2.2.5.1.13 Force Eligible

QMan portals support a management command called "Force Eligible" which allows software to regain control of a scheduled frame queue, usually with the intention to park it. When a frame queue is scheduled, QMan is responsible for its state and software can not meaningfully query it, as any snapshots are implicitly out of date by the time software sees them. Software only knows the frame queue state once QMan-generated events indicate that the frame queue is "quiesced" somehow. Moreover, if the frame queue is not configured for "hold active" behavior, then even the presence of DQRR entries does not help in this regard, as the portal may well have rescheduled the frame queue before software sees the first DQRR entry.

When QMan processes a Force Eligible command, it does two things - it tags the frame queue descriptor with a flag that is visible in subsequent DQRR entries, and, if the frame queue is Tentatively Scheduled (because it is empty), it will move the frame queue to the Truly Scheduled state (linked to a work queue). The result is that the frame queue "will receive dequeue processing soon", whether that was already happening or not. Fundamentally, when QMan is dequeuing from the FQ a short while later, it will treat the frame queue as "hold active", even if it isn't configured for hold active treatment. As such, software can request that the FQ be parked rather than rescheduled once the DQRR entry is consumed. See Parking Scheduled FQs on page 637.

7.4.2.2.5.1.14 Enqueue Rejections

Enqueues may be rejected, immediately or after any delay due to order restoration, and the enqueue mechanisms themselves do not provide any meaningful way to convey the rejection event to the software portal. For this reason, Enqueue Rejection Notifications (ERNs) are messages received on a message ring that carry frames that did not successfully enqueue together with the reason for their rejection.

7.4.2.2.5.1.15 Order Restoration

Frame queue descriptors can serve one or both of two complimentary purposes. A small subset of fields in the FQDs are used to implement an "Order Restoration Point", which allows an FQD to act as a reassembly window for out-of-sequence enqueues. FQDs also contain a sequence number field that generates increasing sequence numbers for all frames dequeued from the FQ. This dequeue activity sequence number is also called an "Order Definition Point". The idea is that frames dequeued from a given FQ (ODP) may get out-of-sequence during processing before they're enqueued onto an egress FQ, so the enqueue function allows one to not only specify the desination FQD, but also an ORP that the enqueue command should first pass through - which might hold up the intended enqueue until other, missing, sequence elements are enqueued. Ie. an ORP-enabled enqueue command requires 2 FQID parameters, which need not necessarily be the same - indeed in many networking examples, the Rx FQ serves as both the ODP and the ORP when enqueuing to the Tx FQ. To see why this choice of ORP FQ makes sense, consider that many Rx flows may need to be order-restored independently, even if all of them are ultimately enqueued to the same destination Tx FQ. It's also possible to enqueue using software-generated sequence numbers, ie. without any FQ dequeue activity acting as an ODP. An ODP is any source of sequence numbers starting at zero and wrapping to zero at 0x3fff (2^{14}-1).

ORP-enabled enqueue functions provide various features, such as filling in missing sequence numbers (eg. when dropping frames), advancing the "Next Expected Sequence Number" despite missing frames (that may or may not show up later), etc. These features are options in the enqueue interfaces, eg. see Enqueue Command (without ORP) on page 650, specifically the qman_enqueue_orp() API.

There are also numerous options that can be set in ORP-enabled FQDs, and these are achieved via the same functions that allow you to manipulate FQDs for any other purpose. Eg. see Frame queue management on page 645, specifically the qman_init_fq() API. Care should be taken when using a FQD as both a FQ and an ORP - in particular, a FQD can not be retired and put out-of-service while the ORP component of the descriptor is still in use, and vice versa.

7.4.2.2.5.2 QMan configuration interface

The QMan configuration interface is an encapsulation of the QMan CCSR register space and the global/error interrupt source. Whereas QMan portals provide independent channels for accessing QMan functionality, the configuration interface

represents the QMan device itself. The QMan configuration interface is presently limited to the device-tree node that represents it.

7.4.2.2.5.2.1 QMan device-tree node

The QMan device tree node represents the QMan device and its CCSR configuration space (as distinct from its corenet portals). When a linux kernel has QMan control support built in, it will react to this device tree node by configuring and managing the QMan device. The device-tree node sits within the CCSR node ("soc") and is of the following form;

```
        soc@fe000000 {
                [...]
                qman: qman@318000 {
                        compatible = "fsl,qman";
                        reg = <0x318000 0x1000>;
                        fsl,qman-fqd = <0x0 0x22000000 0x0 0x00200000>;
                        fsl,qman-pfdr = <0x0 0x21000000 0x0 0x01000000>;
                        fsl,liodn = <0x1f>;
                };
                [...]
          };
```

'compatible' and 'reg' are standard ePAPR properties.

7.4.2.2.5.2.1.1 Frame Queue Descriptors

This property configures the memory used by QMan for storing frame queue descriptors. Each FQD occupies a 64-byte cacheline of memory, so as the above example configures 2MB for FQD memory, the valid range of FQIDs is [1...32767];

```
                        fsl,qman-fqd = <0x0 0x22000000 0x0 0x00200000>;
```

The treatment and alignment requirements of this property are the same as in Free Buffer Proxy Records on page 624.

7.4.2.2.5.2.1.2 Packed Frame Descriptor Records

This property configures the memory used by QMan for storing Packed Frame Descriptor Records. Each PFDR occupies a 64-byte cacheline of memory, and can hold 3 Frame Descriptors. QMan maintains an onboard cache for holding recently enqueued (and/or soon to be dequeued) frames, and in responsive systems that remain within their operating capacity (ie. no spikes) it can often be unnecessary for frames to ever be stored in system memory at all. However, to handle spikes or buffering, a storage density of 3 enqueued frames per-cacheline can be used for estimating a suitable allocation of memory to QMan for PFDRs. In the case of handling ERNs (eg. if congestion controls exist elsewhere than on an ingress network interface), then a storage density of 1 ERN per-cacheline should be used. The above example configures 16MB for PFDR memory (786,432 enqueued frames, or 262,144 ERNs);

```
                        fsl,qman-pfdr = <0x0 0x21000000 0x0 0x01000000>;
```

The treatment and alignment requirements of this property are the same as in Free Buffer Proxy Records on page 624.

7.4.2.2.5.2.1.3 Logical I/O Device Number (QMan)

This property is the same as described in Logical I/O Device Number (BMan) on page 624, but for use by QMan when accessing FQD and PFDR memory (rather than BMan's FBPR memory).

7.4.2.2.5.2.2 QMan pool channel device-tree node

Each QMan software portal has its own dedicated channel of work queues. QMan also provides "pool channels" that all software portals can optionally dequeue from - this is described in Portals on page 633. The device-tree should declare pool channels using device-tree nodes as follows;

```
        qman-pool@1 {
                compatible = "fsl,qman-pool-channel";
                    cell-index = <0x1>;
```

```
                fsl,qman-channel-id = <0x21>;
        };
```

7.4.2.2.5.2.2.1 Channel ID

When FQs are initialized for scheduling, the target work queue is identified by the channel id (a hardware-assigned identifier) and by one of the 8 priority levels within that channel. Channel ids are hardware constants, as conveyed by this device-tree property;

```
                fsl,qman-channel-id = <0x21>;
```

7.4.2.2.5.2.3 QMan portal device-tree node

The QMan Corenet portal interface in QorIQ P4080 provides up to 10 distinct memory-mapped interfaces for use by software to interact efficiently with QMan functionality. These are described in Portals on page 633 and Portal Sub-Interfaces on page 634. Refer to the appropriate SoC reference manuals for non-P4080 specifications.

The QMan driver determines the available corenet portals from the device tree. The portal nodes are at the physical address scope (unlike the device-tree node for the BMan device itself, which is within the "soc" physical address node that represents CCSR). These nodes indicate the physical address ranges of the cache-enabled and cache-inhibited sub-regions of the portal (respectively), and look something like the following;

```
        qman-portal@c000 {
                compatible = "fsl,qman-portal";
                reg = <0xf420c000 0x4000 0xf4303000 0x1000>;
                interrupts = <0x6e 2>;
                interrupt-parent = <&mpic>;
                cell-index = <0x3>;
                cpu-handle = <&cpu3>;
                fsl,qman-channel-id = <0x3>;
                fsl,qman-pool-channels = <&qpool1 &qpool2>;
                fsl,liodn = <0x7 0x8>;
        };
```

As with BMan portal nodes, the "cpu-handle" property is used to express an affinity/association between the given QMan portal and the CPU represented by the referenced device-tree node. Unlike BMan however, the "cpu-handle" property is also used by PAMU configuration, to determine which CPU's L1 or L2 cache should receive stashing transactions emanating from this portal. The "fsl,qman-channel-id" property is already documented in Channel ID on page 640, the other QMan-specific portal properties are described below.

7.4.2.2.5.2.3.1 Portal Access to Pool Channels

In QorIQ P4080, P3041, P5020 hardware, all software portals can dequeue from any/all pool channels. Nonetheless, the portal device-tree nodes allow the architect to specify this and optionally limit the range of pool channels a given portal can dequeue from. This can be particularly useful when partitioning multiple guest operating systems, it essentially allows the architect to partition the use of pool channels as they partition the use of portals. In the above example, the portal is only able to dequeue from 2 pool channels;

```
                fsl,qman-pool-channels = <&qpool1 &qpool2>;
```

7.4.2.2.5.2.3.2 Stashing Logical I/O Device Number

This property, when used in QMan portal nodes, declares two LIODN values for use by QMan when performing dequeue stashing to processor cache. These are documented in Stashing to Processor Cache on page 635. This property is filled in automatically by u-boot, and if hypervisor is in use then it will fill in this property for guest device-trees also. PAMU drivers

(linux-native or within the hypervisor) will configure the settings for these LIODNs according to the CPU that stashing should be directed towards, as per the cpu-handle property;

```
                    fsl,liodn = <0x7 0x8>;
                    cpu-handle = <&cpu3>;
```

7.4.2.2.5.2.3.3 Portal Initialization (QMan)

The driver is informed of the QMan portals that are available to it via the device-tree passed to the system from the boot process. For those portals that aren't reserved for USDPAA usage via the "fsl,usdpaa-portal" property, it will automatically create TLB entries to map the QMan portal corenet sub-regions as cpu-addressable and cache-inhibited or cache-enabled as appropriate.

As with the BMan driver, the QMan driver will automatically associate initialised QMan portals with the CPU to which they are configured, only one a one-per-CPU basis (if multiple portals are configured for the same CPU, only one is used). Please see Portal sharing on page 626 for an explanation of this behaviour in the BMan documentation, the QMan behaviour is identical.

7.4.2.2.5.2.3.4 Auto-Initialization

As with the BMan driver, the QMan driver will, by default, automatically initialize QMan portals as they are parsed out of the device-tree. Please see Portal sharing on page 626 for an explanation of this behavior in the BMan documentation. The QMan behavior is identical.

7.4.2.2.6 QMan portal APIs

The following sections describe interfaces provided by the QMan driver for manipulating portals. These are defined in QMan portal device-tree node on page 640, and described in Portals on page 633 and Portal Sub-Interfaces on page 634.

Note, unlike the BMan documentation, we will not include many of the QMan-related data structures within this documentation as they are significantly more elaborate. It is presumed the reader will consult the corresponding header files for structure data details that aren't sufficiently described here.

7.4.2.2.6.1 QMan High-Level Portal Interface

7.4.2.2.6.1.1 Overview (QMan)

The high-level portal interface provides management and encapsulation of a portal hardware interface. The operations performed on the "portal" are coordinated internally, hiding the user from the I/O semantics, and allowing multiple users/contexts to share portals without collaboration between them. This interface also provides an object representation for congestion group records (CGRs), with optional assists for cases where the user wishes to track congestion entry and exit events, eg. to apply back-pressure on the affected frame queues, etc. There is also an object representation for frame queues that internally coordinates FQ operations, demuxes incoming dequeued frames and messages to the corresponding owner's callbacks, and interprets hardware-provided indications of changes to FQ state.

This interface provides locking and arbitration of portal operations from multiple software contexts and/or threads (ie. the portal is shared). In cases where a resource is busy, the interface also gives callers the option of blocking/sleeping until the resouce is available (and in the case of volatile dequeue commands, the caller may also optionally sleep until the volatile dequeue command has finished). In any case where sleeping is an option, the caller can also specify whether the sleep should be interruptible.

NOTE

Support for blocking/sleeping is limited to Linux, it is not available on run-to-completion systems such as USDPAA.

The demux logic within the portal interface assumes ownership of the "contextB" field of frame queue descriptors (FQDs), so users of this interface can not modify this field. However, callers provide the cache line of memory to be used within the driver for each FQ object when calling qman_create_fq(), so they can extend this structure into adjacent cachelines with their own data and use this instead of contextB for their own purposes. Ie. when callbacks are invoked because of dequeued

frames, enqueue rejections, or retirement notifications, those callbacks will find their custom per-FQ data adjacent to the FQ object pointer they are passed. Moreover, if context-stashing is enabled for the portal and the FQD is configured to stash 1 or more cachelines of context, the QMan driver's demux function will be implicitly accelerated because the FQ object will be prefetched into processor cache. Any adjacent data that is covered by the FQ's stashing configuration could likewise lead to acceleration of the owner's dequeue callbacks, ie. by reducing or eliminating cache misses in fast-path processing.

7.4.2.2.6.1.2 Frame and Message Handling

When DQRR or MR ring entries are produced by hardware to software, callbacks that have been provided by the API user are invoked to allow those entries to be handled prior to the driver consuming them. These callbacks are provided in the 'qman_fq_cb' structure type.

```
struct qman_fq_cb {
          qman_cb_dqrr dqrr; /* for dequeued frames */
          qman_cb_mr ern;          /* for software ERNs */
          qman_cb_mr dc_ern; /* for diverted hardware ERNs */
          qman_cb_mr fqr;           /* retirement messages */
};
typedef enum qman_cb_dqrr_result (*qman_cb_dqrr)(struct qman_portal *qm,
                                 struct qman_fq *fq, const struct qm_dqrr_entry *dqrr);
typedef void (*qman_cb_mr)(struct qman_portal *qm, struct qman_fq *fq,
                                 const struct qm_mr_entry *msg);
enum qman_cb_dqrr_result {
          /* DQRR entry can be consumed */
          qman_cb_dqrr_consume,
          /* Like _consume, but requests parking - FQ must be held-active */
          qman_cb_dqrr_park,
          /* Does not consume, for DCA mode only. This allows out-of-order
           * consumes by explicit calls to qman_dca() and/or the use of implicit
           * DCA via EQCR entries. */
          qman_cb_dqrr_defer
};
```

7.4.2.2.6.1.3 Portal management (QMan)

The portal management API provides qman_affine_cpus(), which returns a mask that indicates which CPUs have auto-initialiazed portals associated with them. See QMan portal device-tree node on page 640. All other QMan API functions must be executed on CPUs contained within this mask, and any interactions they require with h/w will be performed on the corresponding portals.

```
/**
 * qman_affine_cpus - return a mask of cpus that have portal access
 */
const cpumask_t *qman_affine_cpus(void);
```

7.4.2.2.6.1.3.1 Modifying interrupt-driven portal duties (QMan)

Portals have various servicing duties they must perform in reaction to hardware events. The portal management API allows applications to control which of these duties/events are triggered by interrupt-handling versus those which are performed at the application's explicit request via qman_poll() (or more specifically, via qman_poll_dqrr() and qman_poll_slow()). If portal-sharing is in effect (see Portal sharing on page 626), these APIs won't succeed when called from a slave CPU.

```
#define QM_PIRQ_CSCI    0x00100000      /* Congestion State Change */
#define QM_PIRQ_EQCI    0x00080000      /* Enqueue Command Committed */
#define QM_PIRQ_EQRI    0x00040000      /* EQCR Ring (below threshold) */
#define QM_PIRQ_DQRI    0x00020000      /* DQRR Ring (non-empty) */
#define QM_PIRQ_MRI     0x00010000      /* MR Ring (non-empty) */
#define QM_PIRQ_SLOW    (QM_PIRQ_CSCI | QM_PIRQ_EQCI | QM_PIRQ_EQRI | \
                         QM_PIRQ_MRI)
```

```
/**
 * qman_irqsource_get - return the portal work that is interrupt-driven
 *
 * Returns a bitmask of QM_PIRQ_**I processing sources that are currently
 * enabled for interrupt handling on the current cpu's affine portal. These
 * sources will trigger the portal interrupt and the interrupt handler (or a
 * tasklet/bottom-half it defers to) will perform the corresponding processing
 * work. The qman_poll_***() functions will only process sources that are not in
 * this bitmask. If the current CPU is sharing a portal hosted on another CPU,
 * this always returns zero.
 */
u32 qman_irqsource_get(void);
/**
 * qman_irqsource_add - add processing sources to be interrupt-driven
 * @bits: bitmask of QM_PIRQ_**I processing sources
 *
 * Adds processing sources that should be interrupt-driven (rather than
* processed via qman_poll_***() functions). Returns zero for success, or
* -EINVAL if the current CPU is sharing a portal hosted on another CPU.
*/
int qman_irqsource_add(u32 bits);
/**
* qman_irqsource_remove - remove processing sources from being interrupt-driven
* @bits: bitmask of QM_PIRQ_**I processing sources
*
* Removes processing sources from being interrupt-driven, so that they will
* instead be processed via qman_poll_***() functions. Returns zero for success,
* or -EINVAL if the current CPU is sharing a portal hosted on another CPU.
*/
int qman_irqsource_remove(u32 bits);
```

7.4.2.2.6.1.3.2 Processing non-interrupt-driven portal duties (QMan)

If portal-sharing is in effect (see Portal sharing on page 626), these APIs won't succeed when called from a slave CPU.

```
/**
* qman_poll_dqrr - process DQRR (fast-path) entries
* @limit: the maximum number of DQRR entries to process
*
* Use of this function requires that DQRR processing not be interrupt-driven.
* Ie. the value returned by qman_irqsource_get() should not include
* QM_PIRQ_DQRI. If the current CPU is sharing a portal hosted on another CPU,
* this function will return -EINVAL, otherwise the return value is >=0 and
* represents the number of DQRR entries processed.
*/
int qman_poll_dqrr(unsigned int limit);
/**
QMan Portal APIs
QMan, BMan API RM, Rev. 0.13
6-34 NXP Confidential Proprietary NXP Semiconductors
Preliminary—Subject to Change Without Notice
* qman_poll_slow - process anything (except DQRR) that isn't interrupt-driven.
*
* This function does any portal processing that isn't interrupt-driven. If the
* current CPU is sharing a portal hosted on another CPU, this function will
* return -EINVAL, otherwise returns zero for success.
*/
void qman_poll_slow(void);
/**
```

```
 * qman_poll - legacy wrapper for qman_poll_dqrr() and qman_poll_slow()
 *
 * Dispatcher logic on a cpu can use this to trigger any maintenance of the
 * affine portal. There are two classes of portal processing in question;
 * fast-path (which involves demuxing dequeue ring (DQRR) entries and tracking
 * enqueue ring (EQCR) consumption), and slow-path (which involves EQCR
 * thresholds, congestion state changes, etc). This function does whatever
 * processing is not triggered by interrupts.
 *
 * Note, if DQRR and some slow-path processing are poll-driven (rather than
 * interrupt-driven) then this function uses a heuristic to determine how often
 * to run slow-path processing - as slow-path processing introduces at least a
 * minimum latency each time it is run, whereas fast-path (DQRR) processing is
 * close to zero-cost if there is no work to be done. Applications can tune this
 * behavior themselves by using qman_poll_dqrr() and qman_poll_slow() directly
 * rather than going via this wrapper.
 */
void qman_poll(void);
```

7.4.2.2.6.1.3.3 Recovery support (QMan)

Note that the following functions require the QMan portal to have been initialized in "recovery mode", which is not possible with the current release. As such, these functions are for future use only (and documented here only because they're declared in the API header).

```
/**
 * qman_recovery_cleanup_fq - in recovery mode, cleanup a FQ of unknown state
 */
int qman_recovery_cleanup_fq(u32 fqid);
/**
 * qman_recovery_exit - leave recovery mode
 */
int qman_recovery_exit(void);
```

7.4.2.2.6.1.3.4 Stopping and restarting dequeues to the portal

```
/**
 * qman_stop_dequeues - Stop h/w dequeuing to the s/w portal
 *
 * Disables DQRR processing of the portal. This is reference-counted, so
 * qman_start_dequeues() must be called as many times as qman_stop_dequeues() to
 * truly re-enable dequeuing.
 */
void qman_stop_dequeues(void);
/**
 * qman_start_dequeues - (Re)start h/w dequeuing to the s/w portal
 *
 * Enables DQRR processing of the portal. This is reference-counted, so
 * qman_start_dequeues() must be called as many times as qman_stop_dequeues() to
 * truly re-enable dequeuing.
 */
void qman_start_dequeues(void);
```

7.4.2.2.6.1.3.5 Manipulating the portal static dequeue command

```
/**
 * qman_static_dequeue_add - Add pool channels to the portal SDQCR
```

```
 * @pools: bit-mask of pool channels, using QM_SDQCR_CHANNELS_POOL(n)
 *
 * Adds a set of pool channels to the portal's static dequeue command register
 * (SDQCR). The requested pools are limited to those the portal has dequeue
 * access to.
 */
void qman_static_dequeue_add(u32 pools);
/**
 * qman_static_dequeue_del - Remove pool channels from the portal SDQCR
 * @pools: bit-mask of pool channels, using QM_SDQCR_CHANNELS_POOL(n)
 *
 * Removes a set of pool channels from the portal's static dequeue command
 * register (SDQCR). The requested pools are limited to those the portal has
 * dequeue access to.
 */
void qman_static_dequeue_del(u32 pools);
/**
 * qman_static_dequeue_get - return the portal's current SDQCR
 *
 * Returns the portal's current static dequeue command register (SDQCR). The
 * entire register is returned, so if only the currently-enabled pool channels
 * are desired, mask the return value with QM_SDQCR_CHANNELS_POOL_MASK.
 */
u32 qman_static_dequeue_get(void);
```

7.4.2.2.6.1.3.6 Determining if the enqueue ring is empty

```
/**
 * qman_eqcr_is_empty - Determine if portal's EQCR is empty
 *
 * For use in situations where a cpu-affine caller needs to determine when all
 * enqueues for the local portal have been processed by QMan but can't use the
 * QMAN_ENQUEUE_FLAG_WAIT_SYNC flag to do this from the final qman_enqueue().
 * The function forces tracking of EQCR consumption (which normally doesn't
 * happen until enqueue processing needs to find space to put new enqueue
 * commands), and returns zero if the ring still has unprocessed entries,
 * non-zero if it is empty.
 */
int qman_eqcr_is_empty(void);
```

7.4.2.2.6.1.4 Frame queue management

Frame queue objects are stored in memory provided by the caller, which makes the API for this object representation a little peculiar at first sight. The motivating factors are memory management and stashing of frame queue context. Another factor is that frame queue objects are the only objects in the QMan (or BMan) high level interfaces that are essentially arbitrary in number, so having the caller provide storage relieves the driver of having to know the best allocation scheme for all applications.

The qman_create_fq() API creates a new frame queue object, using the caller-supplied storage, and in which the caller has already configured the callback functions to be used for handling hardware-produced data - namely, DQRR entries and MR entries, the latter divided according to the type of message (software-enqueue rejections, hardware-enqueue rejections, or frame queue state changes).

```
#define QMAN_FQ_FLAG_NO_ENQUEUE      0x00000001 /* can't enqueue */
#define QMAN_FQ_FLAG_NO_MODIFY       0x00000002 /* can only enqueue */
#define QMAN_FQ_FLAG_TO_DCPORTAL     0x00000004 /* consumed by CAAM/PME/FMan */
#define QMAN_FQ_FLAG_LOCKED          0x00000008 /* multi-core locking */
#define QMAN_FQ_FLAG_AS_I            0x00000010 /* query h/w state */
```

```
#define QMAN_FQ_FLAG_DYNAMIC_FQID    0x00000020 /* (de)allocate fqid */
struct qman_fq {
        /* Caller of qman_create_fq() provides these demux callbacks */
        struct qman_fq_cb {
                qman_cb_dqrr dqrr;      /* for dequeued frames */
                qman_cb_mr ern;         /* for s/w ERNs */
                qman_cb_mr dc_ern;      /* for diverted h/w ERNs */
                qman_cb_mr fqs;         /* frame-queue state changes*/
        } cb;
          /* Internal to the driver, don't touch. */
          [...]
};
/**
 * qman_create_fq - Allocates a FQ
 * @fqid: the index of the FQD to encapsulate, must be "Out of Service"
 * @flags: bit-mask of QMAN_FQ_FLAG_*** options
 * @fq: memory for storing the 'fq', with callbacks filled in
 *
 * Creates a frame queue object for the given @fqid, unless the
 * QMAN_FQ_FLAG_DYNAMIC_FQID flag is set in @flags, in which case a FQID is
 * dynamically allocated (or the function fails if none are available). Once
 * created, the caller should not touch the memory at 'fq' except as extended to

 * adjacent memory for user-defined fields (see the definition of "struct
 * qman_fq" for more info). NO_MODIFY is only intended for enqueuing to
 * pre-existing frame-queues that aren't to be otherwise interfered with, it
 * prevents all other modifications to the frame queue. The TO_DCPORTAL flag
 * causes the driver to honour any contextB modifications requested in the
 * qm_init_fq() API, as this indicates the frame queue will be consumed by a
 * direct-connect portal (PME, CAAM, or FMan). When frame queues are consumed by

 * software portals, the contextB field is controlled by the driver and can't be

 * modified by the caller. If the AS_SI flag is specified, management commands
 * will be used on portal @p to query state for frame queue @fqid and construct
 * a frame queue object based on that, rather than assuming/requiring that it be
 * Out of Service.
 */
int qman_create_fq(u32 fqid, u32 flags, struct qman_fq *fq);
#define QMAN_FQ_DESTROY_PARKED       0x00000001 /* FQ can be parked or OOS */
/**
 * qman_destroy_fq - Deallocates a FQ
 * @fq: the frame queue object to release
 * @flags: bit-mask of QMAN_FQ_DESTROY_*** options
 *
 * The memory for this frame queue object ('fq' provided in qman_create_fq()) is
 * not deallocated but the caller regains ownership, to do with as desired. The
 * FQ must be in the 'out-of-service' state unless the QMAN_FQ_DESTROY_PARKED
 * flag is specified, in which case it may also be in the 'parked' state.
 */
void qman_destroy_fq(struct qman_fq *fq, u32 flags);
```

7.4.2.2.6.1.4.1 Querying a FQ object

The following functions do not interact with h/w, they simply return the state that the QMan driver tracks within the FQ object.

```
/**
 * qman_fq_fqid - Queries the frame queue ID of a FQ object
 * @fq: the frame queue object to query
 */
```

```
u32 qman_fq_fqid(struct qman_fq *fq);
enum qman_fq_state {
        qman_fq_state_oos,
        qman_fq_state_parked,
        qman_fq_state_sched,
        qman_fq_state_retired
};
#define QMAN_FQ_STATE_CHANGING       0x80000000 /* 'state' is changing */
#define QMAN_FQ_STATE_NE             0x40000000 /* retired FQ isn't empty */
#define QMAN_FQ_STATE_ORL            0x20000000 /* retired FQ has ORL */
#define QMAN_FQ_STATE_BLOCKOOS       0xe0000000 /* if any are set, no OOS */
#define QMAN_FQ_STATE_CGR_EN         0x10000000 /* CGR enabled */
/**
 * qman_fq_state - Queries the state of a FQ object
 * @fq: the frame queue object to query
 * @state: pointer to state enum to return the FQ scheduling state
 * @flags: pointer to state flags to receive QMAN_FQ_STATE_*** bitmask
 *
 * Queries the state of the FQ object, without performing any h/w commands.
 * This captures the state, as seen by the driver, at the time the function
 * executes.
 */
void qman_fq_state(struct qman_fq *fq, enum qman_fq_state *state, u32 *flags);
```

7.4.2.2.6.1.4.2 Initialize a FQ

The qman_init_fq() API requires that the caller fill in the details of the Initialize FQ command that they desire, and uses the 'struct qm_mcc_initfq' structure type to this end. This structure is quite elaborate, please consult the API header file and SDK examples for more informatoin.

```
#define QMAN_INITFQ_FLAG_SCHED       0x00000001 /* schedule rather than park */
#define QMAN_INITFQ_FLAG_NULL        0x00000002 /* zero 'contextB', no demux */
#define QMAN_INITFQ_FLAG_LOCAL       0x00000004 /* set dest portal */
/**
 * qman_init_fq - Initialises FQ fields, leaves the FQ "parked" or "scheduled"
 * @fq: the frame queue object to modify, must be 'parked' or new.
 * @flags: bit-mask of QMAN_INITFQ_FLAG_*** options
 * @opts: the FQ-modification settings, as defined in the low-level API
 *
 * @opts: the FQ-modification settings
*
* Select QMAN_INITFQ_FLAG_SCHED in @flags to cause the frame queue to be
* scheduled rather than parked. Select QMAN_INITFQ_FLAG_NULL in @flags to
* configure a frame queue that will not demux to a 'struct qman_fq' object when
* dequeued frames or messages arrive at a software portal, but which will
* instead trigger the portal's 'null_cb' callbacks (see qman_create_portal()).
* NB, @opts can be NULL.
 *
 * Note that some fields and options within @opts may be ignored or overwritten
 * by the driver;
 * 1. the 'count' and 'fqid' fields are always ignored (this operation only
 * affects one frame queue: @fq).
 * 2. the QM_INITFQ_WE_CONTEXTB option of the 'we_mask' field and the associated
 * 'fqd' structure's 'context_b' field are sometimes overwritten;
 *   - if @flags contains QMAN_INITFQ_FLAG_NULL, then context_b is initialized
 *     to zero by the driver,
 *   - if @fq was not created with QMAN_FQ_FLAG_TO_DCPORTAL, then context_b is
 *     initialized to a value used by the driver for demux.
 *   - if context_b is initialized for demux, so is context_a in case stashing
```

```
 *     is requested (see item 4).
 * (So caller control of context_b is only possible for TO_DCPORTAL frame queue
 * objects.)
 * 3. if @flags contains QMAN_INITFQ_FLAG_LOCAL, the 'fqd' structure's
 * 'dest::channel' field will be overwritten to match the portal used to issue
 * the command. If the WE_DESTWQ write-enable bit had already been set by the
 * caller, the channel workqueue will be left as-is, otherwise the write-enable
 * bit is set and the workqueue is set to a default of 4. If the "LOCAL" flag
 * isn't set, the destination channel/workqueue fields and the write-enable bit
 * are left as-is.
 * 4. if the driver overwrites context_a/b for demux, then if
 * QM_INITFQ_WE_CONTEXTA is set, the driver will only overwrite
 * context_a.address fields and will leave the stashing fields provided by the
 * user alone, otherwise it will zero out the context_a.stashing fields.
 */
int qman_init_fq(struct qman_fq *fq, u32 flags, struct qm_mcc_initfq *opts);
```

7.4.2.2.6.1.4.3 Schedule a FQ

```
/**
 * qman_schedule_fq - Schedules a FQ
 * @fq: the frame queue object to schedule, must be 'parked'
 *
 * Schedules the frame queue, which must be Parked, which takes it to
 * Tentatively-Scheduled or Truly-Scheduled depending on its fill-level.
 */
int qman_schedule_fq(struct qman_fq *fq);
```

7.4.2.2.6.1.4.4 Retire a FQ

```
/**
 * qman_retire_fq - Retires a FQ
 * @fq: the frame queue object to retire
 * @flags: FQ flags (as per qman_fq_state) if retirement completes immediately
 *
 * Retires the frame queue. This returns zero if it succeeds immediately, +1 if
 * the retirement was started asynchronously, otherwise it returns negative for
 * failure. When this function returns zero, @flags is set to indicate whether
 * the retired FQ is empty and/or whether it has any ORL fragments (to show up
 * as ERNs). Otherwise the corresponding flags will be known when a subsequent
 * FQRN message shows up on the portal's message ring.
 *
 * NB, if the retirement is asynchronous (the FQ was in the Truly Scheduled or
 * Active state), the completion will be via the message ring as a FQRN - but
 * the corresponding callback may occur before this function returns!! Ie. the
 * caller should be prepared to accept the callback as the function is called,
 * not only once it has returned.
 */
int qman_retire_fq(struct qman_fq *fq, u32 *flags);
```

7.4.2.2.6.1.4.5 Put a FQ out of service

```
/**
 * qman_oos_fq - Puts a FQ "out of service"
 * @fq: the frame queue object to be put out-of-service, must be 'retired'
 *
 * The frame queue must be retired and empty, and if any order restoration list
```

```
 * was released as ERNs at the time of retirement, they must all be consumed.
 */
int qman_oos_fq(struct qman_fq *fq);
```

7.4.2.2.6.1.4.6 Query a FQD from QMan

The following functions perform query commands via the QMan software portal to obtain information about the FQD corresponding to the given FQ object. The data structures used by the query are quite elaborate, please consult the API header file and SDK examples for more information.

```
/**
 * qman_query_fq - Queries FQD fields (via h/w query command)
 * @fq: the frame queue object to be queried
 * @fqd: storage for the queried FQD fields
 */
int qman_query_fq(struct qman_fq *fq, struct qm_fqd *fqd);
/**
 * qman_query_fq_np - Queries non-programmable FQD fields
 * @fq: the frame queue object to be queried
 * @np: storage for the queried FQD fields
 */
int qman_query_fq_np(struct qman_fq *fq, struct qm_mcr_queryfq_np *np);
```

7.4.2.2.6.1.4.7 Unscheduled (volatile) dequeuing of a FQ

```
#define QMAN_VOLATILE_FLAG_WAIT      0x00000001 /* wait if VDQCR is in use */
#define QMAN_VOLATILE_FLAG_WAIT_INT  0x00000002 /* if wait, interruptible? */
#define QMAN_VOLATILE_FLAG_FINISH    0x00000004 /* wait till VDQCR completes */
/**
 * qman_volatile_dequeue - Issue a volatile dequeue command
 * @fq: the frame queue object to dequeue from (or NULL)
 * @flags: a bit-mask of QMAN_VOLATILE_FLAG_*** options
 * @vdqcr: bit mask of QM_VDQCR_*** options, as per qm_dqrr_vdqcr_set()
 *
 * Attempts to lock access to the portal's VDQCR volatile dequeue functionality.
 * The function will block and sleep if QMAN_VOLATILE_FLAG_WAIT is specified and
 * the VDQCR is already in use, otherwise returns non-zero for failure. If
 * QMAN_VOLATILE_FLAG_FINISH is specified, the function will only return once
 * the VDQCR command has finished executing (ie. once the callback for the last
 * DQRR entry resulting from the VDQCR command has been called). If @fq is
 * non-NULL, the corresponding FQID will be substituted in to the VDQCR command,
 * otherwise it is assumed that @vdqcr already contains the FQID to dequeue
 * from.
 */
int qman_volatile_dequeue(struct qman_fq *fq, u32 flags, u32 vdqcr)
```

7.4.2.2.6.1.4.8 Set FQ flow control state

```
/**
 * qman_fq_flow_control - Set the XON/XOFF state of a FQ
 * @fq: the frame queue object to be set to XON/XOFF state, must not be 'oos',
 * or 'retired' or 'parked' state
 * @xon: boolean to set fq in XON or XOFF state
 *
 * The frame should be in Tentatively Scheduled state or Truly Schedule sate,
 * otherwise the IFSI interrupt will be asserted.
```

```
 */
int qman_fq_flow_control(struct qman_fq *fq, int xon);
```

7.4.2.2.6.1.5 Enqueue Command (without ORP)

```
#define QMAN_ENQUEUE_FLAG_WAIT       0x00010000 /* wait if EQCR is full */
#define QMAN_ENQUEUE_FLAG_WAIT_INT   0x00020000 /* if wait, interruptible? */
#define QMAN_ENQUEUE_FLAG_WAIT_SYNC  0x00000004 /* if wait, until consumed? */
#define QMAN_ENQUEUE_FLAG_WATCH_CGR  0x00080000 /* watch congestion state */
#define QMAN_ENQUEUE_FLAG_DCA        0x00008000 /* perform enqueue-DCA */
#define QMAN_ENQUEUE_FLAG_DCA_PARK   0x00004000 /* If DCA, requests park */
#define QMAN_ENQUEUE_FLAG_DCA_PTR(p)            /* If DCA, p is DQRR entry */ \
                (((u32)(p) << 2) & 0x00000f00)
#define QMAN_ENQUEUE_FLAG_C_GREEN    0x00000000 /* choose one C_*** flag */
#define QMAN_ENQUEUE_FLAG_C_YELLOW   0x00000008
#define QMAN_ENQUEUE_FLAG_C_RED      0x00000010
#define QMAN_ENQUEUE_FLAG_C_OVERRIDE 0x00000018
/**
 * qman_enqueue - Enqueue a frame to a frame queue
 * @fq: the frame queue object to enqueue to
 * @fd: a descriptor of the frame to be enqueued
 * @flags: bit-mask of QMAN_ENQUEUE_FLAG_*** options
 *
 * Fills an entry in the EQCR of portal @qm to enqueue the frame described by
 * @fd. The descriptor details are copied from @fd to the EQCR entry, the 'pid'
 * field is ignored. The return value is non-zero on error, such as ring full
 * (and FLAG_WAIT not specified), congestion avoidance (FLAG_WATCH_CGR
 * specified), etc. If the ring is full and FLAG_WAIT is specified, this
 * function will block. If FLAG_INTERRUPT is set, the EQCI bit of the portal
 * interrupt will assert when QMan consumes the EQCR entry (subject to "status
 * disable", "enable", and "inhibit" registers). If FLAG_DCA is set, QMan will
 * perform an implied "discrete consumption acknowledgement" on the dequeue
 * ring's (DQRR) entry, at the ring index specified by the FLAG_DCA_IDX(x)
 * macro. (As an alternative to issuing explicit DCA actions on DQRR entries,
 * this implicit DCA can delay the release of a "held active" frame queue
 * corresponding to a DQRR entry until QMan consumes the EQCR entry - providing
 * order-preservation semantics in packet-forwarding scenarios.) If FLAG_DCA is
 * set, then FLAG_DCA_PARK can also be set to imply that the DQRR consumption
 * acknowledgement should "park request" the "held active" frame queue. Ie.
 * when the portal eventually releases that frame queue, it will be left in the
 * Parked state rather than Tentatively Scheduled or Truly Scheduled. If the
 * portal is watching congestion groups, the QMAN_ENQUEUE_FLAG_WATCH_CGR flag
 * is requested, and the FQ is a member of a congestion group, then this
 * function returns -EAGAIN if the congestion group is currently congested.
 * Note, this does not eliminate ERNs, as the async interface means we can be
 * sending enqueue commands to an un-congested FQ that becomes congested before
 * the enqueue commands are processed, but it does minimise needless thrashing
 * of an already busy hardware resource by throttling many of the to-be-dropped
 * enqueues "at the source".
 */
int qman_enqueue(struct qman_fq *fq, const struct qm_fd *fd, u32 flags);
```

7.4.2.2.6.1.6 Enqueue Command with ORP

```
/* Same flags as qman_enqueue(), with the following additions;

 * - this flag indicates "Not Last In Sequence", ie. all but the final fragment
```

```
 *   of a frame. */
#define QMAN_ENQUEUE_FLAG_NLIS       0x01000000
/* - this flag performs no enqueue but fills in an ORP sequence number that
 *   would otherwise block it (eg. if a frame has been dropped). */
#define QMAN_ENQUEUE_FLAG_HOLE       0x02000000
/* - this flag performs no enqueue but advances NESN to the given sequence
 *   number. */
#define QMAN_ENQUEUE_FLAG_NESN       0x04000000
/*
 * qman_enqueue_orp - Enqueue a frame to a frame queue using an ORP
 * @fq: the frame queue object to enqueue to
 * @fd: a descriptor of the frame to be enqueued
 * @flags: bit-mask of QMAN_ENQUEUE_FLAG_*** options
 * @orp: the frame queue object used as an order restoration point.
 * @orp_seqnum: the sequence number of this frame in the order restoration path
 *
 * Similar to qman_enqueue(), but with the addition of an Order Restoration
 * Point (@orp) and corresponding sequence number (@orp_seqnum) for this
 * enqueue operation to employ order restoration. Each frame queue object acts
 * as an Order Definition Point (ODP) by providing each frame dequeued from it
 * with an incrementing sequence number, this value is generally ignored unless
 * that sequence of dequeued frames will need order restoration later. Each
 * frame queue object also encapsulates an Order Restoration Point (ORP), which
 * is a re-assembly context for re-ordering frames relative to their sequence
 * numbers as they are enqueued. The ORP does not have to be within the frame
 * queue that receives the enqueued frame, in fact it is usually the frame
 * queue from which the frames were originally dequeued. For the purposes of
 * order restoration, multiple frames (or "fragments") can be enqueued for a
 * single sequence number by setting the QMAN_ENQUEUE_FLAG_NLIS flag for all
 * enqueues except the final fragment of a given sequence number.  Ordering
 * between sequence numbers is guaranteed, even if fragments of different
 * sequence numbers are interlaced with one another. Fragments of the same
 * sequence number will retain the order in which they are enqueued. If no
 * enqueue is to performed, QMAN_ENQUEUE_FLAG_HOLE indicates that the given
 * sequence number is to be "skipped" by the ORP logic (eg. if a frame has been
 * dropped from a sequence), or QMAN_ENQUEUE_FLAG_NESN indicates that the given
 * sequence number should become the ORP's "Next Expected Sequence Number".
 *
 * Side note: a frame queue object can be used purely as an ORP, without
 * carrying any frames at all. Care should be taken not to deallocate a frame
 * queue object that is being actively used as an ORP, as a future allocation
 * of the frame queue object may start using the internal ORP before the
 * previous use has finished.
 */
int qman_enqueue_orp(struct qman_fq *fq, const struct qm_fd *fd, u32 flags,

                                struct qman_fq *orp, u16 orp_seqnum);
```

7.4.2.2.6.1.7 DCA Mode

As described in Order Preservation & Discrete Consumption Acknowledgement on page 637, FQs initialized for "hold active" behavior can have order-preservation behavior if their DQRR entries are consumed either by implicit DCA in the enqueue command when forwarding, or by explicit DCA if the frame is not going to be forwarded. The implicit DCA via enqueue is described in Enqueue Command (without ORP) on page 650, this section describes the API for performing an explicit DCA on a DQRR entry. As with the implicit DCA via enqueue, explicit DCA commands also allow the caller to specify that the FQ be Parked rather than rescheduled once all its DQRR entries are consumed.

```
/**
 * qman_dca - Perform a Discrete Consumption Acknowledgement
```

```
 * @dq: the DQRR entry to be consumed
 * @park_request: indicates whether the held-active @fq should be parked
 *
 * Only allowed in DCA-mode portals, for DQRR entries whose handler callback had
 * previously returned 'qman_cb_dqrr_defer'. NB, as with the other APIs, this
 * does not take a 'portal' argument but implies the core affine portal from the

 * cpu that is currently executing the function. For reasons of locking, this
 * function must be called from the same CPU as that which processed the DQRR
 * entry in the first place.
 */
void qman_dca(struct qm_dqrr_entry *dq, int park_request);
```

7.4.2.2.6.1.8 Congestion Management Records

QMan supports a fixed number[3] of built-in resources called Congestion Group Records (CGRs), that can be used as containers for related frame queues that should collectively benefit from congestion management. The precise algorithms used for congestion management with these records is beyond the scope of the document, please see the Queue Manager section of the appropraite QorIQ SoC Reference Manual for details.

The CGR kernel structure enables access to the CGR hardware functionality. Each object refers to an underlining hardware record via the cgrid field. Many CGR object may reference the same cgrid, but care must be taken when this object resides on different cores as no inter-core protection is provided.

The init frame queue functionality allows the caller to associate a CGR with the associated frame queue. The interface permits the management and modification of the underlining CGRs and notifies the user of congestion state changed. The current interface does not provide a mechanism to manage CGR ids. The application software is expected to arbitrate use of CGR ids.

```
/* Flags to qman_modify_cgr() */
#define QMAN_CGR_FLAG_USE_INIT       0x00000001
/**
 * This is a qman cgr callback function which gets invoked when the
typedef void (*qman_cb_cgr)(struct qman_portal *qm,
           struct qman_cgr *cgr, int congested);
struct qman_cgr {
          /* Set these prior to qman_create_cgr() */
          u32 cgrid; /* 0..255 */
          qman_cb_cgr cb;
          enum qm_channel chan; /* portal channel this object is created on */
          struct list_head node;
};
/* When Weighted Random Early Discard (WRED) is used then the following
 * structure is used to configure the WRED parameters. Refer to the QMan
 * Block Guide for a detailed description of the various parameters.
 */
struct qm_cgr_wr_parm {
          union {
                    u32 word;
                    struct {
                              u32 MA:8;
                              u32 Mn:5;
                              u32 SA:7; /* must be between 64-127 */
                              u32 Sn:6;
                              u32 Pn:6;
                    } __packed;
          };
```

[3] 256 for P4080/P5020/P3041

```
} __packed;
/* This struct represents the 13-bit "CS_THRES" CGR field. In the corresponding
 * management commands, this is padded to a 16-bit structure field, so that's
 * how we represent it here. The congestion state threshold is calculated from
 * these fields as follows;
 *   CS threshold = TA * (2 ^ Tn)
 */
struct qm_cgr_cs_thres {
          u16 __reserved:3;
          u16 TA:8;
          u16 Tn:5;
} __packed;
/* This identical structure of CGR fields is present in the "Init/Modify CGR"
 * commands and the "Query CGR" result. It's suctioned out here into its own
 * struct. */
struct __qm_mc_cgr {
          struct qm_cgr_wr_parm wr_parm_g;
          struct qm_cgr_wr_parm wr_parm_y;
          struct qm_cgr_wr_parm wr_parm_r;
          u8 wr_en_g; /* boolean, use QM_CGR_EN */
          u8 wr_en_y;  /* boolean, use QM_CGR_EN */
          u8 wr_en_r;  /* boolean, use QM_CGR_EN */
          u8 cscn_en;   /* boolean, use QM_CGR_EN */
          union {
                  struct {
                           u16 cscn_targ_upd_ctrl; /* use QM_CSCN_TARG_UDP_ */
                           u16 cscn_targ_dcp_low;  /* CSCN_TARG_DCP low-16bits */
                  };
                  u32 cscn_targ;  /* use QM_CGR_TARG_* */
          };
          u8 cstd_en;   /* boolean, use QM_CGR_EN */
          u8 cs;                  /* boolean, only used in query response */
          struct qm_cgr_cs_thres cs_thres;
          u8 mode;   /* QMAN_CRG_MODE_FRAME not supported in rev1.0 */
} __packed
struct qm_mcc_initcgr {
          u8 __reserved1;
          u16 we_mask; /* Write Enable Mask */
          struct __qm_mc_cgr cgr; /* CGR fields */
          u8 __reserved2[2];
          u8 cgid;
          u8 __reserved4[32];
} __packed;
/**
 * qman_create_cgr - Register a congestion group object
 * @cgr: the 'cgr' object, with fields filled in
 * @flags: QMAN_CGR_FLAG_* values
 * @opts: optional state of CGR settings
 *
 * Registers this object to receiving congestion entry/exit callbacks on the
 * portal affine to the cpu portal on which this API is executed. If opts is
 * NULL then only the callback (cgr->cb) function is registered. If @flags
 * contains QMAN_CGR_FLAG_USE_INIT, then an init hw command (which will reset
 * any unspecified parameters) will be used rather than a modify hw hardware
 * (which only modifies the specified parameters).
 */
int qman_create_cgr(struct qman_cgr *cgr, u32 flags, struct qm_mcc_initcgr *opts);
/**
 * qman_create_cgr_to_dcp - Register a congestion group object to DCP portal
 * @cgr: the 'cgr' object, with fields filled in
```

```
 * @flags: QMAN_CGR_FLAG_* values
 * @dcp_portal: the DCP portal to which the cgr object is registered
 * @opts: optional state of CGR settings
 *
 */
int qman_create_cgr_to_dcp(struct qman_cgr *cgr, u32 flags, u16 dcp_portal,
                           struct qm_mcc_initcgr *opts);
/**
 * qman_delete_cgr - Deregisters a congestion group object
 * @cgr: the 'cgr' object to deregister
 *
 * "Unplugs" this CGR object from the portal affine to the cpu on which this API
 * is executed. This must be excuted on the same affine portal on which it was
 * created.
 */
int qman_delete_cgr(struct qman_cgr *cgr);
/**
 * qman_modify_cgr - Modify CGR fields
 * @cgr: the 'cgr' object to modify
 * @flags: QMAN_CGR_FLAG_* values
 * @opts: the CGR-modification settings
 *
 * The @opts parameter can be NULL. Note that some fields and options within
* @opts may be ignored or overwritten by the driver, in particular the 'cgrid'
* field is ignored (this operation only affects the given CGR object). If
* @flags contains QMAN_CGR_FLAG_USE_INIT, then an init hw command (which will
* reset any unspecified parameters) will be used rather than a modify hw
* hardware (which only modifies the specified parameters).
 */
int qman_modify_cgr(struct qman_cgr *cgr, u32 flags, struct qm_mcc_initcgr *opts);
/**
 * qman_query_cgr - Queries CGR fields
 * @cgr: the 'cgr' object to query
 * @result: storage for the queried congestion group record
 */
int qman_query_cgr(struct qman_cgr *cgr, struct qm_mcr_querycgr *result);
```

7.4.2.2.6.1.9 Zero-Configuration Messaging

As described in Overview (QMan) on page 641, the demux logic of the QMan portal driver uses the contextB field of FQDs, as published in DQRR and MR entries, to determine the corresponding FQ object, and from there the DQRR or MR callback to invoke. However, "default callbacks" can also be associated with a portal that will be used if a "NULL" FQ is dequeued from, where NULL refers to a FQD whose contextB entry has been initialized to NULL (this occurs when using the QMAN_INITFQ_FLAG_NULL flag to the qman_init_fq() API, described in Initialize a FQ on page 647).

The purpose of this mechanism is to allow the user of one portal to enqueue frames on any frame queue that is configured in this way and schedule it to another portal. For virtualization or AMP scenarios, it is a difficult architectural problem to configure all guest operating systems to agree, in advance, on run-time parameters. The use of NULL frame queues allows a control plane guest OS to use any frame queue, configured with a NULL "contextB" field (see the QMAN_INITFQ_FLAG_NULL flag in the "Frame queue management" section below), to send any and all such configuration to another guest by scheduling that NULL frame queue to one of the target guest's portals. The target guest will have the portal's "NULL" callbacks invoked rather than those of any frame queue objects, and as such this provides what could be considered a "zero-configuration" interface - no agreement is required over what frame queue that configuration information will be arriving on, only that the configuration will arrive via the portal as a message on a NULL frame queue.

NOTE

Unless the payload of FDs passed over a zero-config FQ fits entirely within the 32-bit cmd/status field, buffers will presumably be required and the zero-configuration mechanism described here does not address how the sending and receiving ends should agree on what memory resources and management to use for this.

```
/**
 * qman_get_null_cb - get callbacks currently used for "null" frame queues
 *
 * Copies the callbacks used for the affine portal of the current cpu.
 */
void qman_get_null_cb(struct qman_fq_cb *null_cb);
/**
 * qman_set_null_cb - set callbacks to use for "null" frame queues
 *
 * Sets the callbacks to use for the affine portal of the current cpu, whenever
 * a DQRR or MR entry refers to a "null" FQ object. (Eg. zero-conf messaging.)
 */
void qman_set_null_cb(const struct qman_fq_cb *null_cb);
```

7.4.2.2.6.1.10 FQ allocation

7.4.2.2.6.1.10.1 Ad-hoc FQ allocator

As described in Seeding Buffer Pools on page 625, BMan buffer pool ID zero is currently reserved for use as an ad-hoc FQ allocator. As seen in Frame queue management on page 645, this feature can be used implicitly when creating a FQ object by passing the QMAN_FQ_FLAG_DYNAMIC_FQID flag to qman_init_fq(). The advantage of this mechanism is that it works across all cpus/portals, independent of any hypervisor or other system partitioning. The disadvantage of this mechanism is that does not permit the atomic nor contiguous allocation of more than one FQ at a time, and in particular most high-performance uses of FMan require contiguous ranges of FQIDs that also meet certain alignment requirements (ie. that the FQID range begins on an aligned FQID value).

7.4.2.2.6.1.10.2 FQ range allocator

The following APIs allow software to allocate and release arbitrary ranges of FQIDs, but it should be noted that the current version of the NXP Datapath software implements this without any hardware interaction. As such, multiple (guest) systems running on the same chip will each have their own allocator and are not aware of each other's (de)allocations. The range allocator's default state is empty, and it can be seeded by calling qman_release_fqid_range() on initialization with an appropriate FQID range to manage. The intention is for the control-plane software to initialize this range and to perform all allocations and deallocations on behalf of any software running on different system instances.

```
/**
 * qman_alloc_fqid_range - Allocate a contiguous range of FQIDs
 * @result: is set by the API to the base FQID of the allocated range
 * @count: the number of FQIDs required
 * @align: required alignment of the allocated range
 * @partial: non-zero if the API can return fewer than @count FQIDs
 * Returns the number of frame queues allocated, or a negative error code. If
 * @partial is non zero, the allocation request may return a smaller range of
 * FQs than requested (though alignment will be as requested). If @partial is
 * zero, the return value will either be 'count' or negative.
 */
int qman_alloc_fqid_range(u32 *result, u32 count, u32 align, int partial);
/**
 * qman_release_fqid_range - Release the specified range of frame queue IDs
 * @fqid: the base FQID of the range to deallocate
 * @count: the number of FQIDs in the range
 *
```

```
 * This function can also be used to seed the allocator with ranges of FQIDs
 * that it can subsequently use. Returns zero for success.
 */
void qman_release_fqid_range(u32 fqid, unsigned int count);
```

7.4.2.2.6.1.10.3 Future FQ allocator changes

Please note that a future version of the NXP Datapath software will automatically seed the range allocator with all FQIDs available to QMan, it will reimplement these APIs over an IPC layer such that all system instances share a common allocator instance, and the BMan-based FQ allocator will be removed and the corresponding APIs being reimplemented to use this range allocator.

7.4.2.2.6.1.11 Helper functions

In cases where software running on different CPUs communicate using QMan frame queues, there can arise an initialization problem related to synchronisation. If one side is termed the producer and the other the consumer, then the question becomes one of when it is safe for the producer to enqueue to that FQ. It is normal for software consumers to take care of initializing and scheduling FQs, because they must provide initialization and scheduling details in order for dequeue-handling to function correctly. But on the producer side, any attempt to enqueue to the FQ prior to the FQ being initialized will be rejected (enqueues are not permitted to OutOfService FQs). The following inline function can be used directly or as an example of how to determine when a FQ has changed state.

NOTE

It is safe for the producer to enqueue once the FQ has been initialized but not yet scheduled by the consumer.

```
/**
 * qman_poll_fq_for_init - Check if an FQ has been initialized from OOS
 * @fqid: the FQID that will be initialized by other s/w
 *
 * In many situations, a FQID is provided for communication between s/w
 * entities, and whilst the consumer is responsible for initialising and
 * scheduling the FQ, the producer(s) generally create a wrapper FQ object using
 * and only call qman_enqueue() (no FQ initialisation, scheduling, etc). Ie;
 *     qman_create_fq(..., QMAN_FQ_FLAG_NO_MODIFY, ...);
 * However, data can not be enqueued to the FQ until it is initialized out of
 * the OOS state - this function polls for that condition. It is particularly
 * useful for users of IPC functions - each endpoint's Rx FQ is the other
 * endpoint's Tx FQ, so each side can initialise and schedule their Rx FQ object
 * and then use this API on the (NO_MODIFY) Tx FQ object in order to
 * synchronise. The function returns zero for success, +1 if the FQ is still in
 * the OOS state, or negative if there was an error.
 */
static inline int qman_poll_fq_for_init(struct qman_fq *fq)
{
        struct qm_mcr_queryfq_np np;
        int err;
        err = qman_query_fq_np(fq, &np);
        if (err)
                return err;
        if ((np.state & QM_MCR_NP_STATE_MASK) == QM_MCR_NP_STATE_OOS)
                return 1;
        return 0;
}
```

7.4.2.2.7 QMan CEETM APIs

CEETM is a version of egress traffic management in QMan that provides hierarchical class based scheduling and traffic shaping. It is intended for use on links leading to a Wide Area Network (WAN).

7.4.2.2.7.1 QMan CEETM Device-Tree Node

The QMan driver determines the available CEETM resources from the device tree. The definition of CEETM node is defined as:

```
                qman-ceetm@0 {
                       compatible = "fsl,qman-ceetm";
                       fsl,ceetm-lfqid-range = <0xf00000 0x1000>;
                       fsl,ceetm-sp-range = <0 12>;
                       fsl,ceetm-lni-range = <0 8>;
                       fsl,ceetm-channel-range = <0 32>;
                };
```

Each *-range node will specify the CEETM resource(lfqid/sp/lni/cq channel) by a 2-cell value <x y>, in which x is the first value and y is the total number.

There are two ceetm device trees under arch/powerpc/boot/dts/fsl: qoriq-qman-ceetm0.dtsi and qoriq-qman-ceetm1.dts, which are used for DCP0 and DCP1 CEETM resources separately. To enable the CEETM initialization routine, the ceetm device tree should be added in the board's device tree.

For example, in t4240qds.dts, add the following lines at the bottom:

```
/include/ "fsl/qoriq-qman-ceetm0.dtsi"
/include/ "fsl/qoriq-qman-ceetm1.dtsi"
```

In b4860qds.dts, add the following line at the bottom: (because b4860 only support CEETM instance on DCP0).

```
/include/ "fsl/qoriq-qman-ceetm0.dtsi"
```

7.4.2.2.7.2 The token rate of CEETM shaper

7.4.2.2.7.2.1 The token rate structure

The QMan CEETM provides two rate shapers – CR and ER shapers to shape the traffic with the given rate. Both shapers use the token credit rate and credit update reference period to determine the shaper's output rate (in bits/second). The token rate is specified in bytes with an 11 bit integer and a 13 bit fractional part, and can be configured via CEETM shaper configuration commands. Here is the definition of the token rate structure:

```
/* Token Rate Structure
/* The shaping rates are based on a "credit" system and a pre-configured h/w
 * internal timer. The following type represents a shaper "rate" parameter as a
 * fractional number of "tokens". Here's how it works. This (fractional) number
 * of tokens is added to the shaper's "credit" every time the h/w timer elapses
 * (up to a limit which is set by another shaper parameter). Every time a frame
 * is enqueued through a shaper, the shaper deducts as many tokens as there are
 * bytes of data in the enqueued frame. A shaper will not allow itself to
 * enqueue any frames if its token count is negative. As such:
 *
 *         The rate at which data is enqueued is limited by the
 *         rate at which tokens are added.
 *
 * Therefore if the user knows the period between these h/w timer updates in
 * seconds, they can calculate the maximum traffic rate of the shaper (in
```

```
 * bytes-per-second) from the token rate. And vice versa, they can calculate
 * the token rate to use in order to achieve a given traffic rate.
 */
struct qm_ceetm_rate {
        /* The token rate is; whole + (fraction/8192) */
        u32 whole:11; /* 0..2047 */
        u32 fraction:13; /* 0..8191 */
};
```

7.4.2.2.7.2.2 The APIs to convert token rate and shapers output rate

The suggested value for credit update reference period is 1000ns as stated in the PRES field's description of CEETM_CFG_PRES register in Reference Manual. The calculation for shapers output rate is based on:

*shaper output rate (in bits/sec) = token credit rate * 8 / credit update reference period*

For more detail, please refer to the Appendix in this document. The following APIs are used to convert shaper output rate in bps to token rate and vice versa, which can be applied to both LNI and channel shaping:

```
/**
 * qman_ceetm_bps2tokenrate - Given a desired rate shaper rate measured in bps
 * (ie. bits-per-second), compute the 'token_rate' fraction that best
 * approximates that rate.
 *
 * @bps: the input shaper rate in bps.
 * @token_rate: the output token rate computed with the given bps.
 * @rounding: dictates how to round if an exact conversion is not possible; if
 * it is negative then 'token_rate' will round down to the highest value that
 * does not exceed the desired rate, if it is positive then 'token_rate' will
 * round up to the lowest value that is greater than or equal to the desired
 * rate, and if it is zero then it will round to the nearest approximation,
 * whether that be up or down.
 *
 * Returns zero for success, or -EINVAL if prescaler or qman clock is not available.
 */
int qman_ceetm_bps2tokenrate(u32 bps,
                             struct qm_ceetm_rate *token_rate,
                             int rounding);

/**
 * qman_ceetm_tokenrate2bps - Given a 'token_rate', compute the corresponding
 * number of 'bps'.
 * @token_rate: the input token rate fraction.
 * @bps: the output shaper rate in bps.
 * @rounding: has the same semantics as the previous function.
 *
 * Return zero for success, or -EINVAL if prescaler or qman clock is not available.
 */
int qman_ceetm_tokenrate2bps(const struct qm_ceetm_rate *token_rate,
                             u32 *kbps,
                             int rounding);
```

7.4.2.2.7.3 CEETM Sub-portal

CEETM is implemented as a mode of scheduling for sub-portals on specific QMan Direct-Connect Portals. We need to claim the sub-portal on a DCP to support CEETM.

7.4.2.2.7.3.1 Claim/release sub-portal

```
/* *
 * qman_ceetm_sp_claim - Claims the given sub-portal, provided it is available
 * to use and configured for traffic-management.
 * @sp: the returned sub-portal object, if successful.
 * @dcp_id:  specifies the desired FMan block (and thus the relevant CEETM instance).
 * The type enum qm_dc_portal has already been defined in fsl_qman.h
 * @sp_idx:  is the desired sub-portal index from 0 to 15.
 *
 * Returns zero for success, or -ENODEV if the sub-portal is in use, or -EINVAL
 * if the sp_idx is out of range.
 *
 *
 * Note that if there are multiple driver domains (eg. a linux kernel versus
 * user-space drivers in USDPAA, or multiple guests running under a
 * hypervisor) then a sub-portal may be accessible by more than one instance
 * of a qman driver and so it may be claimed multiple times. If this is the
 * case, it is up to the system architect to prevent conflicting configuration
 * actions coming from the different driver domains. The qman drivers do not
 * have any behind-the-scenes coordination to prevent this from happening.
 */
int qman_ceetm_sp_claim(struct qm_ceetm_sp **sp,
                        enum qm_dc_portal dcp_id,
                        unsigned int sp_idx);

/**
 * qman_ceetm_sp_release - Releases a previously claimed sub-portal.
 * @sp: the sub-portal to be released.
 *
 * Returns 0 for success, or -EBUSY for failure if the dependencies are not
 * released yet.
 */
int qman_ceetm_sp_release(struct qm_ceetm_sp *sp);
```

7.4.2.2.7.4 CEETM LNI - Logic Network Interface

Each QMan CEETM instance supports up to 8 Logical Network Interfaces (LNIs) which can be mapped to a DCP sub-portal. Each LNI aggregates frames from one or more QMan CEETM channels and priority schedules unshaped frames, CR frames and ER frames It applies a dual rate shaper to aggregate CR/ER frames from shaped channel. The user can enable and disable the shaper, change the shaper rate for LNI, as well as control the CEETM traffic class flow control because it is maintained on a per LNI basis and applied to all class queue channels within the LNI.

7.4.2.2.7.4.1 Claim/release LNI

```
/**
 * qman_ceetm_lni_claim - Claims an unclaimed LNI.
 * @lni: the returned LNI object, if successful.
 * @dcp_id: specifies the desired FMan block (and thus the relevant CEETM instance).
 * @lni_idx: the desired LNI index.
 *
 * Returns zero for success, or -EINVAL for failure, which will happen
 * if the LNI is not available or has already been claimed (and not yet
 * successfully released),or lni_idx is out of range.
```

```
 *
 * Note that there may be multiple driver domains (or instances) that need to transmit
 * out the same LNI, so this claim is only guaranteeing exclusivity within the
 * domain of the driver being called. See qman_ceetm_sp_claim() and
 * qman_ceetm_sp_get_lni() for more information.
 */
int qman_ceetm_lni_claim(struct qm_ceetm_lni **lni,
                         enum qm_dc_portal dcp_id,
                         unsigned int lni_idx);

/**
 * qman_ceetm_lni_release - Releases a previously claimed LNI.
 * @lni: the LNI needs to be released.
 *
 * This will only succeed if all dependent objects have been released.
 * Returns zero for success. Returns -EBUSY if the dependencies are not released.
 */
int qman_ceetm_lni_release(struct qm_ceetm_lni *lni);
```

7.4.2.2.7.4.2 Map sub-portal with LNI

```
/**
 * qman_ceetm_sp_set_lni
 * qman_ceetm_sp_get_lni – Set/get the LNI that the sub-portal is currently mapped to.
 * @sp: the given sub-portal.
 * @lni(in “set” function): the LNI object which the sub-portal will be mapped to.
 * @lni_idx(in “get” function): the LNI index which the sub-portal is mapped to.
 *
 * Returns zero for success. * Returns –EINVAL for “set” function when this sp-lni mapping has
been set, or
 * configure mapping command returns error, and
 * -EINVAL for “get” function when this sp-lni mapping is not set, or the query
 * mapping command returns error.
 *
 * This may be useful in situations where multiple driver
 * domains have access to the same sub-portals in order to all be able to
 * transmit out the same physical interface (perhaps they're on different IP
 * addresses or VPNs, so FMan is splitting Rx traffic and here we need to
 * converge Tx traffic). In that case, a control-plane is likely to use
 * qman_ceetm_lni_claim() followed by qman_ceetm_sp_set_lni() to configure the
 * sub-portal, and other domains are likely to use qman_ceetm_sp_get_lni()
 * followed by qman_ceetm_lni_claim() in order to determine the LNI that the
 * control-plane had assigned. This is why the "get" returns an index, whereas
 * the "set" takes an (already claimed) LNI object.
 */
int qman_ceetm_sp_set_lni(struct qm_ceetm_sp *sp,
                          struct qm_ceetm_lni *lni);
int qman_ceetm_sp_get_lni(struct qm_ceetm_sp *sp,
                          unsigned int *lni_idx);
```

7.4.2.2.7.4.3 Configure LNI shaper

The LNI provides two rate shapers that can be used to shape the traffic from all class queue channels which are shaped on input to the channel scheduler. The LNI shapers are used to shape or rate limit the aggregate of the class queue channels

which have each been shaped. The two shaper rates are applied only to frames shaped by the corresponding shaper at the channel shaper level.

```
/**
 * qman_ceetm_lni_enable_shaper
 * qman_ceetm_lni_disable_shaper - Enables/disables shaping on the LNI.
 * @lni: the given LNI.
 * @coupled: indicates whether CR and ER shapers are coupled.
 * @oal: the overhead accounting length which is added to the actual length of
 * each frame when performing shaper calculations.
 *
 * When the number of (unused) committed-rate tokens reach the committed-rate
 * token limit, @coupled indicates whether surplus tokens should be added to
 * the excess-rate token count (up to the excess-rate token limit). Whenever
 * a claimed LNI is first enabled for shaping, its committed and excess token
 * rates and limits are zero, so will need to be changed to do anything useful.
 * The shaper can subsequently be enabled/disabled without resetting the shaping
 * parameters, but the shaping parameters will be reset when the LNI is released.
 *
 * Returns 0 for success, or errno for “enable” function in the cases as:
 * a) -EINVAL if the shaper is already enabled.
 * b) -EIO if the configure shaper command returns error.
 * For “disable” function, returns:
 * a) -EINVAL if the shaper has been disabled.
 * b) -EIO if the query shaper command returns error.
 */
int qman_ceetm_lni_enable_shaper(struct qm_ceetm_lni *, int coupled, int oal);
int qman_ceetm_lni_disable_shaper(struct qm_ceetm_lni *);

/**
 * qman_ceetm_lni_is_shaper_enabled - Check LNI shaper status
 * @lni: the give LNI
 */
int qman_ceetm_lni_is_shaper_enabled(struct qm_ceetm_lni *lni);


/**
 * qman_ceetm_lni_set_commit_rate
 * qman_ceetm_lni_get_commit_rate
 * qman_ceetm_lni_set_excess_rate
 * qman_ceetm_lni_get_excess_rate - Set/get the shaper CR/ER token rate and token
 * limit of the given LNI.
 * @lni: the given LNI.
 * @token_rate: the desired token rate for “set” function, or the token rate of
 * the LNI queried by “get” function.
 * @token_limit: the desired token limit for “set” function, or the token limit of
 * the LNI queried by “get” function.
 *
 * Returns zero for success. The “set” function returns  -EINVAL if the given
 * LNI is unshaped, or -EIO if the configure shaper command returns error.
 * The “get” function returns -EINVAL if the token rate or token limit is not set,
 * or the query command returns error
 */
int qman_ceetm_lni_set_commit_rate(struct qm_ceetm_lni *,
                           const struct qm_ceetm_rate *token_rate,
                           u16 token_limit);
int qman_ceetm_lni_get_commit_rate(struct qm_ceetm_lni *,
                           struct qm_ceetm_rate *token_rate,
```

```
                          u16 *token_limit);
int qman_ceetm_lni_set_excess_rate(struct qm_ceetm_lni *,
                          const struct qm_ceetm_rate *token_rate,
                          u16 token_limit);
int qman_ceetm_lni_get_excess_rate(struct qm_ceetm_lni *,
                          struct qm_ceetm_rate *token_rate,
                          u16 *token_limit);

/**
 * qman_ceetm_lni_set_commit_rate_bps
 * qman_ceetm_lni_get_commit_rate_bps
 * qman_ceetm_lni_set_excess_rate_bps
 * qman_ceetm_lni_get_excess_rate_bps - Set/get the shaper CR/ER rate
 * and token limit for the given LNI.
 * @lni: the given LNI.
 * @bps: the desired shaping rate in bps for "set" function, or the shaping rate
 * of the LNI queried by "get" function.
 * @token_limit: the desired token bucket limit for "set" function, or the token
 * limit of the given LNI queried by "get" function.
 *
 * Returns zero for success. The "set" function returns -EINVAL if the given
 * LNI is unshaped or -EIO if the configure shaper command returns error.
 * The "get" function returns -EINVAL if the token rate or the token limit is
 * not set or the query command returns error.
 */
int qman_ceetm_lni_set_commit_rate_bps(struct qm_ceetm_lni *lni,
                       u64 bps,
                       u16 token_limit);
int qman_ceetm_lni_get_commit_rate_bps(struct qm_ceetm_lni *lni,
                       u64 *bps, u16 *token_limit);
int qman_ceetm_lni_set_excess_rate_bps(struct qm_ceetm_lni *lni,
                       u64 bps,
                       u16 token_limit);
int qman_ceetm_lni_get_excess_rate_bps(struct qm_ceetm_lni *lni,
                       u64 *bps, u16 *token_limit);
```

7.4.2.2.7.4.4 Configure LNI traffic class flow control

```


/**
 * qman_ceetm_lni_set_tcfcc -configure "Traffic Class Flow Control".
 * qman_ceetm_lni_get_tcfcc – query “Traffic Class Flow Control”.
 * @lni: the given LNI.
 * @cq_level: between 0 and 15, representing individual class queue levels (CQ0 to
 * CQ7 for every channel) and grouped class queue levels (CQ8 to CQ15 for every
 * channel).
 * @traffic_class: between 0 and 7 when associating a given class queue level to a
 * traffic class, or -1 when disabling traffic class flow control for this class
 * queue level.
 *
 * Returns zero for success, or -EINVAL if the cq_level or traffic_class is out of
 * the range or -EIO if configure/query tcfcc command returns error.
 *
 * Refer to the section of QMan CEETM traffic class flow control in the
 * reference manual.
 *
 */
```

```
int qman_ceetm_lni_set_tcfcc(struct qm_ceetm_lni *lni,
                             unsigned int cq_level,
                             int traffic_class);
int qman_ceetm_lni_get_tcfcc(struct qm_ceetm_lni *lni,
                             unsigned int *cq_level,
                             int *traffic_class);
```

7.4.2.2.7.5 CEETM class queue channel

Each instance of CEETM supports 32 class queue channels for allocation across the 8 LNIs. Each channel can be configured to deliver frames to any one of the LNIs, can be configured as shaped or unshaped channel. When shaped, a dual-rate shaper applies to the aggregate of CR/ER frames from the channel. Each channel contains a total of 16 class queues (CQ), with 8 independent classes and 8 grouped classes which can be configured as 1 group of 8 classes or 2 groups of 4 classes. The weighted bandwidth fairness scheduling applies within the grouped classes, and strict priority scheduling applies to 8 independent classes and 2 class groups. Once the channel is configured as shaped, the 8 independent classes and 2 class groups can be configured to supply CR frames, ER frames or both. The channel is configured independently from other channels.

7.4.2.2.7.5.1 Claim/release class queue channel

```
/**
 * qman_ceetm_channel_claim - Claims an unclaimed CQ channel that is mapped to the
 * given LNI.
 * @channel: the returned class queue channel object, if successful.
 * @lni: the LNI that the channel belongs to.
 *
 * Channels are always initially "unshaped".
 *
 * Returns zero for success, or -ENODEV if there is no channel available(all 32
 * channels are claimed) or -EINVAL if the channel mapping command returns error.
 */
int qman_ceetm_channel_claim(struct qm_ceetm_channel **channel,
                             struct qm_ceetm_lni *lni);

/**
 * qman_ceetm_channel_release - Releases a previously claimed CQ channel.
 * @channel: the channel needs to be released.
 *
 * Returns zero for success, or -EBUSY if the dependencies are still
 * in use. Note any shaping of the channel will be
 * cleared to leave it in an unshaped state.
 */
int qman_ceetm_channel_release(struct qm_ceetm_channel *channel);
```

7.4.2.2.7.5.2 Configure the shaper of class queue channel

```
/**
 * qman_ceetm_channel_enable_shaper
 * qman_ceetm_channel_disable_shaper - Enable/Disable shaping on the given channel.
 * @channel: the given channel.
 * @coupled: indicates whether surplus tokens should be added to the excess-rate
 * token count (up to the excess-rate token limit) when the number of (unused)
 * committed-rate tokens reach the committed-rate token limit.
 *
 * Whenever a claimed channel is first enabled for shaping, its committed and
 * excess token rates and limits are zero, so will need to be changed to do
```

```
 * anything useful. The shaper can subsequently be enabled/disabled without
 * resetting the shaping parameters, but the shaping parameters will be reset
 * when the channel is released.
 *
 * Returns 0 for success and -EINVAL for failure if the channel shaping has been
 * enabled or disabled, or the management command returns error
 */
int qman_ceetm_channel_enable_shaper(struct qm_ceetm_channel *channel, int coupled);
int qman_ceetm_channel_disable_shaper(struct qm_ceetm_channel *channel);


/**
 * qman_ceetm_channel_is_shaper_enabled - Check channel shaper status.
 * @channel: the give channel.
 */
int qman_ceetm_channel_is_shaper_enabled(struct qm_ceetm_channel *channel);


/**
 * qman_ceetm_channel_set_commit_rate
 * qman_ceetm_channel_get_commit_rate
 * qman_ceetm_channel_set_excess_rate
 * qman_ceetm_channel_get_excess_rate – Set/get the channel shaper CR/ER token rate
 * and token limit.
 * @channel: the given channel.
 * @token_rate: the desired token rate for “set” function, or the queried token rate
 * for “get”function.
 * @token_limit: the desired token limit for “set” function, or the queried token
 * limit for “get” function.
 *
 * Returns zero for success. The “set” function returns –EINVAL if the channel
 * is unshaped or –EIO if the configure shaper command returns error. The
 * “get” function returns -EINVAL if the token rate or token limit is not set, or
 * the query shaper command returns error.
 */
int qman_ceetm_channel_set_commit_rate(struct qm_ceetm_channel *channel,
                                const struct qm_ceetm_rate *token_rate,
                                u16 token_limit);
int qman_ceetm_channel_get_commit_rate(struct qm_ceetm_channel *channel,
                                struct qm_ceetm_rate *token_rate,
                                u16 *token_limit);
int qman_ceetm_channel_set_excess_rate(struct qm_ceetm_channel *channel,
                                const struct qm_ceetm_rate *token_rate,
                                u16 token_limit);
int qman_ceetm_channel_get_excess_rate(struct qm_ceetm_channel *channel,
                                struct qm_ceetm_rate *token_rate,
                                u16 *token_limit);

/**
 * qman_ceetm_channel_set_commit_rate_bps
 * qman_ceetm_channel_get_commit_rate_bps
 * qman_ceetm_channel_set_excess_rate_bps
 * qman_ceetm_channel_get_excess_rate_bps - Set/get channel CR/ER shaper
 * parameters.
 * @channel: the given channel.
 * @token_rate: the desired shaper rate in bps for "set" function, or the
 * shaper rate in bps for "get" function.
 * @token_limit: the desired token limit for "set" function, or the queried
 * token limit for "get" function.
 *
```

```
 * Return zero for success. The "set" function returns -EINVAL if the channel
 * is unshaped, or -EIO if the configure shaper command returns error. The
 * "get" function returns -EINVAL if token rate of token limit is not set, or
 * the query shaper command returns error.
 */
int qman_ceetm_channel_set_commit_rate_bps(struct qm_ceetm_channel *channel,
                        u64 bps, u16 token_limit);
int qman_ceetm_channel_get_commit_rate_bps(struct qm_ceetm_channel *channel,
                        u64 *bps, u16 *token_limit);
int qman_ceetm_channel_set_excess_rate_bps(struct qm_ceetm_channel *channel,
                        u64 bps, u16 token_limit);
int qman_ceetm_channel_get_excess_rate_bps(struct qm_ceetm_channel *channel,
                        u64 *bps, u16 *token_limit);
```

7.4.2.2.7.5.3 Configure the token limit as the weight for the unshaped channel

QMan CEETM uses an algorithm called unshaped fair queuing (uFQ) for the unshaped channel. The algorithm allows unshaped channels to be included in the CR time eligible list, and thus use the configured commit rate token bucket limit value as their fair queuing weight.

```
/**
 * qman_ceetm_channel_set_weight
 * qman_ceetm_channel_get_weight - Set/get the weight for unshaped channel.
 * @channel: the given channel.
 * @token_limit: the desired token limit as the weight of unshaped channel for “set”
 * function, or the queried token limit for “get” function.
 *
 * Returns zero for success, or -EINVAL if the channel is a shaped channel, or
 * the management command returns error
 */
int qman_ceetm_channel_set_weight(struct qm_ceetm_channel *channel,
                                  u16 token_limit);
int qman_ceetm_channel_get_weight(struct qm_ceetm_channel *channel,
                                  u16 *token_limit);
```

7.4.2.2.7.5.4 Set CR/ER eligibility for CQs within a CEETM channel

The APIs below allow the user to set the 8 independent CQs and 2 CQ groups within a shaped CEETM channel to be CR and/or ER eligible.

```
/**
 * qman_ceetm_channel_set_group_cr_eligibility
 * qman_ceetm_channel_set_group_er_eligibility - Set channel group eligibitity
 * @channel: the given channel object
 * @group_b: indicates whether there is group B in this channel.
 * @cre: the commit rate eligibility, 1 for enable, 0 for disable.
 *
 * Return zero for success, or -EINVAL if eligiblity setting fails.
*/
int qman_ceetm_channel_set_group_cr_eligiblility(struct qm_ceetm_channel
*channel, int group_b, int cre);
int qman_ceetm_channel_set_group_er_eligiblility(struct qm_ceetm_channel
*channel, int group_b, int ere);
```

```
/**
 * qman_ceetm_channel_set_cq_cr_eligibility
 * qman_ceetm_channel_set_cq_er_eligibility - Set channel cq eligibitity
 * @channel: the given channel object
 * @idx: is from 0 to 7 (representing CQ0 to CQ7).
 * @cre: the commit rate eligibility, 1 for enable, 0 for disable.
 *
 * Return zero for success, or -EINVAL if eligiblity setting fails.
*/
int qman_ceetm_channel_set_cq_cr_eligiblility(struct qm_ceetm_channel *channel,
unsigned int idx, int cre);
int qman_ceetm_channel_set_cq_er_eligiblility(struct qm_ceetm_channel *channel,
unsigned int idx, int ere);
```

7.4.2.2.7.6 CEETM class queue

Each CEETM class has a dedicated class queue (CQ), it can have dedicated or shared Class Congestion Group Record.

7.4.2.2.7.6.1 Claim/release CQ

```
/**
 * qman_ceetm_cq_claim - Claims an individual class queue.
 * @cq: the returned class queue object, if successful.
 * @channel: the given class queue channel.
 * @idx: is from 0 to 7 (representing CQ0 to CQ7).
 * @ccg: represents the class congestion group that this class queue
 * should be subscribed to, or NULL if no congestion group membership is desired.
 *
 * Returns zero for success, or -EINVAL if @idx is out of range 0 to 7 or this class
 * queue has been claimed, or configure class queue command returns error, or
 * returns -ENOMEM if allocating CQ memory fails.
 */
int qman_ceetm_cq_claim(struct qm_ceetm_cq **cq,
                        struct qm_ceetm_channel *channel,
                        unsigned int idx,
                        struct qm_ceetm_ccg *ccg);

/**
 * qman_ceetm_cq_claim_A - Claims a class queue within the channel group A.
 * @cq: the returned class queue object, if successful.
 * @channel: the given class queue channel.
 * @idx: If the channel is configured for 1 group only, @idx is from 8 to 15 (CQ8 to
 * CQ15) and only group A exists, otherwise @idx is from 8 to 11 (CQ8 to CQ11 for
 * group A).
 * @ccg: represents the class congestion group that this class queue should be
 * subscribed to, or NULL if no ccg is desired.
 *
 * Returns zero for success, or -EINVAL if @idx is out of range or if the class
 * queue has been claimed, or configure class queue command returns error, or
 * returns -ENOMEM if allocating CQ memory fails.
 */
int qman_ceetm_cq_claim_A(struct qm_ceetm_cq **cq,
                          struct qm_ceetm_channel *channel,
                          unsigned int idx,
                          struct qm_ceetm_ccg *ccg);
```

```
/**
 * qman_ceetm_cq_claim_B - Claims a class queue within the channel group B.
 * @cq: the returned class queue object, if successful.
 * @channel: the given class queue channel.
 * @idx: if the channel is configured with 2 groups, 'idx' is from 12 to 15(CQ12 to
 * CQ15 for group B).
 * @ccg: represents the class congestion group that this class queue should be
 * subscribed to, or NULL if no ccg is desired.
 *
 * Returns zero for success, or -EINVAL if @idx is out of range  or the class
 * queue has been claimed, or configure class queue command returns error, or
 * returns -ENOMEM if allocating CQ memory fails.
 */
int qman_ceetm_cq_claim_B(struct qm_ceetm_cq **cq,
                          struct qm_ceetm_channel *channel,
                          unsigned int idx,
                          struct qm_ceetm_ccg *ccg);

/**
 * qman_ceetm_cq_release - Releases a previously claimed class queue.
 * @cq: the class queue to be released.
 *
 * This will only succeed if all dependent objects (eg. logical FQIDs) have been
 * released.
 * Returns zero for success or -EBUSY for failure if the dependencies are not
 * released (e.g. LFQID is not released)
 */
int qman_ceetm_cq_release(struct qm_ceetm_cq *cq);

/**
 * qman_ceetm_drain_cq - drain the CQ till it is empty.
 * @cq: the give CQ object.
 * Return 0 for success or -EINVAL for unsuccessful command to empty CQ.
 */
int qman_ceetm_drain_cq(struct qm_ceetm_cq *cq);
```

7.4.2.2.7.6.2 Change CQ weight

```
/**
 * qman_ceetm_queue_set_weight
 * qman_ceetm_queue_get_weight - Configure/query the weight ofa grouped class queue.
 * @cq: the given class queue.
 * @weight_code: the desired weight code to change for this given class queue for
 * "set" function or the queried weight code of the give class queue for "get"
 * function.
 *
 * Grouped class queues have a default weight code of zero, which corresponds to
 * a scheduler weighting of 1. This function can be used to modify a grouped
 * class queue to another weight, valid values are from 0 to 255. (Use the
 * helpers qman_ceetm_wbfs2ratio() and qman_ceetm_ratio2wbfs() to convert
 * between these 'weight_code' values and the corresponding sharing weight.) As
 * the weight code ranges from 0 to 255, the corresponding scheduling weight
 * ranges from 1.00 to 248 in pseudo-exponential steps).
 *
 * Returns zero for success or -EIO if the configure  weight code command returns
 * error for "set" function, or -EINVAL if the query command returns error for "get"
 * function.
```

```
 * Please refer to section “CEETM Weighted Scheduling among Grouped Classes” in
 * Reference Manual for weight and weight code.
 */
int qman_ceetm_queue_set_weight(struct qm_ceetm_cq *cq,
                                struct qm_ceetm_weight_code *weight_code);
int qman_ceetm_queue_get_weight(struct qm_ceetm_cq *cq,
                                struct qm_ceetm_weight_code *weight_code);


/**
 * qman_ceetm_wbfs2ratio - Given a weight code ('wbfs'), an accurate fractional
 * representation of the corresponding weight is given (in order to not lose
 * any precision).
 * @weight_code: The given weight code in WBFS.
 * @numerator: the numerator part of the weight computed by the weight code.
 * @denominator: the denominator part of the weight computed by the weight code
 *
 * Returns zero for success, or -EINVAL if the given weight code is illegal.
 */
int qman_ceetm_wbfs2ratio(struct qm_ceetm_weight_code *weight_code,
                          u32 *numerator,
                          u32 *denominator);

/**
 * qman_ceetm_ratio2wbfs - Given a weight, find the nearest possible weight code.
 * @numerator: numerator part of the given weight.
 * @denominator: denominator part of the given weight.
 * @weight_code: the weight code computed from the given weight.
 *
 * If the user needs to know how close this is, convert the resulting weight code
 * back to a weight and compare.
 *
 * Returns zero for success, or -ERANGE if "numerator/denominator" is outside the
 * range of weights.
 */
int qman_ceetm_ratio2wbfs(u32 numerator,
                          u32 denominator,
                          struct qm_ceetm_weight_code *weight_code);

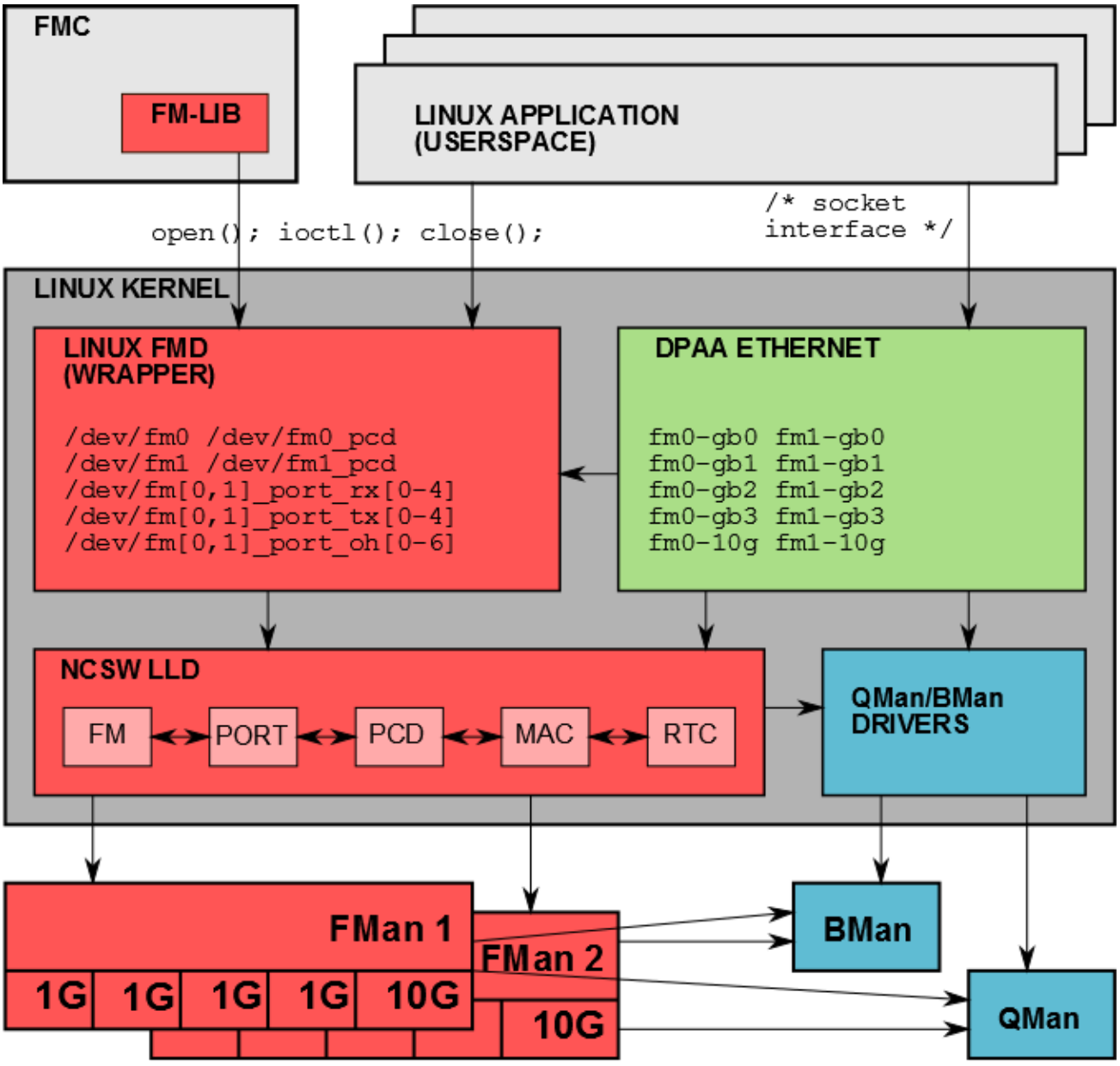
/**
 * qman_ceetm_set_queue_weight_in_ratio
 * qman_ceetm_get_queue_weight_in_ratio - Configure/query the weight of a
 * grouped class queue.
 * @cq: the given class queue.
 * @ratio: the weight in ratio. It should be the real ratio number multiplied
 * by 100 to get rid of fraction. User needs to check the "WBFS weight code
 * to weight mapping" table in BG for the possible weight values to be used.
 *
 * Returns zero for success, or -EIO if the configure weight command returns
 * error for "set" function, or -EINVAL if the query command returns
 * error for "get" function.
 */
int qman_ceetm_set_queue_weight_in_ratio(struct qm_ceetm_cq *cq, u32 ratio);
int qman_ceetm_get_queue_weight_in_ratio(struct qm_ceetm_cq *cq, u32 *ratio);
```

7.4.2.2.7.6.3 Query CQ statistics

```
/**
 * qman_ceetm_cq_get_dequeue_statistics - Get the statistics provided by CEETM
 * CQ counters.
 * @cq: the given CQ object.
 * @flags: indicates whether the statistics counter will be cleared after query.
 * @frame_count: The number of the frames that have been counted since the
 * counter was cleared last time.
 * @byte_count: the number of bytes in all frames that have been counted.
 *
 * Return zero for success or -EINVAL if query statistics command returns error.
 *
 */
int qman_ceetm_cq_get_dequeue_statistics(struct qm_ceetm_cq *cq, u32 flags,
                                         u64 *frame_count, u64 *byte_count);
```

7.4.2.2.7.7 CEETM logical FQID

Each class queue in CEETM mode is identified via a “logical frame queue identifier (LFQID)” to maintain semantic compatibility with enqueue commands to FQs (non-CEETM queues). The upper 1M FQIDs is used for the LFQID, and each DCP owns 4K LFQIDs. Any enqueue to these LFQIDs will be directed to the CEETM logic used by one of the DCP portals.

7.4.2.2.7.7.1 Claim/release LFQID

```
/**
 * qman_ceetm_lfq_claim - Claims an unused logical FQID, associates it with the
 * given class queue.
 * @lfq: the returned lfq object, if successful.
 * @cq: the given class queue which needs to claim a LFQID.
 *
 * Returns 0 for success, or -ENODEVif no LFQID is available, or -ENOMEM if
 * allocating memory for lfq fails, or -EINVAL if configuring LFQMT fails
 */
int qman_ceetm_lfq_claim(struct qm_ceetm_lfq **lfq,
                         struct qm_ceetm_cq *cq);

/**
 * qman_ceetm_lfq_release -  Releases a previously claimed logical FQID.
 * @lfq: the logic fq to be released.
 *
 * Return zero for success. The failure condition is TBD.
 */
int qman_ceetm_lfq_release(struct qm_ceetm_lfq *lfq);
```

7.4.2.2.7.7.2 Configure/Query Dequeue Context Table

```
/**
 * qman_ceetm_lfq_set_context
 * qman_ceetm_lfq_get_context - Set/get the context_a/context_b pair to the
```

```
 * "dequeue context table" associated with the logical FQID.
 * @lfq: the given lfq object.
 * @context_a: contextA of the dequeue context.
 * @context_b: contextB of the dequeue context.
 *
 * Returns zero for success, or -EINVAL if there is error to set/get the context
 * pair.
 */
int qman_ceetm_lfq_set_context(struct qm_ceetm_lfq *lfq,
                               u64 context_a,
                               u32 context_b);
int qman_ceetm_lfq_get_context(struct qm_ceetm_lfq *lfq,
                               u64 *context_a,
                               u32 *context_b);
```

7.4.2.2.7.7.3 Create FQ for LFQ

```
/**
 * qman_ceetm_create_fq - Initialise a FQ object for the LFQ.
 * @lfq: the given logic FQ.
 * @fq: the FQ object created for the LFQ
 *
 * The FQ object can be used in qman_enqueue() and qman_enqueue_orp() APIs to target
 * a logical FQID (and the class queue it is associated with). Note that this FQ
 * object can only be used for enqueues, and in the case of qman_enqueue_orp()
 * it can not be used as the 'orp' parameter, only as 'fq'. This FQ object can not
 * (and shouldn't) be destroyed, it is only valid as long as the underlying 'lfq'
 * remains claimed. It is the user's responsibility to ensure that the underlying
 * 'lfq' is not released until any enqueues to this FQ object have completed.
 * The only field the user needs to fill in is fq->cb.ern, as that enqueue rejection
 * handler is the callback that could conceivably be called on this FQ object. This
 * API can be called multiple times to create multiple FQ objects referring to the
 * same logical FQID, and any enqueue rejections will respect the callback of the
 * object that issued the enqueue (and will identify the object via the parameter
 * passed to the callback too). There is no 'flags' parameter to this API as there
 * is for qman_create_fq() - the created FQ object behaves as though
 * qman_create_fq() had been called with the single flag QMAN_FQ_FLAG_NO_MODIFY.
 *
 * Returns 0 for success. The failure case is TBD
 */
int qman_ceetm_create_fq(struct qm_ceetm_lfq *lfq, struct qman_fq *fq);
```

7.4.2.2.7.8 CEETM Class Congestion Group(CCG)

CEETM supports both Weighted Random Early Discard (WRED) and tail drop congestion management with its 512 Class Congestion Groups (CCGs). The CCG are divided into channels. Each channel contains 16 CCGs, and each CCG channel is tied one-to-one with a CQ channel. The CCG to be used for a particular CQ can be assigned to any of the 16 CCG within the channel. The description of CCG can be found at section “CEETM Class Congestion Management and Avoidance” in the Reference Manual.

7.4.2.2.7.8.1 Claim/release CCG

```
/**
 * qman_ceetm_ccg_claim - Claims an unused CCG.
```

```
 * @ccg: the returned CCG object, if successful.
 * @idx: the ccg index from 0-15 within the given channel.
 * @channel: the given class queue channel which tied to the CCG channel.
 * @cscn: the callback function of this CCG.
 * @cb_ctx: specify the callback and corresponding context to be used if state
 * change notifications are later enabled for this CCG.
 *
 * The congestion group is local to the given class queue channel, so only
 * class queues within the channel can be associated with that congestion
 * group. The association of class queues to congestion groups occurs when the
 * class queues are claimed, see qman_ceetm_cq_claim() and related functions.
 * Congestion groups are in a "zero" state when initially claimed, and they are
 * returned to that state when released.
 *
 * Returns 0 for success, or -EINVAL if no CCG is available.
 */
int qman_ceetm_ccg_claim(struct qm_ceetm_ccg **ccg,
                         struct qm_ceetm_channel *channel,
                         void (*cscn)(struct qm_ceetm_ccg *,
                                      void *cb_ctx,
                                      int congested),
                         void *cb_ctx);

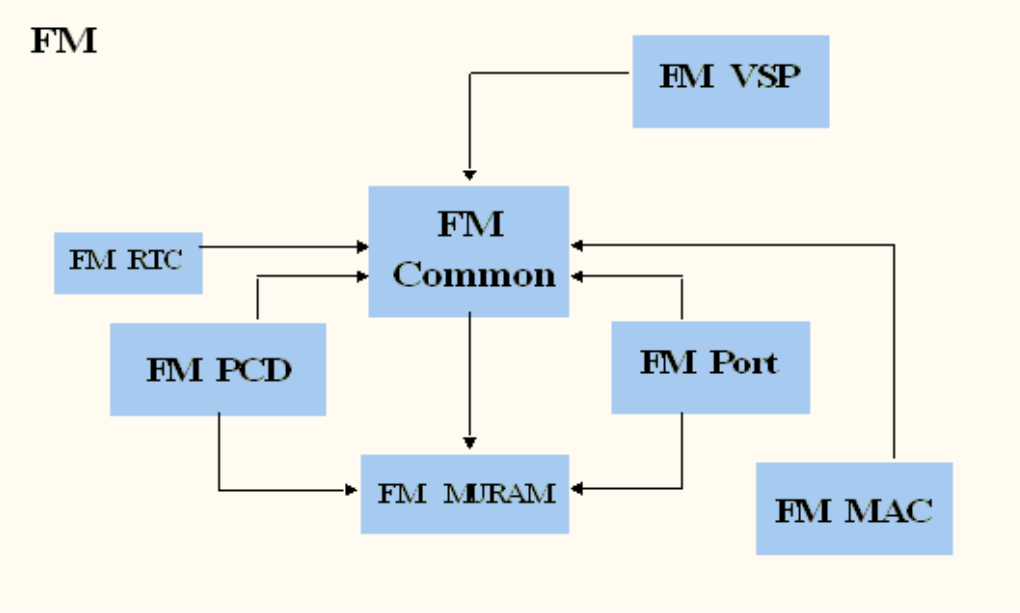
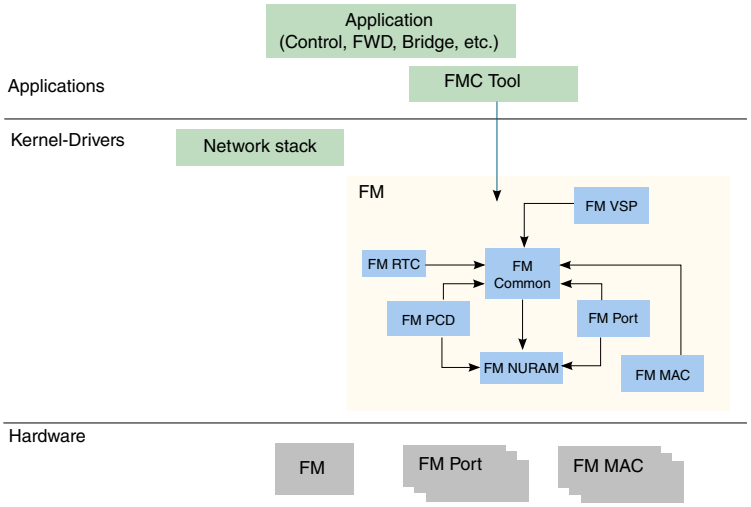
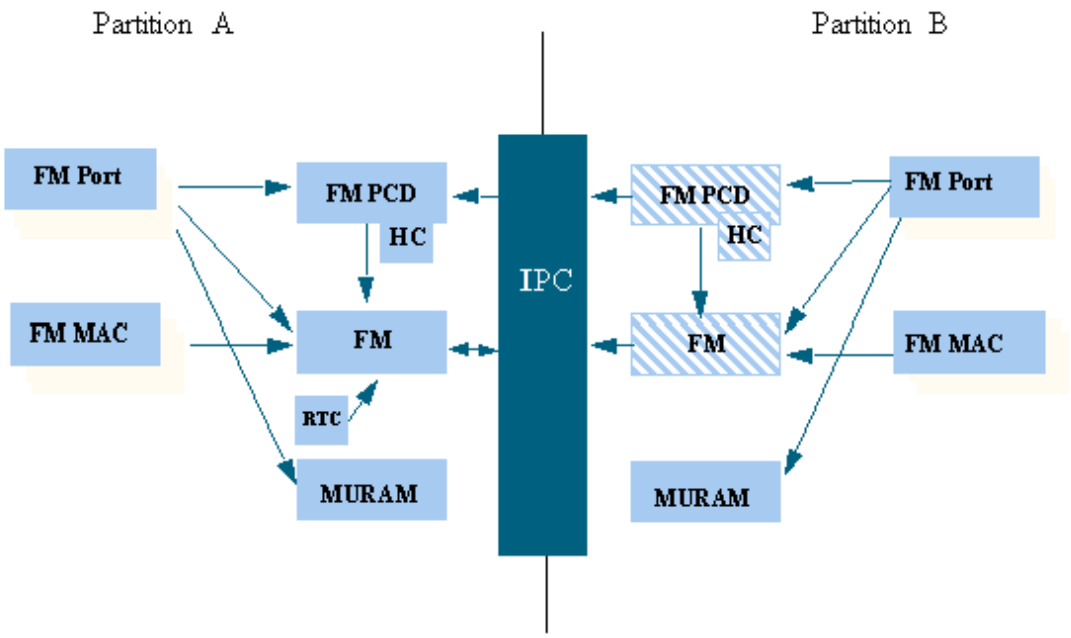
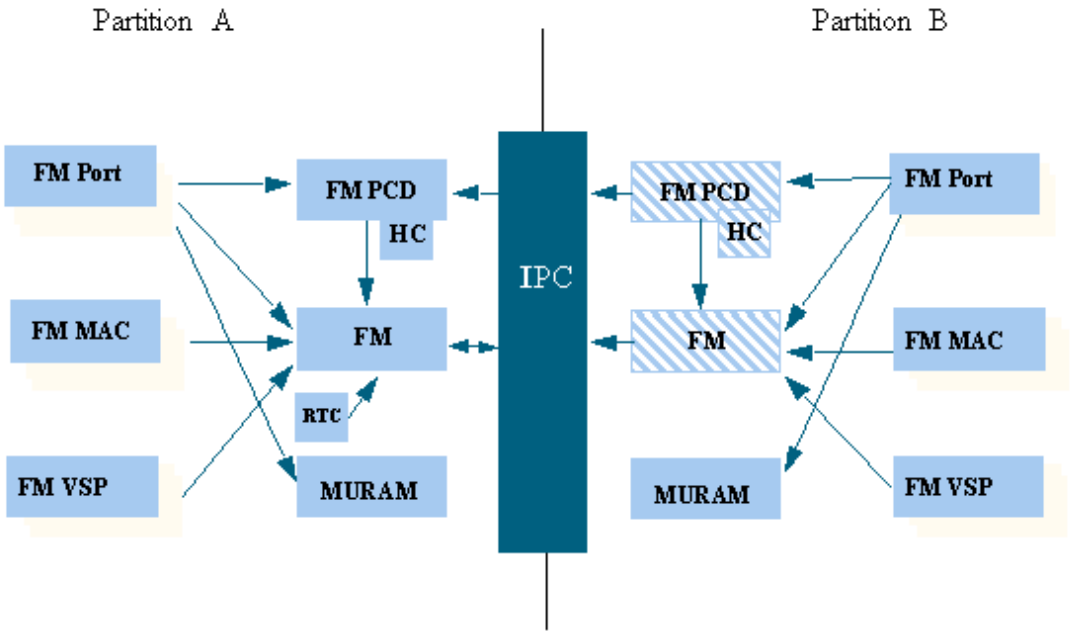
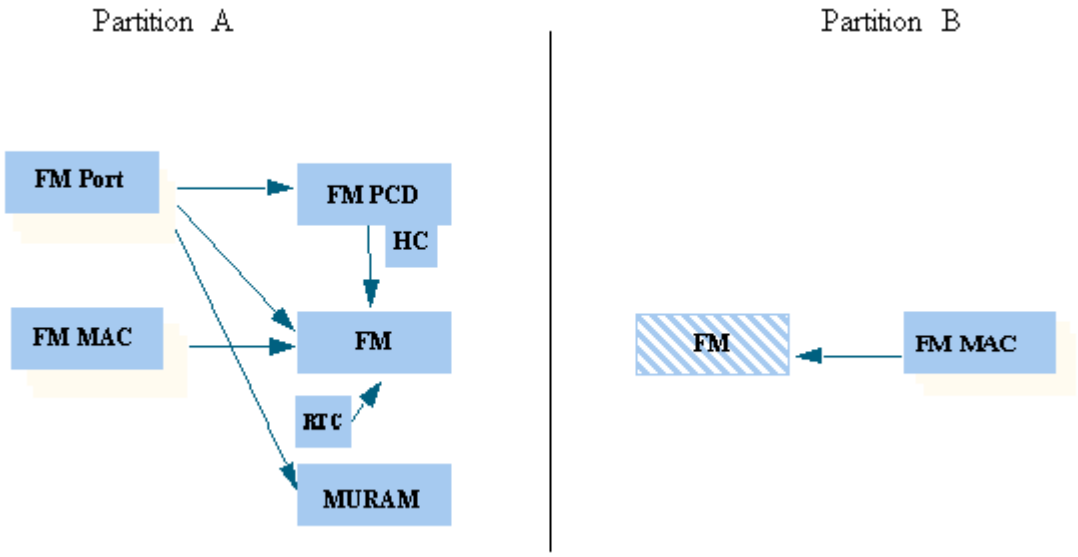
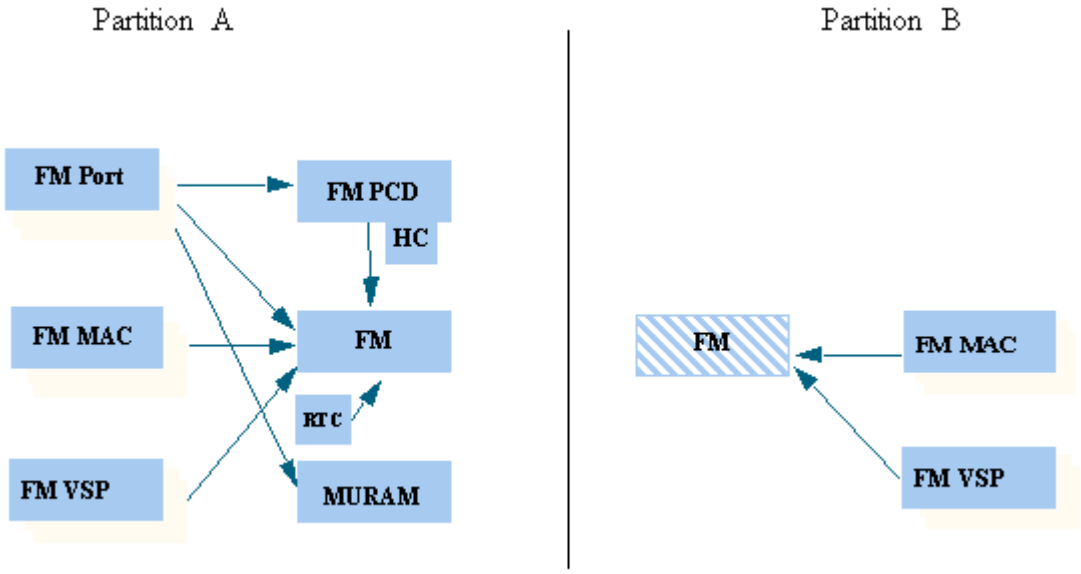
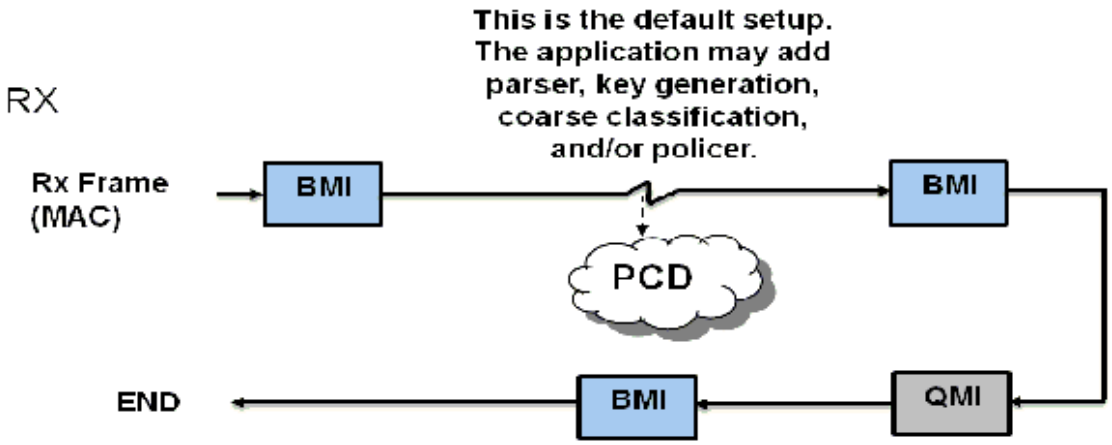
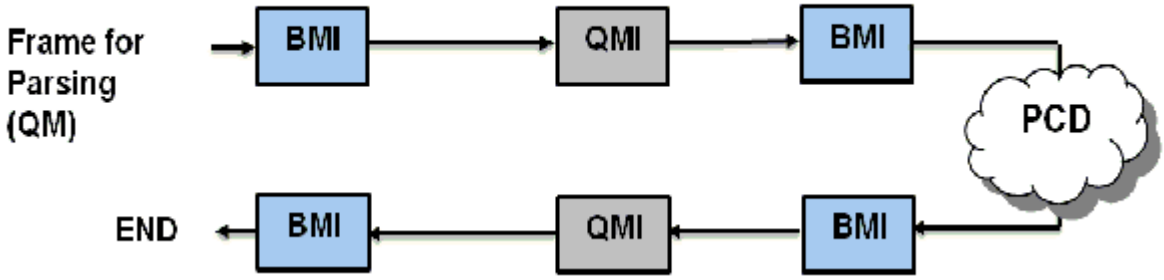
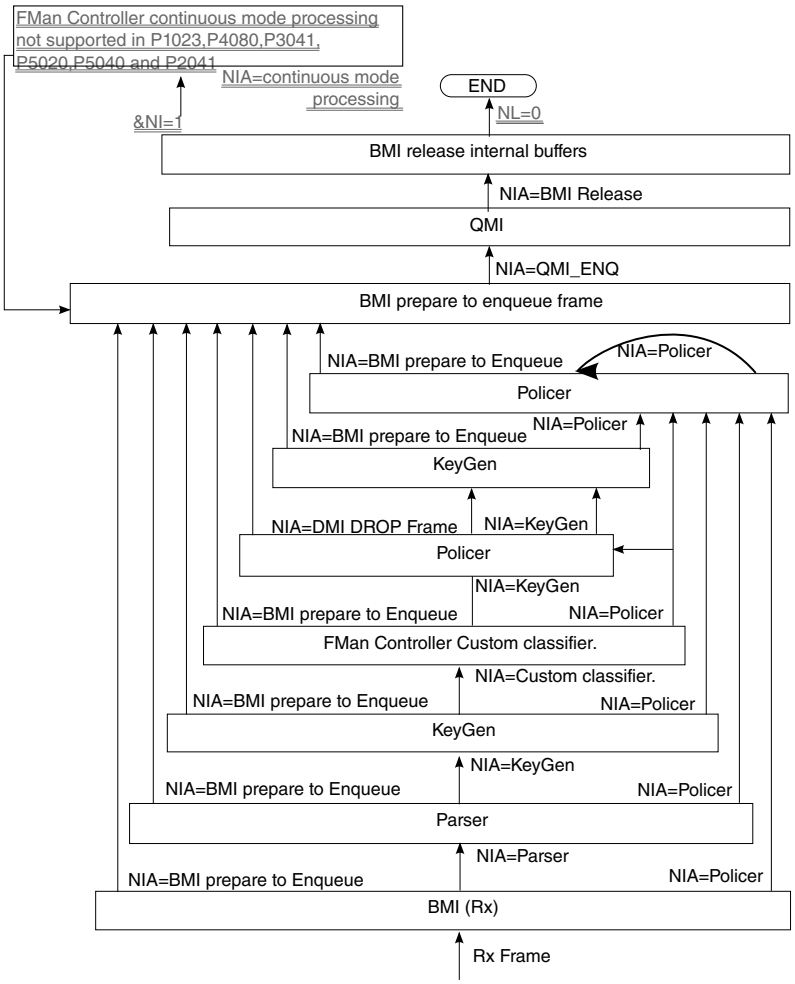
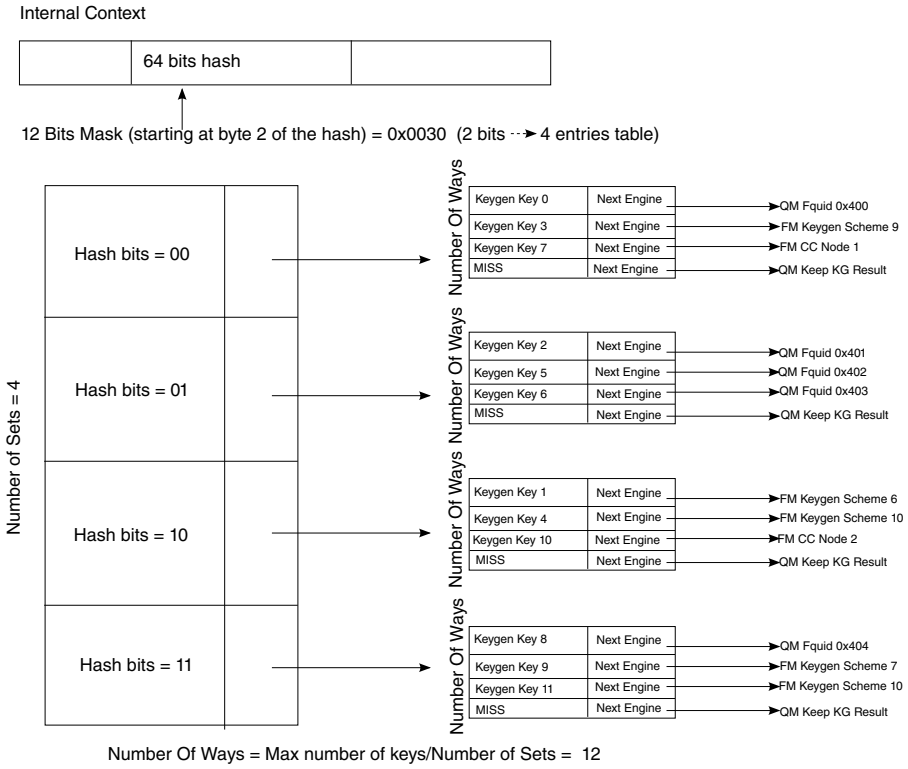
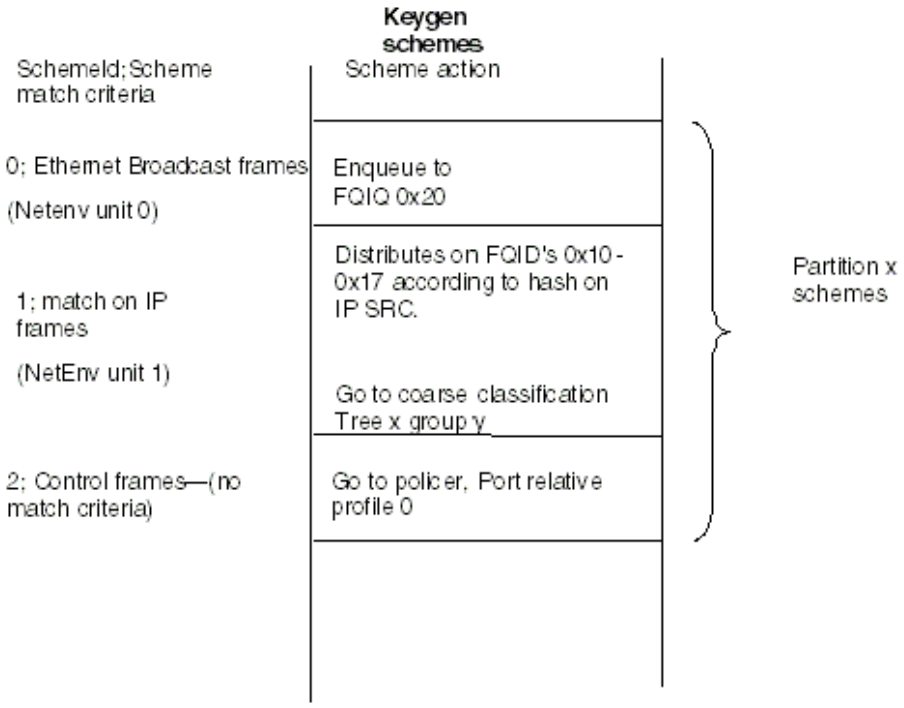
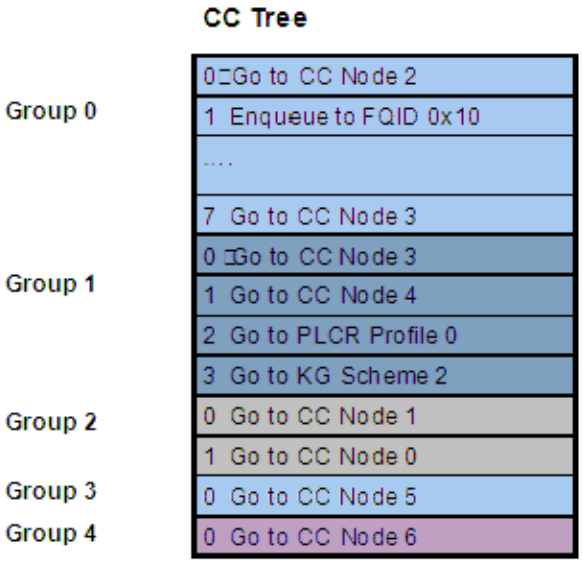
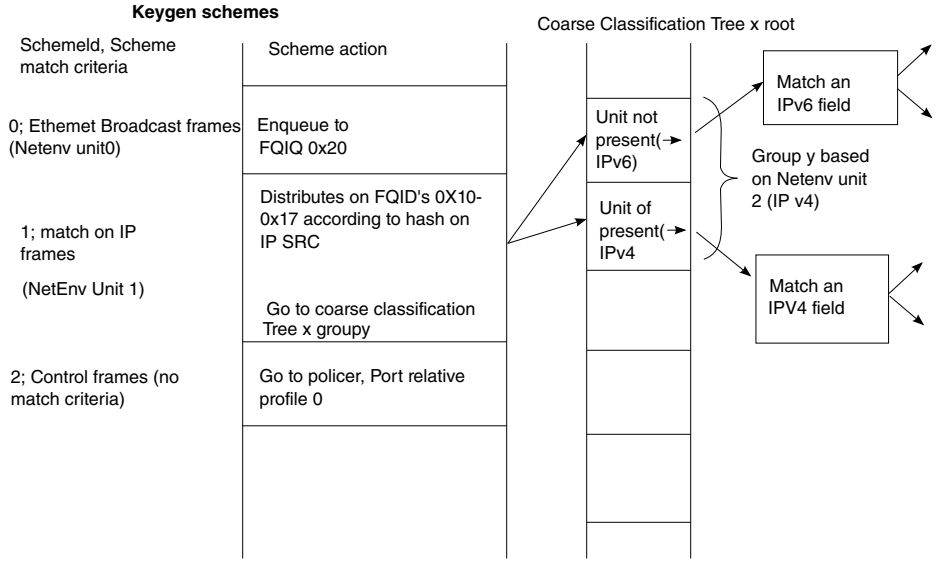
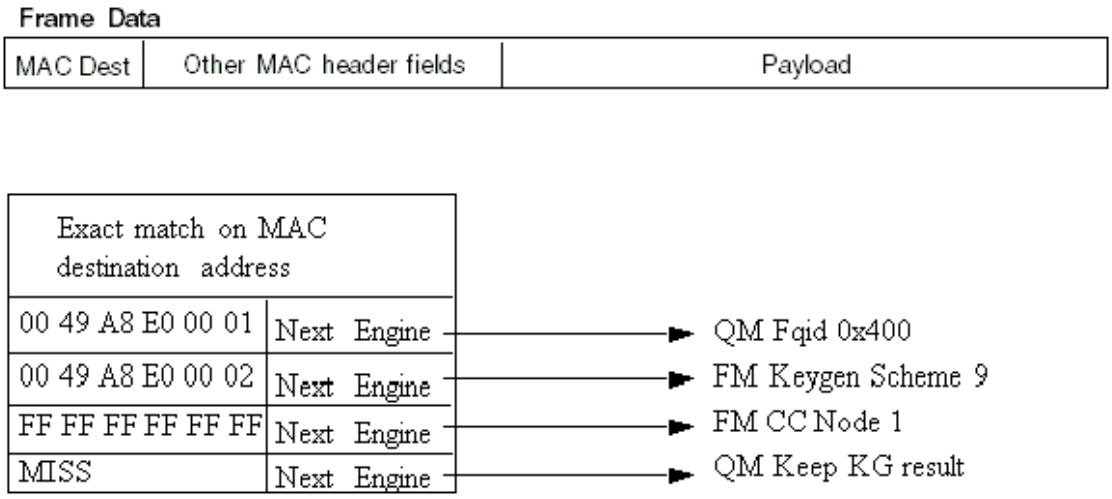
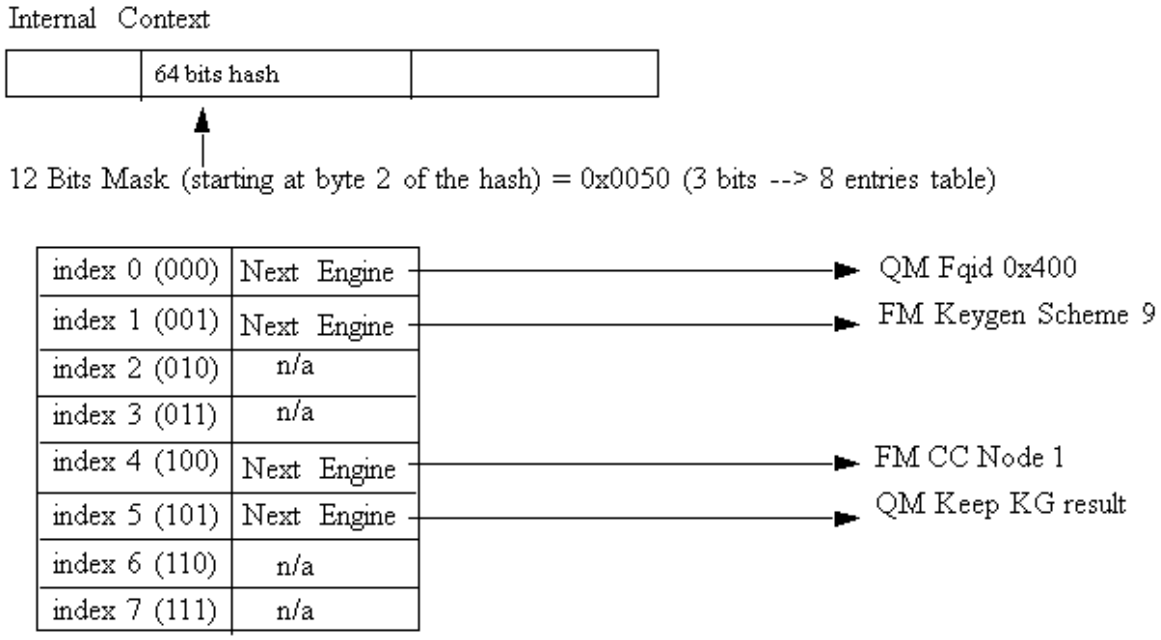
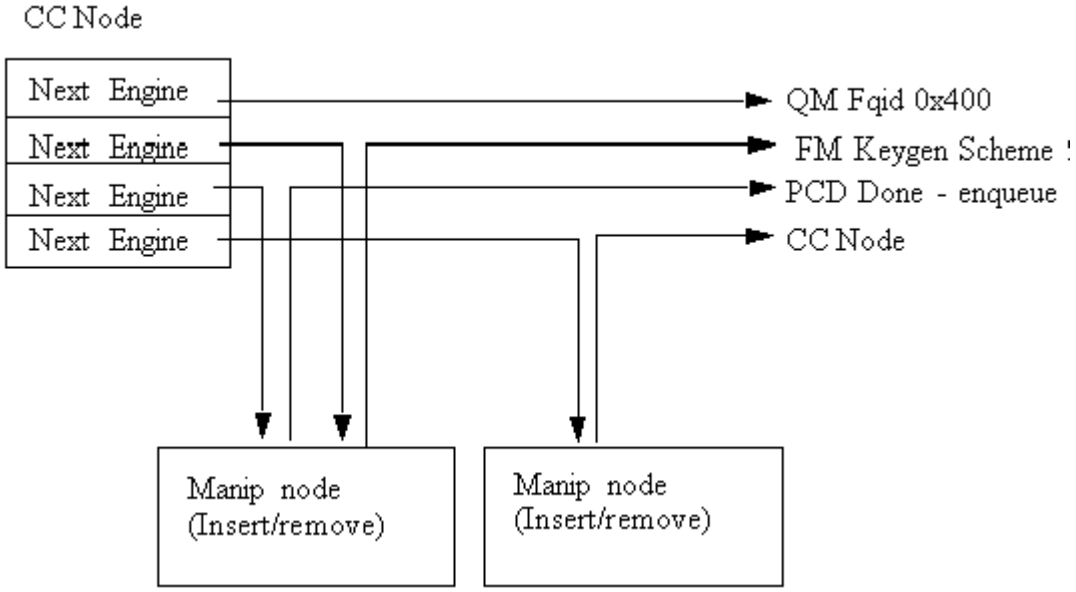
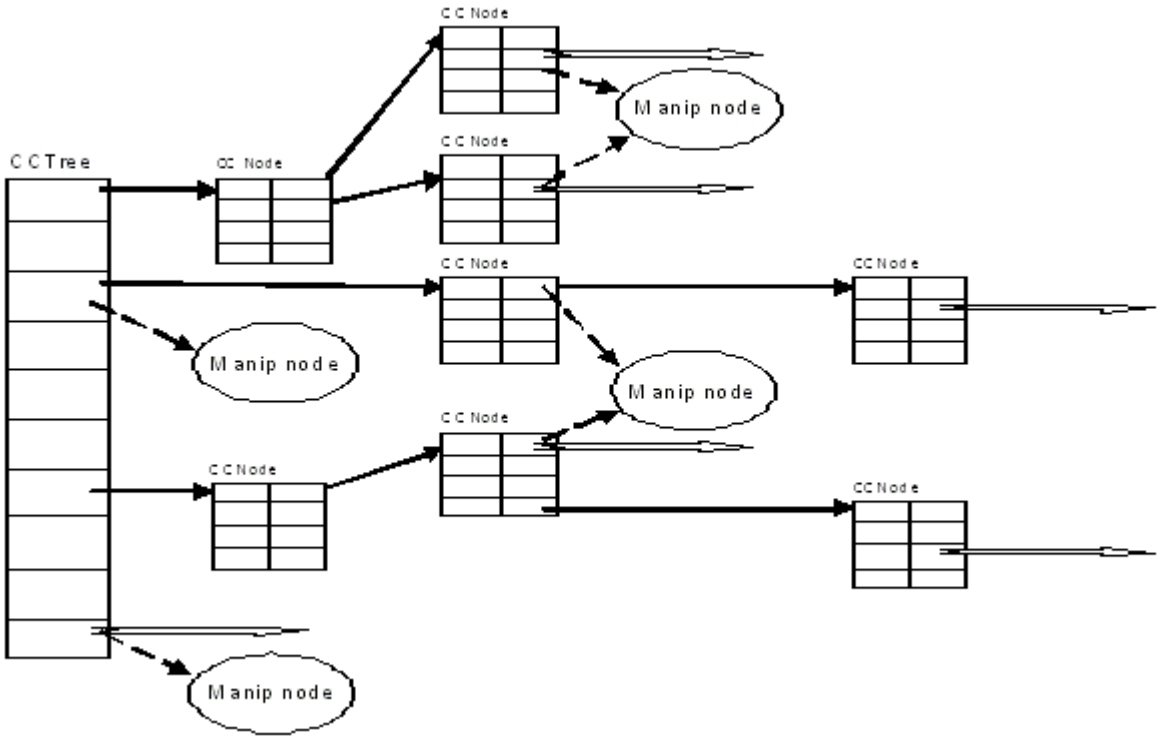
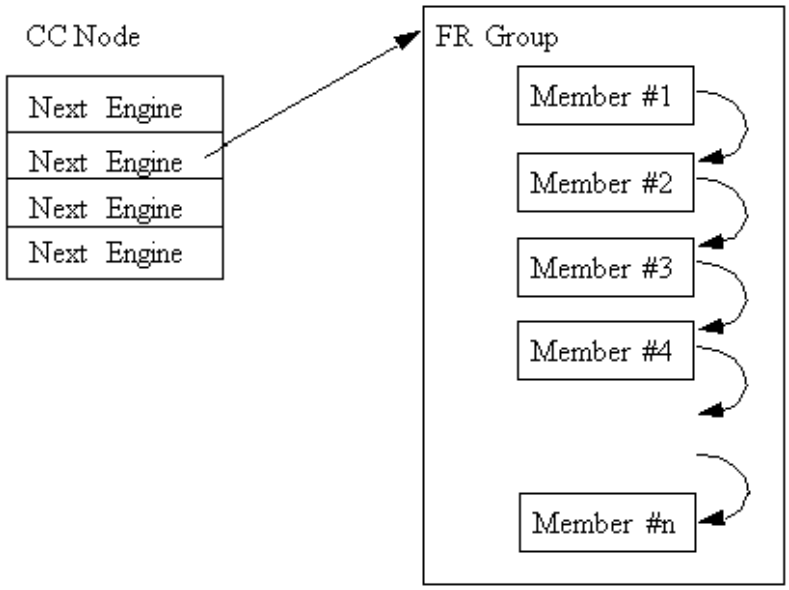
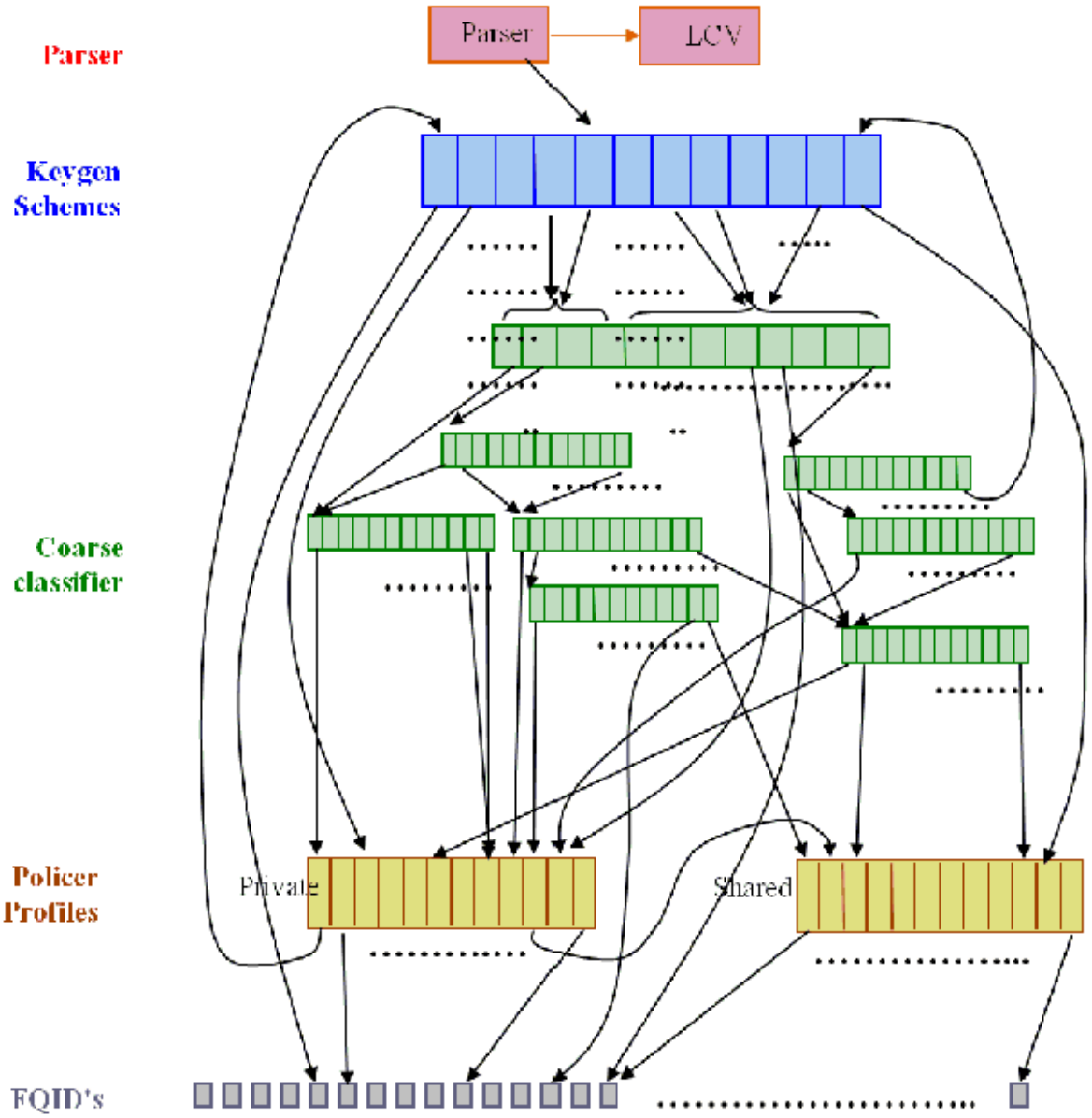
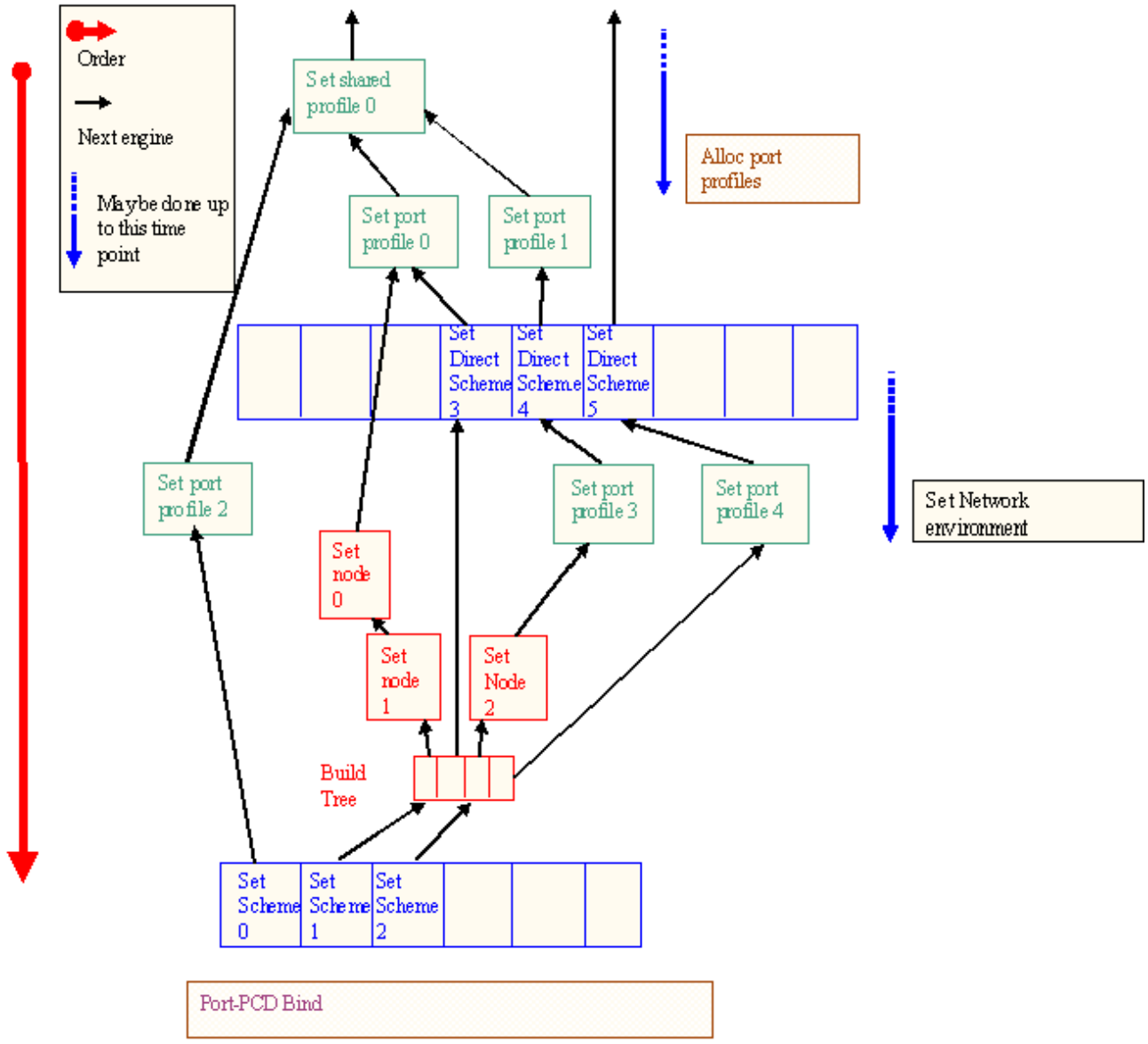
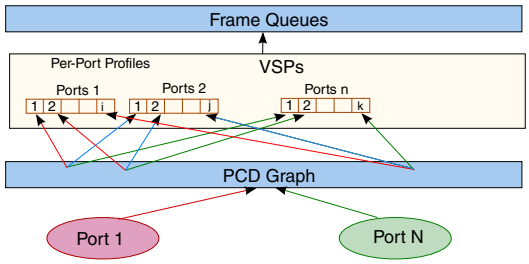
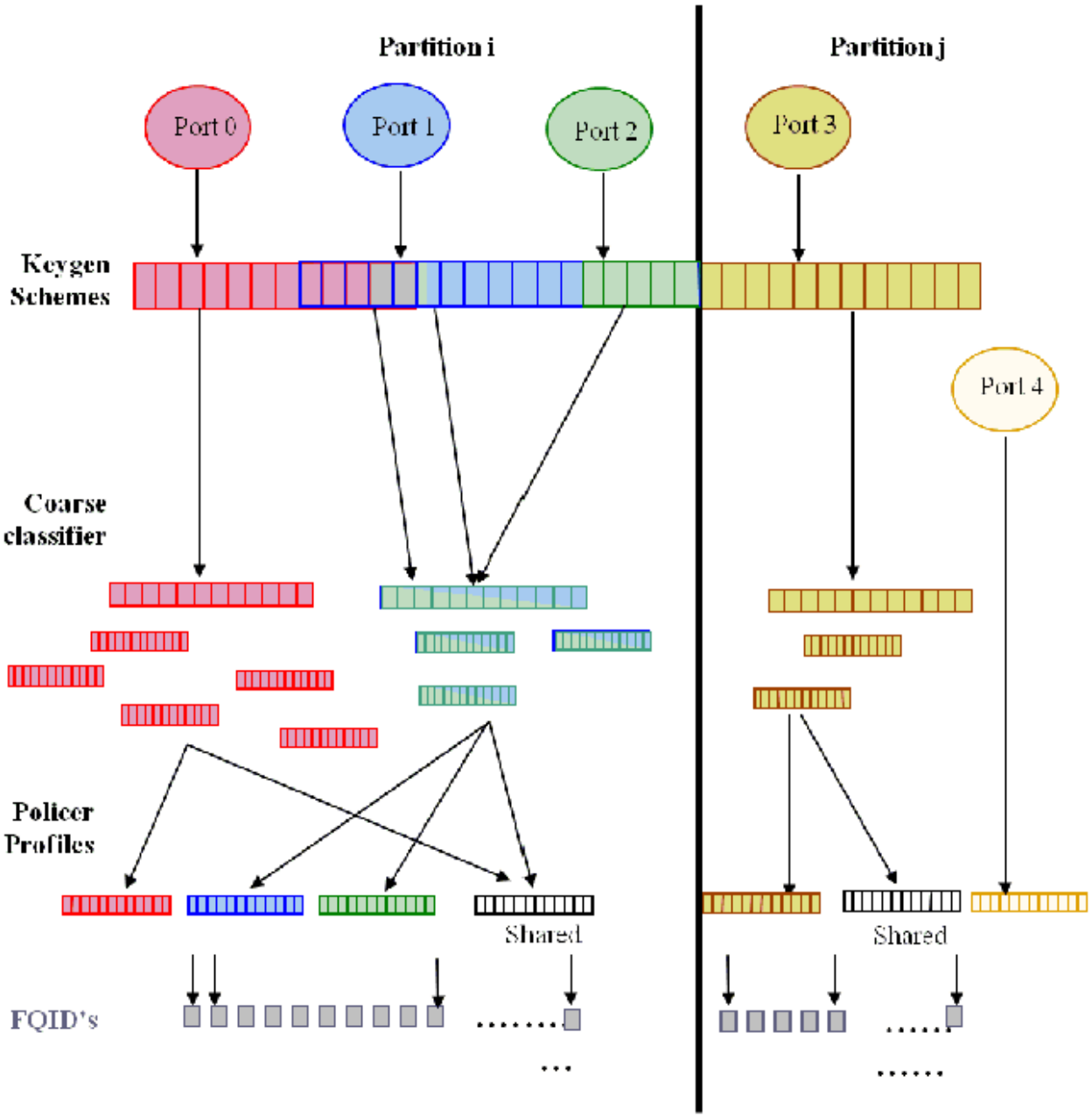
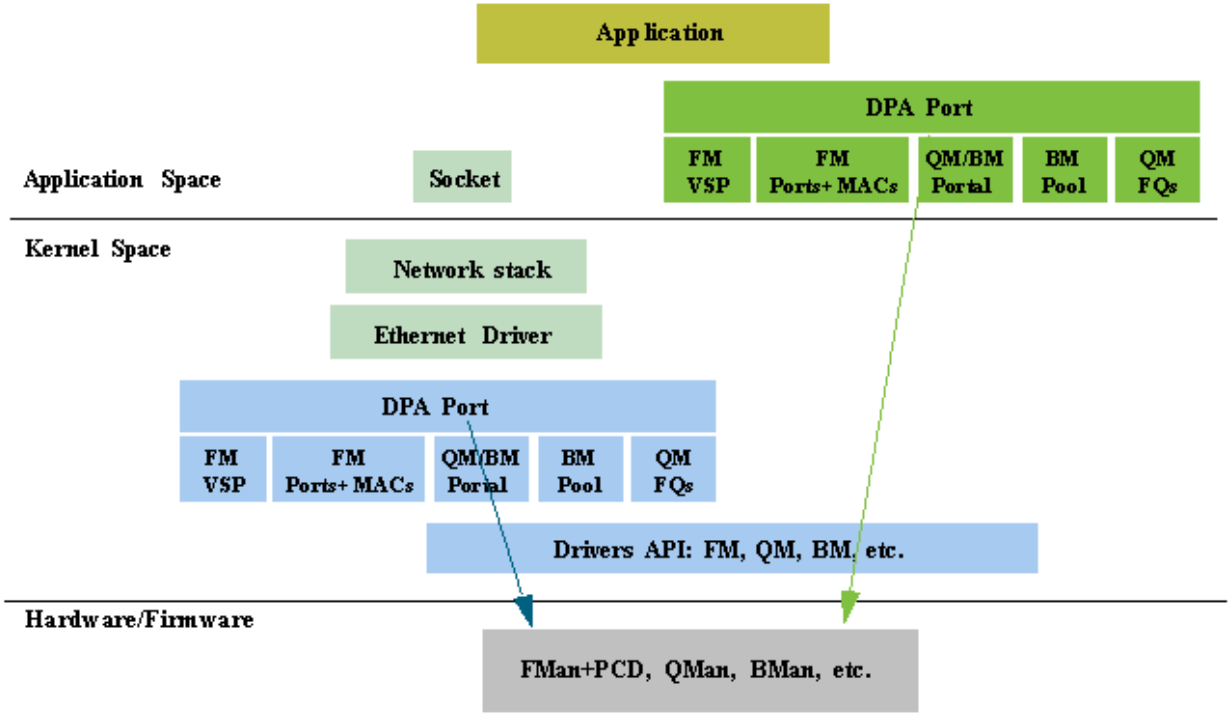
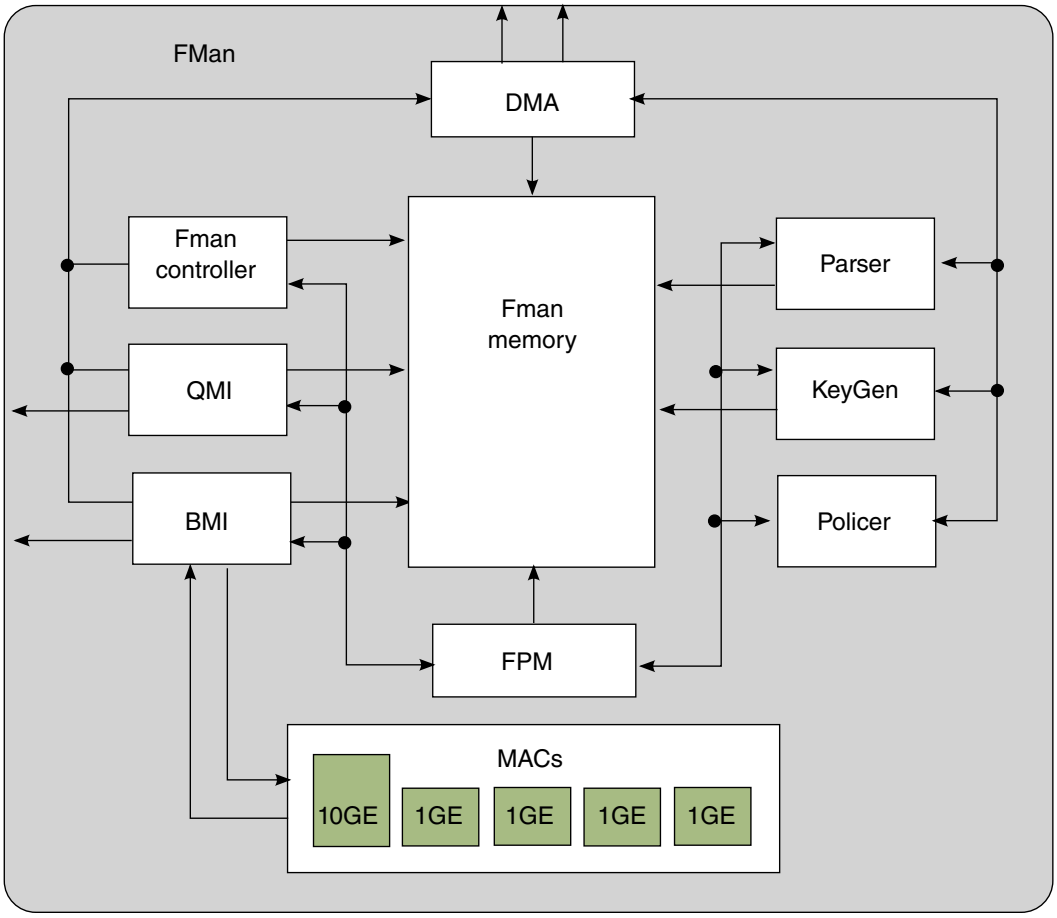
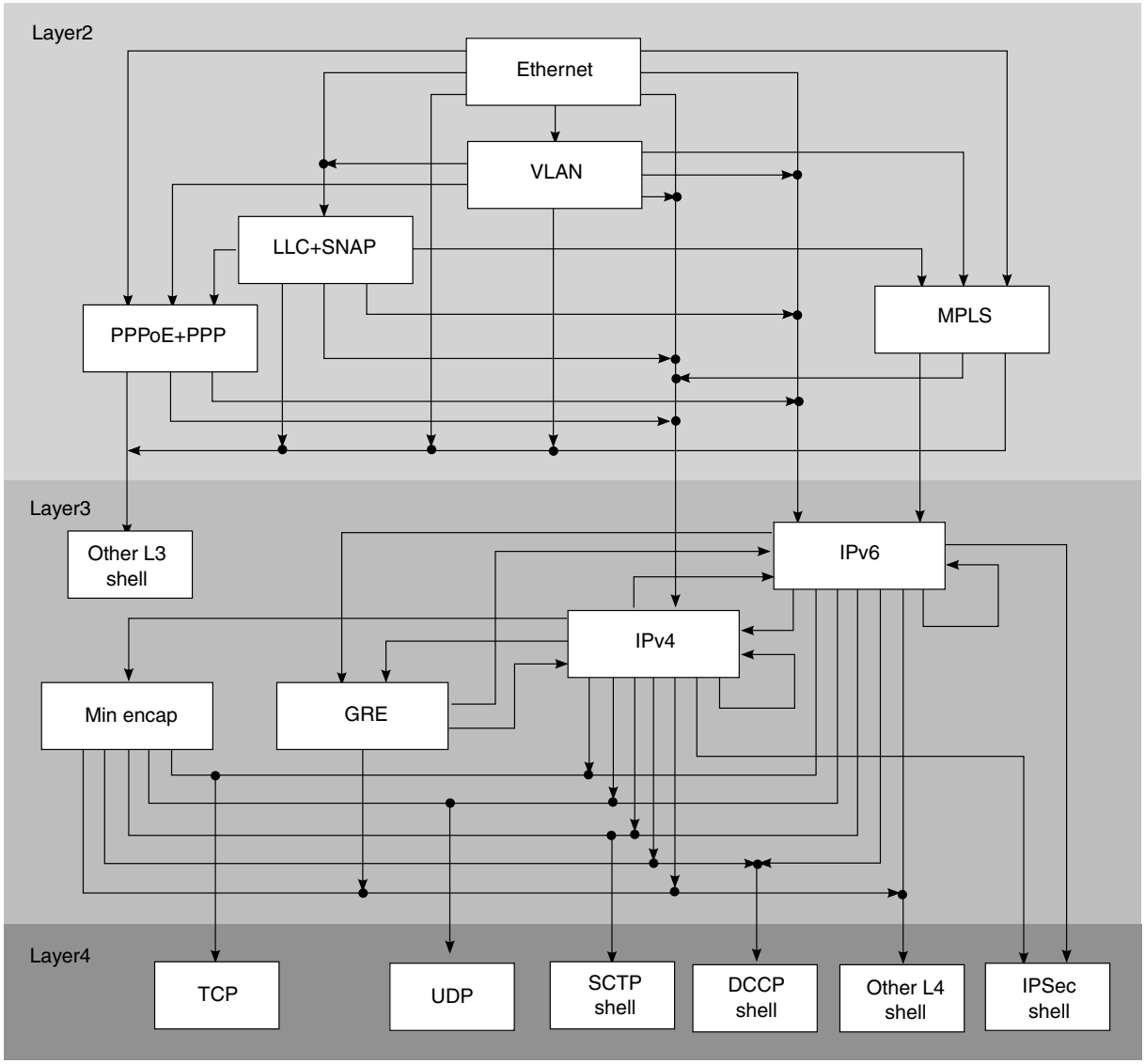
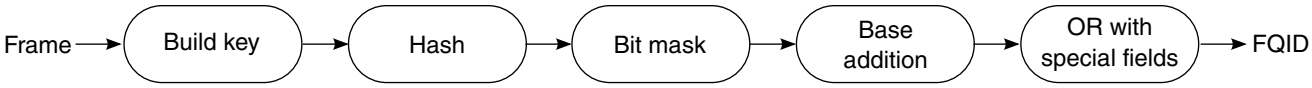
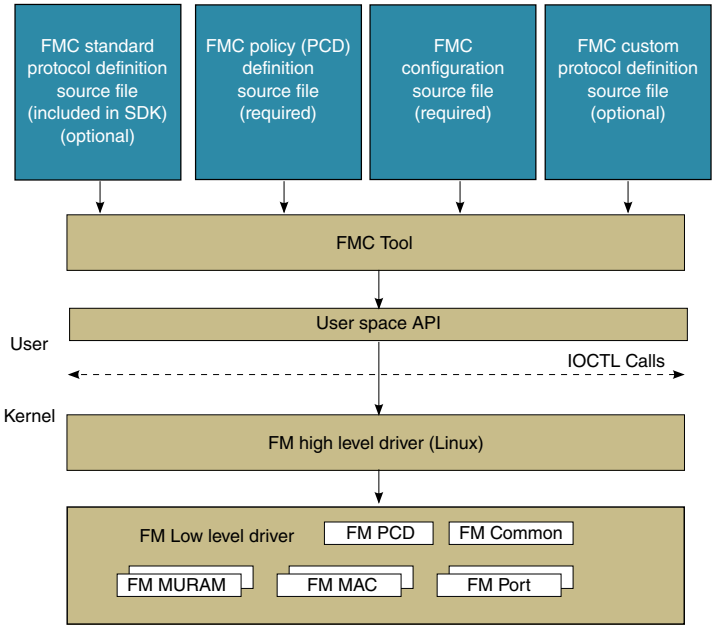
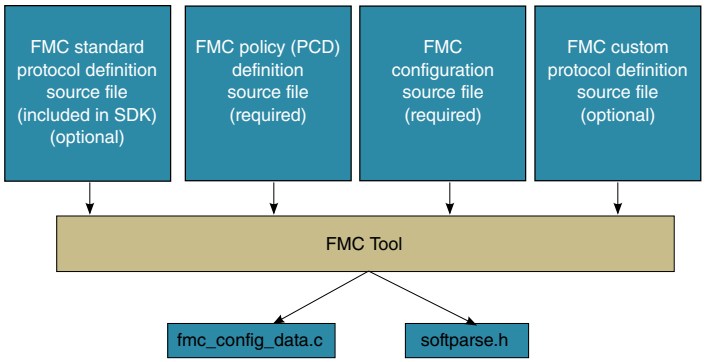
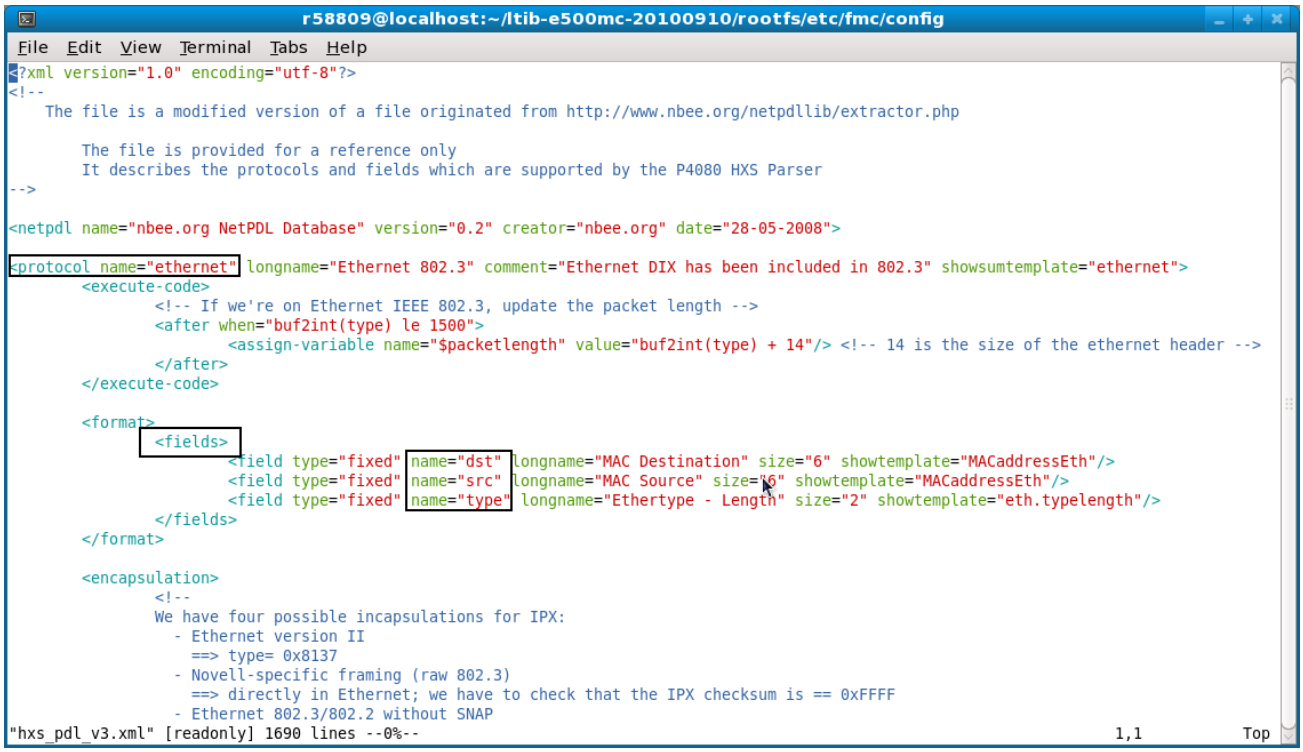
/**
 * qman_ceetm_ccg_release - Releases a previously claimed CCG.
 * ccg: the CCG to be released.
 *
 * Returns zero for success, or -EBUSY if the given CCG's dependent objects(class
 * queues that are associated with the CCG) have not been released.
 */
int qman_ceetm_ccg_release(struct qm_ceetm_ccg *ccg);
```

7.4.2.2.7.8.2 Configure CCG

```
/* This struct is used to specify attributes for a CCG. The 'we_mask' field
 * controls which CCG attributes are to be updated, and the remainder specify
 * the values for those attributes. A CCG counts either frames or the bytes
 * within those frames, but not both ('mode'). A CCG can optionally cause
 * enqueues to be rejected, due to tail-drop or WRED, or both (they are
 * independent options, 'td_en' and 'wr_en_g,wr_en_y,wr_en_r'). Tail-drop can be
 * level-triggered due to a single threshold ('td_thres') or edge-triggered due
 * to a "congestion state", but not both ('td_mode'). Congestion state has
 * distinct entry and exit thresholds ('cs_thres_in' and 'cs_thres_out'), and
 * notifications can be sent to software the CCG goes in to and out of this
 * congested state ('cscn_en').
 * The struct qm_cgr_cs_thres has already been defined in fsl_qman.h
 */
struct qm_ceetm_ccg_params {
        /* Boolean fields together in a single bitfield struct */
        struct {
                /* Whether to count bytes or frames. 1==frames */
                int mode:1;
                /* En/disable tail-drop. 1==enable */
                int td_en:1;
                /* Tail-drop on congestion-state or threshold. 1=threshold */
                int td_mode:1;
                /* Generate congestion state change notifications. 1==enable */
                int cscn_en:1;
```

```
                /* Enable WRED rejections (per colour). 1==enable */
                int wr_en_g:1;
                int wr_en_y:1;
                int wr_en_r:1;
        } __packed;
        /* Tail-drop threshold. See qm_cgr_thres_[gs]et64(). */
        struct qm_cgr_cs_thres td_thres;
        /* Congestion state thresholds, for entry and exit. */
        struct qm_cgr_cs_thres cs_thres_in;
        struct qm_cgr_cs_thres cs_thres_out;
        /* Overhead accounting length. Per-packet "tax", from -128 to +127 */
        signed char oal;
        /* WRED parameters. */
        struct qm_cgr_wr_parm wr_parm_g;
        struct qm_cgr_wr_parm wr_parm_y;
        struct qm_cgr_wr_parm wr_parm_r;
};

/* Bits used in 'we_mask' to qman_ceetm_ccg_set(), controls which attributes of
 * the CCGR are to be updated. */
#define QM_CCGR_WE_CDV     0x0000 /* cdv */
#define QM_CCGR_WE_MODE         0x0001 /* mode (bytes/frames) */
#define QM_CCGR_WE_TD_EN        0x0004 /* congestion state tail-drop enable */
#define QM_CCGR_WE_TD_MODE      0x4000 /* tail-drop mode (state/threshold) */
#define QM_CCGR_WE_TD_THRES     0x2000 /* tail-drop threshold */
#define QM_CCGR_WE_CS_THRES_IN  0x0002 /* congestion state entry threshold */
#define QM_CCGR_WE_CS_THRES_OUT 0x1000 /* congestion state exit threshold */
#define QM_CCGR_WE_CSCN_EN      0x0010 /* congestion notification enable */
#define QM_CCGR_WE_CSCN_TUPD    0x0008 /* CSCN target update */
#define QM_CCGR_WE_OAL          0x0800 /* overhead accounting length */
#define QM_CCGR_WE_WR_PARM_G    0x0400 /* WRED parameters - green */
#define QM_CCGR_WE_WR_PARM_Y    0x0200 /* WRED parameters - yellow */
#define QM_CCGR_WE_WR_PARM_R    0x0100 /* WRED parameters - red */
#define QM_CCGR_WE_WR_EN_G      0x0080 /* WRED enable - green */
#define QM_CCGR_WE_WR_EN_Y      0x0040 /* WRED enable - yellow */
#define QM_CCGR_WE_WR_EN_R      0x0020 /* WRED enable - red */

/**
 * qman_ceetm_ccg_set
 * qman_ceetm_ccg_get - Configure/query a subset of CCG attributes.
 * @ccg: the given CCG.
 * @we_mask: the write enable mask for the CCG attributes.
 * @params: the parameters set for this CCG.
 *
 * Returns zero for success, or -EINVAL if there is error for this CCG setting.
 */
int qman_ceetm_ccg_set(struct qm_ceetm_ccg *ccg,
                       u32 we_mask,
                       const struct qm_ceetm_ccg_params *params);
int qman_ceetm_ccg_get(struct qm_ceetm_ccg *ccg,
                       struct qm_ceetm_ccg_params *params);
```

7.4.2.2.7.8.3 Query CCG statistics

```
/**
 * qman_ceetm_ccg_get_reject_statistics - Get the statistics provided by
 * CEETM CCG counters.
 * @ccg: the given CCG object.
 * @flags: indicates whether the statistics counter will be cleared after query.
```

```
 * @frame_count: The number of the frames that have been counted since the
 * counter was cleared last time.
 * @byte_count: the number of bytes in all frames that have been counted.
 *
 * Return zero for success or -EINVAL if query statistics command returns error.
 *
 */
int qman_ceetm_ccg_get_reject_statistics(struct qm_ceetm_ccg *ccg, u32 flags,
                                         u64 *frame_count, u64 *byte_count);
```

7.4.2.2.7.8.4 Set/get Congestion State Change Notification Target

```
/** qman_ceetm_cscn_swp_get - Query whether a given software portal index is
  * in the cscn target mask.
  * @ccg: the give CCG object.
  * @swp_idx: the index of the software portal.
  * @cscn_enabled: 1: cscn is enabled in this swp. 0: cscn is not enabled
  * in this swp.
  *
  * Return 0 for success, or -EINVAL if command in set/get function fails.
  */
int qman_ceetm_cscn_swp_get(struct qm_ceetm_ccg *ccg,
                            u16 swp_idx,
                            unsigned int *cscn_enabled);

/** qman_ceetm_cscn_dcp_set - Add or remove a direct connect portal from the\
  * target mask.
  * qman_ceetm_cscn_swp_get - Query whether a given direct connect portal index
  * is in the cscn target mask.
  * @ccg: the give CCG object.
  * @dcp_idx: the index of the direct connect portal.
  * @cscn_enabled: 1: Set the dcp to be cscn target. 0: remove the dcp from
  * the target mask.
  *
  * Return 0 for success, or -EINVAL if command in set/get function fails.
  */
int qman_ceetm_cscn_dcp_set(struct qm_ceetm_ccg *ccg,
                            u16 dcp_idx,
                            unsigned int cscn_enabled);
int qman_ceetm_cscn_dcp_get(struct qm_ceetm_ccg *ccg,
                            u16 dcp_idx,
                            unsigned int *cscn_enabled);
```

7.4.2.2.8 Other QMan APIs

The following sections describe the interfaces provided by the QMan driver for manipulating QMan device.

7.4.2.2.8.1 Waterfall Power Management

Waterfall power management is a mechanism that works between the QMan and the e6500's Drowsy Core mode. The basic idea is that when using multiple cores and pool channels to forward frames, and the QMan receives less traffic than required to keep all cores busy, it no longer assigns to the higher numbered cores (i.e. software portals) in a pool. This feature is supported from QMan3.0, and the APIs below can only be called at SoC with QMan3.0

```
/**
 * qman_set_wpm - Set waterfall power management
```

```
 *
 * @wpm_enable: boolean, 1 = enable wpm, 0 = disable wpm.
 *
 * Return 0 for success, return -ENODEV if QMan misc_cfg register is not
 * accessible.
 */
int qman_set_wpm(int wpm_enable);
/**
 * qman_get_swp - Query the waterfall power management setting
 *
 * @wpm_enable: boolean, 1 = enable wpm, 0 = disable wpm.
 *
 * Return 0 for success, return -ENODEV if QMan misc_cfg register is not
 * accessible.
 */
int qman_get_wpm(int *wpm_enable);
```

7.4.2.2.9 USDPAA-specific APIs

7.4.2.2.9.1 Overview

The USDPAA SDK includes a port of the QMan and BMan drivers to linux user space, and assumes a pthreads-based programming model, and the use of a single application/process instance.

Unlike conventional user space interfaces to hardware (in which the kernel manipulates hardware interfaces on behalf of user space processes), the USDPAA QMan and BMan drivers operate on the hardware directly from user space. The underlying mechanisms for this are based on the USDPAA Linux character device. Put briefly, the kernel exposes the devices to user space as character devices with various attributes, which allow the user space drivers to mmap() the Corenet portal regions directly. Interrupt handling works by disabling the portal interrupt when it asserts and conveying the interrupt event to user-space via the USDPAA device's file-descriptor (the user-space portal driver can re-enable the interrupt whenever it so chooses).

The key distinction between the QMan (or BMan) drivers found in the Linux kernel and the one in USDPAA is that the latter does not perform a global initialization step to parse all available portals and initialize and assign them to CPUs/threads. The model in USDPAA assumes that the application is responsible for creating its own pthreads, and is responsible for making those threads affine to the appropriate CPUs (if indeed it chooses to). Such applications simply call into a USDPAA-specific API to "enable" the thread for QMan/BMan portal usage (specifying the desired CPU-affinity for the portal), and the USDPAA driver at that point will "find" an unused portal suitable for the requested CPU and initialize and bind it to the pthread via thread-local storage, or fail if no such portal was available. Portals can likewise be released from threads when they are no longer required, without requiring the pthread itself to be destroyed.

Not only do both the Linux kernel and USDPAA drivers initialize all portals at start-up, but their portals are managed as CPU-*local* associations. USDPAA on the other hand manages portals as *thread-local* associations. Nonetheless, portals are (typically) configured for a particular CPU-affinity, meaning that ring and data stashing will target the designated CPU, so USDPAA applications are advised to run their threads on the CPUs for which their portals are configured - failing to do so will not break the system nor the application, but will yield significant performance degradation relative to an optimized system.

NOTE

The "recovery_mode" parameters in the following APIs are currently unsupported, they are reserved until a future release, so should always be set to 0 (or "FALSE").

7.4.2.2.9.2 Thread initialization

An application pthread can request that the QMan drivers initialize and bind a portal for use by that thread by calling the following APIs. This API must be called and return success prior to calling any of the interfaces mentioned in QMan portal

APIs on page 641 or BMan CoreNet portal APIs on page 627. Note, this interface is dependent on the device-tree layer having been initialized, see Device-tree dependency on page 676.

```
int qman_thread_init(void);
int qman_thread_init_idx(uint32_t idx);
int bman_thread_init(void);
int bman_thread_init_idx(uint32_t idx);
```

Note that the 'idx' parameter influences the driver's search for the portal with the specified portal index. The pthread is required to already be affine to the given CPU.

As usual with "int"-valued APIs, a return value of zero indicates success, otherwise a standard negative error number is returned in event of failure.

Likewise, the thread-portal association can be ended (and thus make the underlying portal available for another user/thread) by calling the following APIs.

```
int qman_thread_finish(void);
int bman_thread_finish(void);
```

7.4.2.2.9.3 FQID/BPID allocation

As of the current release of USDPAA, support for allocation of FQIDs and BPIDs is via hard-coded ranges encoded within the drivers. This will be revised in a future release.

To enable the allocation mechanisms in the USDPAA QMan (and BMan) drivers, the following APIs must be called once from a thread that has already successfully bound to a QMan (or BMan) portal via qman_thread_init() (or bman_thread_init(), respectively).

```
int qman_global_init(int recovery_mode);
int bman_global_init(int recovery_mode);
```

Note that this API does not need to be called from all threads, only from one, but that it should be called before using any QMan/BMan APIs that require dynamic allocation of FQIDs or BPIDs.

7.4.2.2.9.4 Interrupt handling

7.4.2.2.9.4.1 USDPAA file-descriptors

Interrupt handling in USDPAA is done by reading a standard file descriptor that is created when a portal is assigned to a thread. The application should use the poll() or select() API to determine when the file descriptor becomes readable which indicates an interrupt has occurred. The following APIs return the current USDPAA file-descriptors for portals bound to the calling thread.

```
int qman_thread_fd(void);
int bman_thread_fd(void);
```

7.4.2.2.9.4.2 Processing interrupt-driven portal duties

It should be noted that the QMan and BMan drivers allow applications to dynamically configure which portal "duties" are to be triggered and performed by interrupts versus polling. See Modifying interrupt-driven portal duties (BMan) on page 627 and Modifying interrupt-driven portal duties (QMan) on page 642 for details. Prior to going into a blocking read(), select(), poll(), [...] on the file-descriptor, the application will need to ensure that the "duties" it wishes to wait for are put into IRQ mode (ie. added to the portal's "irqsource") prior to blocking, otherwise the presence of such work will not trigger any interrupt and so will not wake up the sleeping process. Likewise, if going back into polling mode and expecting polling APIs to process such portal duties, the application will need to remove such duties from the "irqsource".

For portal duties that are in the "irqsource", and after any USDPAA-supported file-descriptor operation that indicates that an interrupt was received, the application pthread can call the following APIs to post-process any such interrupt-driven duties:

```
/* Post-process interrupts. NB, the kernel IRQ handler disables the interrupt
 * line before notifying us, and this post-processing re-enables it once
 * processing is complete. As such, it is essential to call this before going
 * into another blocking read/select/poll. */
void qman_thread_irq(void);
void bman_thread_irq(void);
```

7.4.2.2.9.5 Device-tree dependency

The QMan/BMan drivers “find” portals by referring to information extracted from the device-tree, as represented in the procfs filesystem located at /proc/device-tree. This information is handled by a USDPAA “of” driver, which must be initialised before any attempts are made to initialise threads for QMan/BMan use as described in Thread initialization on page 674. As of the current USDPAA release, the “of” driver must be explicitly initialised by applications prior to QMan/BMan thread initialisation, though this may change in future releases. (As a side note, other USDPAA mechanisms, such as application-side configuration parsing may also be dependent on this device-tree layer, and so the “of” init should occur prior to all such dependencies.)

The API to initialise the “of” driver is of_init(), which parses the /proc/device-tree representation into an internal representation of C data-structures.

```
#ifndef OF_INIT_DEFAULT_PATH
#define OF_INIT_DEFAULT_PATH "/proc/device-tree"
#endif
int of_init_path(const char *dt_path);
/* Use of this wrapper is recommended. */
static inline int of_init(void)
{
        return of_init_path(OF_INIT_DEFAULT_PATH);
}
```

Conversely, the of_finish() API cleans up all data structures and handles created by of_init(), which may be useful to satisfy leak-checking tools, or persistent process/thread recycling schemes.

```
void of_finish(void);
```

7.4.2.2.10 Sysfs and debugfs QMan/BMan interfaces

This chapter describes the qman and bman interfaces available via sysfs and debugfs.

NOTE

For non-P4080 devices, it is generally recommended to determine these from the device-tree, the SoC-specific Reference Manual, and/or by examining the sysfs filesystem at run-time.

7.4.2.2.10.1 QMan sysfs

7.4.2.2.10.1.1 /sys/devices/ffe000000.soc/ffe318000.qman

Description:

This directory contains a snapshot of the internal state of the qman device.

7.4.2.2.10.1.2 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture

Description:

This directory contains a snapshot of error related qman attributes.

7.4.2.2.10.1.3 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture/sbec_<0..6>

Description:

Provides a count of the number of single bit ECC errors that have occurred when reading from one of the QMan internal memories. The range <0..6> represent a QMAN internal memory region defined as follows:

0: FQD cache memory

1: FQD cache tag memory

2: SFDR memory

3: WQ context memory

4: Congestion Group Record memory

5: Internal Order Restoration List memory

6: Software Portal ring memory

This file is read-reset.

7.4.2.2.10.1.4 /sys/devices/ffe000000.soc/ffe318000.qman/sfdr_in_use

Description:

Reports the number of SFDR currently in use. The minimum value is 1. This file is not available on Rev 1.0 of P4080 QorIQ.

This file is read-only

/sys/devices/ffe000000.soc/ffe318000.qman/pfdr_fpc

Description:

Total Packed Frame Descriptor Record Free Pool Count in external memory.

This file is read-only

7.4.2.2.10.1.5 /sys/devices/ffe000000.soc/ffe318000.qman/pfdr_cfg

Description:

Used to read the configuration of the dynamic allocation policy for PFDRs. The value is used to account for PFDR that may be required to complete any currently executing operations in the sequencers.

This file is read-only.

7.4.2.2.10.1.6 /sys/devices/ffe000000.soc/ffe318000.qman/idle_stat

Description:

This file can be used to determine when QMan is both idle and empty. The possible values are:

0: All work queues in QMan are NOT empty and QMan is NOT idle.

1: All work queues in QMan are NOT empty and QMan is idle.

2: All work queues in QMan are empty

3: All work queues in QMan are empty and QMan is idle.

This file is read-only.

7.4.2.2.10.1.7 /sys/devices/ffe000000.soc/ffe318000.qman/err_isr

Description:

QMan contains one dedicated interrupt line for signaling error conditions to software. This file identifies the source of the error interrupt within QMan. The value is displayed in hexadecimal format. Refer to the appropriate QorIQ SOC Reference Manual for a description of the QMAN_ERR_ISR register.

This file is read-only.

7.4.2.2.10.1.8 /sys/devices/ffe000000.soc/ffe318000.qman/dcp<0..3>_dlm_avg

Description:

These files contain an EWMA (exponentially weighted moving average) of dequeue latency samples for dequeue commands received on the sub portal. The range <0..3> refers to each of the direct-connect portals. The display format is as follows: <avg_interger>.<avg_fraction>

This file can be seeded with a interger value. The input interger is processed in the following manner: <avg_fraction> = lowest 8 bits / 256 , <avg_interger> = next 12 bits

ex: echo 0x201 > dcp0_dlm_avg

cat dcp0_dlm_avg

0.00390625

This file is read-write

7.4.2.2.10.1.9 /sys/devices/ffe000000.soc/ffe318000.qman/ci_rlm_avg

Description:

This file contains an EWMA (exponentially weighted moving average) of read latency samples for reads on CoreNet initiated by QMan. The display format is as follows: <avg_interger>.<avg_fraction>

This file can be seeded with a interger value. The input interger is processed in the following manner: <avg_fraction> = lowest 8 bits / 256 , <avg_interger> = next 12 bits

ex: echo 0x201 > ci_rlm_avg

cat ci_rlm_avg

0.00390625

This file is read-write

7.4.2.2.10.2 BMan sysfs

7.4.2.2.10.2.1 /sys/devices/ffe000000.soc/ffe31a000.bman

Description:

This directory contains a snapshot of the internal state of the BMan device.

7.4.2.2.10.2.2 /sys/devices/ffe000000.soc/ffe31a000.bman/error_capture

Description:

This directory contains a snapshot of error related BMan attributes.

7.4.2.2.10.2.3 /sys/devices/ffe000000.soc/ffe31a000.bman/error_capture/sbec_<0..1>

Description:

Provides a count of the number of single bit ECC errors that have occurred when reading from one of the BMan internal memories. The range <0..1> represent a BMAN internal memory region defined as follows:

0: Stockpile memory 0

1: Software Portal ring memory

This file is read-reset.

7.4.2.2.10.2.4 /sys/devices/ffe000000.soc/ffe31a000.bman/pool_count

Description:

This directory contains a snapshot of the number of free buffers available in any of the buffer pools.

7.4.2.2.10.2.5 /sys/devices/ffe000000.soc/ffe31a000.bman/fbpr_fpc

Description:

This file returns a snapshot of the Free Buffer Proxy Record free pool size. Total Free Buffer Proxy Record Free Pool Count in external memory.

This file is read-only

7.4.2.2.10.2.6 /sys/devices/ffe000000.soc/ffe31a000.bman/err_isr

Description:

BMan contains one dedicated interrupt line for signaling error conditions to software. This file identifies the source of the error interrupt within BMan. The value is displayed in hexadecimal format. Refer to the appropriate QorIQ SOC Reference Manual for a description of the BMAN_ERR_ISR register.

This file is read-only.

7.4.2.2.10.3 QMan debugfs

7.4.2.2.10.3.1 /sys/kernel/debug/qman

Description:

This directory contains various QMan device debugging attributes.

7.4.2.2.10.3.2 /sys/kernel/debug/qman/query_cgr

Description:

Query the entire contents of a Congestion Group Record. The file takes as input the Congestion Group Record ID. The output of the file returns the various CGR fields.

For example, if we want to query cgr_id 10 we would do the following:

echo 10 > query_cgr

cat query_cgr

Query CGR id 0xa

wr_parm_g MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_parm_y MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_parm_r MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_en_g: 0, wr_en_y: 0, we_en_r: 0

cscn_en: 0

cscn_targ: 0

cstd_en: 0

cs: 0

cs_thresh_TA: 0, cs_thresh_Tn: 0

i_bcnt: 0

a_bcnt: 0

7.4.2.2.10.3.3 /sys/kernel/debug/qman/query_congestion

Description:

Query the state of all 256 Congestion Groups in QMan. This is a read-only file. The output of the file returns the state of all congestion group records. The state of a congestion group is either "in congestion" or "not in congestion". Since CGR are normally not in congestion, only CGR which are in congestion are returned. If no CGR are in congestion, then this is indicated.

For example, if we want to perform a query we would do the following:

cat query_congestion

Query Congestion Result

All congestion groups not congested.

7.4.2.2.10.3.4 /sys/kernel/debug/qman/query_fq_fields

Description:

Query the frame queue programmable fields. This file takes as input the frame queue id to be queried on a subsequent read. The output of this file returns all the frame queue programmable fields. The default frame queue id is 1.

Refer to the appropriate QorIQ SOC Reference Manual for detailed explanation on the return values.

For example, if we determine that our application is using frame queue 482 we could use this file in the following manner:

echo 482 > query_fq_fields

cat query_fq_fields

Query FQ Programmable Fields Result fqid 0x1e2

orprws: 0

oa: 0

olws: 0

cgid: 0

fq_ctrl:

Aggressively cache FQ

Don't block active

Context-A stashing

Tail-Drop Enable

dest_channel: 33

dest_wq: 7

ics_cred: 0

td_mant: 128

td_exp: 7

ctx_b: 0x19e

ctx_a: 0x78b59e18

ctx_a_stash_exclusive:

FQ Ctx Stash

Frame Annotation Stash

ctx_a_stash_annotation_cl: 1

ctx_a_stash_data_cl: 2

ctx_a_stash_context_cl: 2

7.4.2.2.10.3.5 /sys/kernel/debug/qman/query_fq_np_fields

Description:

Query the frame queue non programmable fields. This file takes as input the frame queue id to be queried on a subsequent read. The output of this file returns all the frame queue non programmable fields. The default frame queue id is 1.

Refer to the appropriate QorIQ SOC Reference Manual for detailed explanation on the return values.

For example, if we determine that our application is using frame queue 482 we could use this file in the following manner:

echo 482 > query_fq_np_fields

cat query_fq_np_fields

Query FQ Non Programmable Fields Result fqid 0x1e2

force eligible pending: no

retirement pending: no

state: Out of Service

fq_link: 0x0

odp_seq: 0

orp_nesn: 0

orp_ea_hseq: 0

orp_ea_tseq: 0

orp_ea_hptr: 0x0

orp_ea_tptr: 0x0

pfdr_hptr: 0x0

pfdr_tptr: 0x0

is: ics_surp contains a surplus

ics_surp: 0

byte_cnt: 0

frm_cnt: 0

ra1_sfdr: 0x0

ra2_sfdr: 0x0

od1_sfdr: 0x0

od2_sfdr: 0x0

od3_sfdr: 0x0

7.4.2.2.10.3.6 /sys/kernel/debug/qman/query_cq_fields

Description:

Query all the fileds of in a particular CQD. This file takes input as the DCP id plus the class queue id to be queried on a subsequent read. The output of this file returns all the class queue fields. The default class queue id is 1 of DCP 0

Refer to the appropriate QorIQ SOC Reference Manual for detailed explanation on the return values.

For example, if we determine that our application is using class queue 4 of DCP 1, we could use this file in the following manner:

echo 0x01000004 > query_cq_fields

(The most left 8 bits are used to specify DCP id, and the rest of 24 bits are used to specify the class queue id)

cat query_fq_fields

Query CQ Fields Result cqid 0x4 on DCP 1

ccgid: 4

state: 0

pfdr_hptr: 0

pfdr_tptr: 0

od1_xsfdr: 0

od2_xsfdr: 0

od3_xsfdr: 0

od4_xsfdr: 0

od5_xsfdr: 0

od6_xsfdr: 0

ra1_xsfdr: 0

ra2_xsfdr: 0

frame_count: 0

7.4.2.2.10.3.7 /sys/kernel/debug/qman/query_ceetm_ccgr

Description:

Query the configuration and state fields within a CEETM Congestion Group Record that relate to congestion management(CM). This file takes input as the DCP id(most left 8 bits) and CEETM Congestion Group Record ID(most right 24 bits). The output of the file returns the various CCGR fields.

For example, if we want to query ccgr_id 7 of DCP 0, we would do the following:

echo 0x00000007 > query_ceetm_ccgr

cat query_ceetm_ccgr

Query CCGID 7

Query CCGR id 7 in DCP 0

wr_parm_g MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_parm_y MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_parm_r MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_en_g: 0,

wr_en_y: 0,

we_en_r: 0

cscn_en: 0

cscn_targ_dcp:

cscn_targ_swp:

td_en: 0

cs_thresh_in_TA: 0,

cs_thresh_in_Tn: 0

cs_thresh_out_TA: 0,

cs_thresh_out_Tn: 0

td_thresh_TA: 0,

td_thresh_Tn: 0

mode: byte count

i_cnt: 0

a_cnt: 0

7.4.2.2.10.3.8 /sys/kernel/debug/qman/query_wq_lengths

Description:

Query the length of the Work Queues in a particular channel. This file takes as input a specified channel id. The output of this file returns the lengths of the work queues on the specified channel.

For example, if we want to query channel 1 we would do the following:

echo 1 > query_wq_lengths

cat query_wq_lengths

Query Result For Channel: 0x1

wq0_len : 0

wq1_len : 0

wq2_len : 0

wq3_len : 0

wq4_len : 0

wq5_len : 0

wq6_len : 0

wq7_len : 0

7.4.2.2.10.3.9 /sys/kernel/debug/qman/fqd/avoid_blocking_[enable | disable]

Description:

Query Avoid_Blocking bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Avoid_Blocking bit mask enabled or disabled.

For example, if we want to find all frame queues with Avoid_Blocking enabled, we would do the following:

```
# cat avoid_blocking_enable
List of fq ids with: Avoid Blocking :enabled
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001de,0x0001df,0x0001e0,0x0001e1,
0x0001ea,0x0001eb,0x0001ec,0x0001ed,0x0001f6,0x0001f7,0x0001f8,0x0001f9,
...
Total FQD with: Avoid Blocking : enabled = 528
Total FQD with: Avoid Blocking : disabled = 32239
```

7.4.2.2.10.3.10 /sys/kernel/debug/qman/fqd/prefer_in_cache_[enable | disable]

Description:

Query Prefer_in_Cache bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Prefer_in_Cache bit mask enabled or disabled.

For example, if we want to find all frame queues with Prefer_in_Cache enabled, we would do the following:

```
# cat prefer_in_cache_enable
List of fq ids with: Prefer in cache :enabled
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,
...
Total FQD with: Prefer in cache : enabled = 560
Total FQD with: Prefer in cache : disabled = 32207
```

7.4.2.2.10.3.11 /sys/kernel/debug/qman/fqd/cge_[enable | disable]

Description:

Query Congestion_Group_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Congestion_Group_Enable bit mask enabled or disabled.

For example, if we want to find all frame queues with Congestion_Group_Enable disabled, we would do the following:

```
# cat cge_disable
List of fq ids with: Congestion Group Enable :disabled
0x000001,0x000002,0x000003,0x000004,0x000005,0x000006,0x000007,0x000008,
0x000009,0x00000a,0x00000b,0x00000c,0x00000d,0x00000e,0x00000f,0x000010,
...
Total FQD with: Congestion Group Enable : enabled = 0
Total FQD with: Congestion Group Enable : disabled = 32767
```

7.4.2.2.10.3.12 /sys/kernel/debug/qman/fqd/cpc_[enable | disable]

Description:

Query CPC_Stash_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their CPC_Stash_Enable bit mask enabled or disabled.

For example, if we want to find all frame queues with CPC Stash disabled, we would do the following:

```
# cat cpc_disable
List of fq ids with: CPC Stash Enable :disabled
0x000001,0x000002,0x000003,0x000004,0x000005,0x000006,0x000007,0x000008,
```

```
0x000009,0x00000a,0x00000b,0x00000c,0x00000d,0x00000e,0x00000f,0x000010,
...
Total FQD with: CPC Stash Enable : enabled = 0
Total FQD with: CPC Stash Enable : disabled = 32767
```

7.4.2.2.10.3.13 /sys/kernel/debug/qman/fqd/cred

Description:

Query Intra-Class Scheduling bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Intra-Class Scheduling Credit value greater than 0.

```
# cat cred
List of fq ids with Intra-Class Scheduling Credit > 0
Total FQD with ics_cred > 0 = 0
```

7.4.2.2.10.3.14 /sys/kernel/debug/qman/fqd/ctx_a_stashing_[enable | disable]

Description:

Query Context_A bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Context_A bit mask enabled or disabled.

For example, if we want to find all frame queues with Context_A enabled, we would do the following:

```
# cat ctx_a_stashing_enable
List of fq ids with: Context-A stashing :enabled
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001de,0x0001df,0x0001e0,0x0001e1,
0x0001ea,0x0001eb,0x0001ec,0x0001ed,0x0001f6,0x0001f7,0x0001f8,0x0001f9,
...
Total FQD with: Context-A stashing : enabled = 528
Total FQD with: Context-A stashing : disabled = 32239
```

7.4.2.2.10.3.15 /sys/kernel/debug/qman/fqd/hold_active_[enable | disable]

Description:

Query Hold_Active bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Hold_Active bit mask enabled or disabled.

For example, if we want find all frame queues with Hold_Active enabled, we would do the following:

```
# cat hold_active_enable
List of fq ids with: Hold active in portal :enabled
Total FQD with: Hold active in portal : enabled = 0
Total FQD with: Hold active in portal : disabled = 32767
```

7.4.2.2.10.3.16 /sys/kernel/debug/qman/fqd/orp_[enable | disable]

Description:

Query ORP bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their ORP bit mask enabled or disabled.

For example, if we want find all frame queues with ORP enabled, we would do the following:

```
# cat orp_enable
List of fq ids with: ORP Enable :enabled
Total FQD with: ORP Enable : enabled = 0
Total FQD with: ORP Enable : disabled = 32767
```

7.4.2.2.10.3.17 /sys/kernel/debug/qman/fqd/sfdr_[enable | disable]

Description:

Query Force_SFDR_Allocate bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Force_SFDR_Allocate bit mask enabled or disabled.

For example, if we want to find all frame queues with Force_SFDR_Allocate enabled, we would do the following:

```
# cat sfdr_enable
List of fq ids with: High-priority SFDRs :enabled(1)
Total FQD with: High-priority SFDRs : enabled = 0
Total FQD with: High-priority SFDRs : disabled = 32767
```

7.4.2.2.10.3.18 sys/kernel/debug/qman/fqd/state_[active | oos | parked | retired | tentatively_sched | truly_sched]

Description:

Query Frame Queue State in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which are in the specified state: active, oos, parked, retired, tentatively scheduled or truly scheduled.

For example, the following returns all the frame queues in the Tentatively Scheduled state:

```
# cat state_tentatively_sched
List of fq ids in state: Tentatively Scheduled
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,
...
Out of Service count = 32201
Retired count = 0
Tentatively Scheduled count = 566
Truly Scheduled count = 0
Parked count = 0
Active, Active Held or Held Suspended count = 0
```

7.4.2.2.10.3.19 /sys/kernel/debug/qman/fqd/tde_[enable | disable]

Description:

Query Tail_Drop_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Tail_Drop_Enable bit mask enabled or disabled.

For example, the following returns all the frame queues with Tail_Drop_Enable bit enabled:

```
# cat tde_enable
List of fq ids with: Tail-Drop Enable :enabled(1)
```

```
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,
...
Total FQD with: Tail-Drop Enable : enabled = 560
Total FQD with: Tail-Drop Enable : disabled = 32207
```

7.4.2.2.10.3.20 /sys/kernel/debug/qman/fqd/wq

Description:

Query Destination Work Queue in all frame queue descriptors. This file takes as input work queue id combined with channel id (destination work queue). The output of this file returns all the frame queues with destination work queue number as specified in the input.

For example, the following returns all the frame queues with their destination work queue number equal to 0x10f:

```
# echo 0x10f > wq
# cat wq
List of fq ids with destination work queue id = 0x10f
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001de,0x0001df,0x0001e0,0x0001e1,
0x0001ea,0x0001eb,0x0001ec,0x0001ed,0x0001f6,0x0001f7,0x0001f8,0x0001f9,
0x0001fa,0x0001fb,0x0001fd,0x0001fe
Summary of all FQD destination work queue values
Channel: 0x0 WQ: 0x0 WQ_ID: 0x0, count = 32199
Channel: 0x0 WQ: 0x0 WQ_ID: 0x4, count = 1
Channel: 0x0 WQ: 0x3 WQ_ID: 0x7, count = 64
Channel: 0x1 WQ: 0x3 WQ_ID: 0xf, count = 64
Channel: 0x2 WQ: 0x3 WQ_ID: 0x17, count = 64
Channel: 0x3 WQ: 0x3 WQ_ID: 0x1f, count = 64
Channel: 0x4 WQ: 0x3 WQ_ID: 0x27, count = 64
Channel: 0x5 WQ: 0x3 WQ_ID: 0x2f, count = 64
Channel: 0x6 WQ: 0x3 WQ_ID: 0x37, count = 64
Channel: 0x7 WQ: 0x3 WQ_ID: 0x3f, count = 64
Channel: 0x21 WQ: 0x3 WQ_ID: 0x10f, count = 20
Channel: 0x42 WQ: 0x3 WQ_ID: 0x217, count = 8
Channel: 0x45 WQ: 0x0 WQ_ID: 0x228, count = 1
Channel: 0x60 WQ: 0x3 WQ_ID: 0x307, count = 8
Channel: 0x61 WQ: 0x3 WQ_ID: 0x30f, count = 8
Sysfs and Debugfs QMan/BMan interfaces
QMan, BMan API RM, Rev. 0.13
NXP Semiconductors NXP Confidential Proprietary 8-67
Preliminary—Subject to Change Without Notice
Channel: 0x62 WQ: 0x3 WQ_ID: 0x317, count = 8
Channel: 0x65 WQ: 0x0 WQ_ID: 0x328, count = 1
Channel: 0xa0 WQ: 0x0 WQ_ID: 0x504, count = 1
```

7.4.2.2.10.3.21 /sys/kernel/debug/qman/fqd/summary

Description:

Provides a summary of all the fields in all frame queue descriptors. This is a read only file.

```
# cat summary
Out of Service count = 32201
Retired count = 0
Tentatively Scheduled count = 566
Truly Scheduled count = 0
```

```
Parked count = 0
Active, Active Held or Held Suspended count = 0
---------------------------------------
Prefer in cache count = 560
Hold active in portal count = 0
Avoid Blocking count = 528
High-priority SFDRs count = 0
CPC Stash Enable count = 0
Context-A stashing count = 528
ORP Enable count = 0
Tail-Drop Enable count = 560
```

7.4.2.2.10.3.22 /sys/kernel/debug/qman/ccsrmempeek

Description:

Provides access to Queue Manager ccsr memory map. This file takes as input an offset from the QMan CCSR base address. The output of this file returns the 32-bit value of the memory address as specified in the input.

For example, to query the QM IP Block Revision 1 register (which is at offset 0xbf8 from the QMan CCSR base address), we would do the following:

```
# echo 0xbf8 > ccsrmempeek
# cat cccsrmempeek
QMan register offset = 0xbf8
value = 0x0a010101
```

7.4.2.2.10.3.23 /sys/kernel/debug/qman/query_ceetm_xsfdr_in_use

Description:

Query the number of XSFDRs currently in use by the CEETM logic of the DCP portal. This file takes input as the DCP id. The output of the file returns the number of XSFDR in use. Please note this feature is only available in T4/B4 rev2 silicon.

For example, if we want to query XSFDR in use number of DCP 0, we would do the following:

echo 0 > query_ceetm_xsfdr_in_use

cat query_ceetm_xsfdr_in_use

DCP0: CEETM_XSFDR_IN_USE number is 0

7.4.2.2.10.4 BMan debugfs

7.4.2.2.10.4.1 /sys/kernel/debug/bman

Description:

This directory contains various BMan device debugging attributes.

7.4.2.2.10.4.2 /sys/kernel/debug/bman/query_bp_state

Description:

This file requests a snapshot of the availability and depletion state of each of BMan's buffer pools. This is a read only file.

For example, if we want to perform a query we could use this file in the following manner:

cat query_bp_state

bp_id free_buffers_avail bp_depleted

0 yes no

1 no no

2 no no

3 no no

4 no no

5 no no

6 no no

7 no no

8 no no

9 no no

10 no no

11 no no

12 no no

13 no no

14 no no

15 no no

16 no no

17 no no

18 no no

19 no no

20 no no

21 no no

22 no no

23 no no

24 no no

25 no no

26 no no

27 no no

28 no no

29 no no

30 no no

31 no no

32 no no

33 no no

34 no no

35 no no

36 no no

37 no no

38 no no

39 no no

40 no no

41 no no

42 no no

43 no no

44 no no

45 no no

46 no no

47 no no

48 no no

49 no no

50 no no

51 no no

52 no no

53 no no

54 no no

55 no no

56 no no

57 no no

58 no no

59 no no

60 no no

61 no no

62 no no

63 yes no

7.4.2.2.11 Error handling and reporting

This chapter describes the QMan and BMan error handling and reporting.

7.4.2.2.11.1 Handling and Reporting

The QMan and BMan error interrupt sevices routines log the occurrence of every error interrupt. Some error interrupts can be triggered multiple times. To prevent a flood of error logging when this interrupts are raised, they are only logged on their first occurance at which time they are disabled. The logs are generated via the pr_warning() kernel api. At the end of the interrupt service routine the ISR register is cleared. These logs are available on the console, dmesg and related log file.

The following QMan error conditions are logged a single time:

QM_EIRQ_PLWI and QM_EIRQ_PEBI.

The following BMan error conditions are logged a single time:

BM_EIRQ_FLWI (low water mark).

7.4.2.2.12 Operating system specifics

This chapter captures O/S-specific issues and distinctions, as the rest of the document essentially describes the interfaces in a generalized manner.

7.4.2.2.12.1 Portal maintenance

By default, the Linux kernel initializes QMan and BMan portals to perform all processing via interrupt-handling. As such there are no persistent threads or polling requirements in order to use portals in the Linux kernel.

Whereas for USDPAA (linux user space), the default is for all processing to be driven by polling, and support for the use of interrupts is disabled. The applications are required to call qman_poll() and bman_poll() within their run-to-completion loops to ensure that portal processing occurs regularly.

As described in Processing non-interrupt-driven portal duties (BMan) on page 628 (for BMan) and Processing non-interrupt-driven portal duties (QMan) on page 643 (for QMan), it is also possible to dynamically control at run-time which portal duties are interrupt-driven versus poll-driven, so the aforementioned defaults for Linux are start-up defaults. However, USDPAA needs to be built with "CONFIG_FSL_DPA_IRQ_SAFETY" defined in order to allow any duties to be interrupt-driven, whereas it is disabled by default (in inc/public/conf.h) due to a very slight performance improvement that it yields.

7.4.2.2.12.2 Callback context

In the Linux kernel, all interrupt-driven portal duties are handled in interrupt context, whereas all other portal duties are invoked from within the qman_poll() and bman_poll() functions, which are invoked by the application.

In USDPAA, even interrupt-driven portal duties are handled in an application context. Interrupts are handled within the kernel and locally disabled, and the presence of such interrupt events is available to the application via the USDPAA file-descriptor representing the portal devices. See Interrupt handling on page 675 for more information. Interrupt-driven portal duties are thus processed when the application calls the qman_thread_irq() and bman_thread_irq() functions, and other portal duties are processed when the application calls qman_poll() and bman_poll().

7.4.2.2.12.3 Blocking semantics

Many high-level QMan and BMan API functions provide "WAIT" flags, to allow the API to block as part of its operation.

In the Linux kernel, "WAIT" behavior is implemented by allowing the calling thread to sleep until a given condition is satisfied. The limitation then to using "WAIT" flags is that the caller can not be in atomic context - i. e. not executing within an interrupt handler, tasklet, bottom-half, etc, nor with any spinlocks held. One consequence is that "WAIT" flags can not be used within a callback.

On run-to-completion systems such as USDPAA, "WAIT" behavior is unsupported and unavailable.

7.4.3 Configuring DPAA Frame Queues

7.4.3.1 Introduction

Describes configurations of Queue Manager (QMan) Frame Queues (FQs) associated with Frame Manager (FMan) network interfaces for the QorIQ Data Path Acceleration Architecture (DPAA). The relationship of the FMan and the QMan channels and work queues are illustrated by examples.

The basic configuration examples for QMan FQs provided yield straightforward and reliable DPAA performance. These simple examples may then be fine tuned for special use cases. For additional information and understanding of advanced system level features please refer to the DPAA Reference Manual.

DPAA provides the networking specific I/Os, accelerator/offload functions, and basic infrastructure to enable efficient data passing, without locks or semaphores, within the multi-core QorIQ SoC between:

1. The Power Architecture cores (and software)
2. The network and I/O interfaces through which that data arrives and leaves
3. The accelerator blocks used by the software to assist in processing that data.

Hardware-managed queues which reside in and are managed by the QMan provide the basic infrastructure elements to enable efficient data path communication. The data resides in delimited work units of frames/packets between cores, hardware accelerators and network interfaces. These hardware-managed queues, known as Frame Queues (FQs), are FIFOs of related frames. These frames comprise buffers that hold a data element, generally a packet. Frames can be single buffers or multiple buffers (using scatter/gather lists).

FQ assignment to consumers i.e., cores, hardware accelerators, network interfaces, are programmable (not hard coded). Specifically, FQs are assigned to work queues which in turn are grouped into channels. Channels which represent a grouping of FQs from which a consumer can dequeue from, are of two types:

- Pool channel: a channel that can be shared between consumers which facilitates load balancing/spreading. (Applicable to cores only. Does not apply to hardware accelerators or netwok interfaces)
- Dedicated channel: a channel that is dedicated to a single consumer.

Each pool or dedicated channel has eight (8) work queues. There are two high priority work queues that have absolute, strict priority over the other six (6) work queues which are grouped into medium and low priority tiers. Each tier contains three work queues which are serviced using a weighted round robin based algorithm. More than one FQ can be assigned to the same work queue as channels implementing a 2-level hierarchical queuing structure. That is, FQs are enqueued/dequeued onto/from work queues. Within a work queue a modified deficit round algorithm is used to determine the number of bytes of data that can be consumed from a FQ at the head of a work queue. The FQ, if not empty, is enqueued back onto the tail of the same work queue once its consumption allowance has been met.

NOTE

- The configuration information provided in this document applies to the QorIQ family of SoCs built on Power Architecture and DPAA technology
- The configuration information provided in this document assumes a top bin platform frequency.

7.4.3.2 FMan Network interface Frame Queue Configuration

Configuring the QMan Frame Queues (FQs) associated with the FMan network interfaces for QorIQ DPAA.

Each network interface has an ingress and an egress direction. The ingress direction is defined as the direction from the network interface to the cores. The egress direction is defined as the direction from the cores to the network interfaces.

FQs associated with FMan network interfaces can be either ingress or egress FQs. Ingress FQs are referred to FQs used in the ingress direction to store packets received from network interfaces to be processed by the cores. Egress FQs are referred to FQs used in the egress direction to store packets to be transmitted by FMan out of its network interfaces.

7.4.3.3 FMan network interface ingress FQs configuration

Dependencies for configuration of the ingress Frame Queues (FQs) is dependent on the QMan mechanism used to load balance/spread received packets across the multiple cores in QorIQ DPAA.

Two mechanisms are offered:

1. Dynamic load balancing
   - Load spread the packets (from ingress FQs) to the cores based on actual core availability/readiness.
   - Achieved through the use of QMan pool channel (i.e. a channel which can be shared by multiple cores).
   - Maintaining packet ordering (e.g. when packets are being forwarded) is achieved through the following two mechanisms:
     a. Order preservation; ensures that related packets (e.g. a sequence of packets moving between two end points) are processed in order (and typically one at a time).
     b. Order restoration; allows packets to be processed out of order and then restores their order later on before they are transmitted out to the network interfaces.

- Improves core work load balancing over a static distribution based approach scheme but will not maintain core affinity because a FQ may get processed by multiple cores.

2. Static distribution
   - Static association between FQs and cores; FQs are always processed by the same core.
   - Achieved through the use of QMan dedicated channel (i.e. a channel which supplies FQs to a specific core).
   - Static not dynamic, doesn't react to core load, assigns work to the cores in a static or fixed manner.
   - Does not not require any special order preservation/restoration mechanism as packet ordering is implicitly preserved.

For all of these mechanisms, QMan requires that related packets, which must be processed and/or transmitted in order, be placed on the same FQ. This does not mean that only related packets are placed on a given FQ; many sets of related packets (“flows”) can be placed on a single FQ. FMan is responsible for achieving this placement/FQ selection function through its distribution capabilities. For instance, FMan can be configured to apply a hash function to a set of packet header fields and use the hash value to select the FQ. This set of packet header fields can be for example, a 5-tuple consisting of:

- source IP address
- destination IP address
- protocol
- source port
- destination port

Note that the FMan processing may be out of order, but it has internal mechanism to ensure that packets are enqueued in order of reception.

These mechanisms can be configured and used simultaneously on an SoC device.

7.4.3.4 Ingress FQs common configuration guidelines

Guidelines and examples for configuring ingress Frame Queues (FQs) in the QorIQ DPAA are shown.

Following guidelines apply regardless of the load balancing mechanism(s) configured:

- Maximum number of ingress FQs for all ingress interfaces on the device (including any of the separate FQs that are used to serve as an order restoration point (ORP)): 1024
- Maximum number of ingress FQs per work queue (FIFO of FQs):
- • 64 if the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s.
  - 128 if the aggregate bandwidth of the configured network interface(s) on the device is 10 Gbit/s or lower.
- The aggregate bandwidth of the configured network interface(s) on the device receiving packets into FQs associated to the same work queue should not exceed 10 Gbit/s. In other words, the recommended maximum incoming rate into a single work queue is 10 Gbit/s. If the configured network interface(s) on the device is higher than 10 Gbit/s, then multiple work queues should be used.
- Since the Single Frame Descriptor Record (SFDRs) reservation scheme is recommended for the egress FQs (FMan network interface egress FQs configuration) and any other FQs assigned to high priority work queues will also use these reserved SFDRs, careful consideration should be given to the required number of ingress FQs assigned to the high priority work queues as SFDRs are a scarce QMan resource (there is a total of 2K SFDRs). One needs to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. ingress FQs assigned to medium or low priority work queues).

As an example, if one allocates 1024 ingress FQs and the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s, then a minimum of 16 work queues would be required based on the above guidelines. Assuming that all 1024 FQs are to be scheduled at the same priority using a dynamic load balancing scheme, a minimum of 6 pool channels would need to be used (based on the fact that up to 3 work queues can be used within a medium or low priority tier).

The guideline “maximum of 1024 ingress FQs for all ingress interfaces” results from the size of the internal memory in QMan that is used to cache Frame Queue Descriptors (FQDs). This internal memory is sized to 2K entries. To achieve high, deterministic and reliable performance under worst-case packet workload (back-to-back 64-byte packets enqueued to FQs on a rotating basis), all ingress FQDs must remain in the QMan internal cache. FQD cache misses increase the time required to enqueue packets as the FQD may need to be read from external memory. This in return could result in received packets being discarded by the MAC due MAC FIFO overflow condition as a result of the back-pressure applied by the FMan to the MAC as there is little buffering between the MAC and the point at which incoming packets are enqueued onto the ingress FQs.

Although a device configured with a number of ingress FQs higher than the size of the QMan FQD internal cache would operate at high performance with no packet discards if the incoming traffic exhibited some level of temporal locality, it is generally recommended that the device be engineered such that ingress path operates at line rate under worst case packet workload to avoid unnecessary packets losses and to make effective use of QMan to prioritize and apply appropriate QoS if there is congestion in a downstream element (e.g. cores). Since all FQs defined on the device shared the QMan 2K internal FQD cache, the recommended maximum number of ingress and egress FQs is even more constrained so that there is adequate space left for caching FQDs assigned to accelerators.

With regards to congestion management, the default mechanism for managing ingress FQ lengths is through buffer management. Input to FQs is limited to the availability of buffers in the buffer pool used to supply buffers to the FQs. Although very efficient and simple, when a buffer pool is shared by multiple FQs, there is no protection between the FQs sharing the buffer pool and as a result a FQ could potentially occupy all the buffers.

Queue management mechanisms can be configured (e.g. tail drop/WRED) to improve congestion control however appropriate software must be in place to handle enqueue rejections as a result of queue congestion.

7.4.3.5 Dynamic load balancing with order preservation - ingress FQs configuration guidelines

Dynamic load balancing with order preservation provides a very effective workload distribution technique to achieve optimal utilization of all cores as it distributes packets to the cores based on actual core availability/readiness.

Order preservation allows FQs to be dynamically reassigned from one core to another while preserving per-FQ packet ordering. It never allows packets from the same FQ to be processed at multiple cores at the same time; a specific FQ is only processed by one core at any given time. Once the FQ is released by the core, it can be processed by any of the cores. To keep multiple cores active there must be multiple FQs distributing packets to the cores, each with a set of (potentially) related packets.

In packet-forwarding scenarios, Discrete Consumption Acknowledgement (DCA) embedded in the enqueue commands should be used to forward packets as this ensures that QMan will release the ingress FQ on software’s behalf once it has finished processing the enqueue command. This provides order preservation semantic from end-to-end (from dequeue to enqueue). To support the above, software portals that will be issuing DCA notifications to QMan must be configured with DCA mode enabled.

Following are specific configuration guidelines for ingress FQs used for dynamic load balancing with order preservation:.

- FQ must be associated to a pool channel (i.e. a channel which can be shared by multiple cores).
- Within a pool channel, minimum number of FQs per active portal (core): 4.
- Frame Queue Descriptor (FQD) attributes settings:
  - Prefer in cache.
  - Hold active set.
  - Don’t set avoid blocking.
  - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required.
  - Don’t set force SFDR allocate unless FQ needs performance optimization.
  - FQD CPC stashing enabled.

- Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
- Order Restoration Point (ORP) disabled.

7.4.3.6 Dynamic load balancing with order restoration - ingress FQs configuration guidelines

Dynamic load balancing with order restoration dispatches packets from the same Frame Queue (FQ) to different processor cores without attempting to maintain order. QMan provides order restoration with specific configurations shown.

The packet order in the original FQ (e.g. ingress FQ) is restored once the cores complete its processing and return the packets to QMan for sending to the next destination (e.g. egress FQ for transmission).

Dynamic load balancing with order restoration has the advantage that parallel processing of related traffic is possible; allows to process without packet drops a flow that exceed the processing rate of a core. However order restoration does make use of more resources than the other distribution schemes. Its usage must also be balanced with applications need to atomically access shared data.

Order restoration is achieved through the following two QMan components:

- Order Definition Points (ODPs)
  - A point through which packets pass, where their order or sequence relative to each other is defined.
  - For convenience each FQ has an ODP for packets dequeued from that FQ.
- Order Restoration Points (ORPs)
  - A point through which packets pass, where their order or sequence is restored to that defined at the related ODP.
  - If a packet is out of sequence it is held until it is in sequence.
  - ORP data structure is maintained in a FQ; it is recommended that a dedicated/separate FQ be allocated solely for this purpose.

Following are specific configuration guidelines for ingress FQs used for dynamic load balancing with order restoration:

- FQ must be associated to a pool channel (i.e. a channel which can be shared by multiple cores).
- For each ingress FQ supporting order restoration, a separate FQ should be allocated to serve as the ORP.
- Ingress FQ descriptor attributes settings.
  - Prefer in cache
  - Don’t set hold active.
  - Set avoid blocking.
  - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required.
  - Don’t set force SFDR allocate unless FQ needs performance optimization.
  - FQD CPC stashing enabled.
  - Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
  - ORP disabled.

Following are specific configuration guidelines for ORP FQs:

- FQs used for ORP don’t need to be associated with a pool or dedicated channel.
- ORP FQ descriptor attributes settings:
  - Prefer in cache .
  - Don’t set hold active.
  - Don’t set avoid blocking.

- Intra-class scheduling credit set to 0.
- Don't set force SFDR allocate .
- FQD CPC stashing enabled.
- ORP enabled.
- Recommended ORP restoration window size: 128.

7.4.3.7 Static distribution - Ingress FQs Configuration Guidelines

With a static distribution approach, a single FQ is always processed by the same processor core. Specific guidelines for processor core affinity are presented.

Although not as effective as a dynamic based approach from a resource utilization aspect, static distribution maintains core affinity meaning that the mapping from the flow to the core is preserved.

Distribution of packets (selection of FQ) can based on hash keys, ensuring that packets from the same traffic flow will always go to the same cores. The FQ selection function is achieved by FMan.

Following are specific configuration guidelines for ingress FQs used for static distribution:

- FQ must be associated to a dedicated channel (i.e. a channel which supplies FQs to a specific core); multiple FQs can be associated to a single dedicated channel.
- Within a dedicated channel, minimum number of FQs: 1.
- FQ descriptor attributes settings:
  - Prefer in cache .
  - Don't set hold active
  - Don't set avoid blocking.
  - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required. On P4080/P3041/P5020, due to errata number QMAN-A002, allowable values for ICS are: 0 and 15' h7FFF.
  - Don't set force SFDR allocate unless FQ needs performance optimization.
  - FQD CPC stashing enabled.
  - Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
  - ORP disabled.

7.4.3.8 FMan network interface egress FQs configuration

Configuration guidelines for egress Frame Queues (FQs) for QorIQ DPAA

FQ Configurations:

- Maximum number of egress FQs for all network interfaces: 128.
- Minimum number of egress FQs per network interface: 1.
- Maximum number of egress FQs per work queue: 8.
- Egress FQ descriptor attributes settings:
  - Prefer in cache.
  - Don't set hold active .
  - Don't set avoid blocking.
  - Set force SFDR allocate to ensure that egress queues make use of the reserved SFDRs; the SFDR reservation threshold field of the QMan SFDR configuration register must also be set accordingly (5 SFDRs per egress FQ + 3 extra SFDRs as required by QMan).

- Intra-class scheduling set to zero (0) unless a more advanced scheduling scheme is required.
- FQD CPC stashing enabled.
- ORP disabled.

.

7.4.3.9 Accelerator Frame Queue Configuration

Configurations for Frame Queues (FQs) used to communicate with accelerators for QorIQ DPAA are shown.

FQ accelerator Guidelines:

- Since the Single Frame Descriptor Record (SFDRs) reservation scheme is recommended for the egress FQs (FMan network interface egress FQs configuration) and any other FQs assigned to high priority work queues will also use these reserved SFDRs, careful consideration should be given to the required number of accelerator FQs assigned to the high priority work queues as SFDRs are a scarce QMan resource (there is a total of 2K SFDRs). One needs to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. accelerator FQs assigned to medium or low priority work queues).
- Accelerator FQ descriptor attributes settings:
  - Don't set prefer in cache.
  - Don't set hold active .
  - Don't set avoid blocking.
  - FQD CPC stashing enabled.
  - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required.
  - Don't set force SFDR allocate unless FQ needs performance optimization.
  - Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
  - ORP disabled.

Generally accelerators are used in a request/response manner and in cases where a pair of FQs is needed per session/flow to communicate with accelerators, one may need to allocate a very large number of FQs (in the order of thousands). At times when many FQs allocated to an accelerator are active, this situation can result in having significant amount of cache consumed for storing the corresponding FQ descriptors. This in turn may negatively impact overall system performance.

To ensure optimal resource utilization (e.g. QorIQ caches), maximize throughput and avoid overload, it is recommended that the number of outstanding requests/responses to an accelerator be regulated. Typically, for a given accelerator, regulating the number of outstanding requests/responses across all its FQs to a few hundredths should be sufficient to maintain high throughput without overloading the system. Regulating the number of outstanding requests/responses to an accelerator can be achieved through various methods.

One method is to keep track in software of the total number of outstanding requests/responses to an accelerator and once this number exceeds a threshold, software would stop sending requests to that accelerator.

Another method is to make use of the congestion management capabilities of QMan. Specifically, all FQs allocated to an accelerator can be aggregated into a congestion group. Each congestion group can be configured to track the number of Frames in all FQs in the congestion group. Once this number exceeds a configured threshold, the congestion group enters congestion. When a congestion group enters congestion, QMan can be configured to rejects enqueues to any FQs in the congestion group and/or sent notification indicating that the congestion group has entered congestion. If a Frame (or request) is not going to be enqueued, it will be returned to the configured destination via an enqueue rejection notification. Congestion state change notifications are generated when the congestion group either enters congestion or exits congestion. On software portals, the congestion state change notification is sent via an interrupt.

7.4.3.10 DPAA Frame Queue Configuration Guideline Summary

Summary of Configurations for Frame Queue (FQ) communication with accelerators for QorIQ DPAA

Four tables comprise this summary:

- Global Configuration settings
- Network interface ingress FQ guidelines
- Network interface egress FQ guidelines
- Accelerator FQ guidelines

Table 121. Global Configuration Settings Summary

Parameter or subject	Guideline
FQD stashing	Recommend QMan explicitly stash FQDs: • QMan; both the global CPC stash enable bit in the QMan FQD_AR register and the CPC stash enable bit in the FQD must be set. • PAMU; PAACT tables used by PAMU also configured appropriately .
PFDR stashing	Recommend QMan explicitly stash PFDRs: • QMan; the global CPC stash enable bit in the QMan PFDR_AR register must be set . • PAMU; PAACT tables used by PAMU must also be configured appropriately .
SFDR reservation threshold	Set SFDR reservation threshold in QMan SFDR configuration register to: • Total number of FQs using reserved SFDRs times 5 (5 SFDRs per FQ) plus 3 extra SFDRs as required by QMan. Recommend that all egress FQs use reserved SFDRs .

Table 122. Network Interface Ingress FQs Guidelines Summary

Parameter or subject	Guideline
Maximum number of ingress FQs for all ingress interfaces on the device (including any of the separate FQs that are used to serve as an order restoration point (ORP))	1024 FQs
Maximum number of ingress FQs per work queue.	• 64 FQs per work queue if the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s. • 128 FQs per work queue if the aggregate bandwidth of the configured network interface(s) on the device is 10 Gbit/s or lower.
The maximum aggregate bandwidth of the configured network interface(s) on the device receiving packets into FQs associated to the same work queue	10 Gbit/s

Table continues on the next page...

Table 122. Network Interface Ingress FQs Guidelines Summary (continued)

Parameter or subject	Guideline
Within a pool channel, minimum number of FQs per active portal (cores).	4 FQs
Within a dedicated channel, minimum number of FQs:	1 FQ
Assignment to high priority work queues.	Should be limited enough to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. ingress FQs assigned to medium or low priority work queues).
Order restoration point (ORP).	A separate FQ should be allocated and dedicated to serve as the ORP for each ingress FQ supporting order restoration.

Ingress FQ descriptor load balancing and performance related settings.	• Prefer_in_Cache: 1
	• CPC Stash Enable: 1
	• ORP_Enable: 0
	• Avoid_Blocking:
	  • 0 if static distribution or dynamic load balancing with order preservation.
	  • 1 if dynamic load balancing with order restoration.
	• Hold_Active
	  • 0 if static distribution or dynamic load balancing with order restoration .
	  • 1 if dynamic load balancing with order preservation.
	• Force_SFDR_Allocate: 0 unless FQ needs performance optimization.
	• Intra-Class Scheduling Credit: 0 unless a more advanced scheduling scheme is required.

ORP FQ descriptor order restoration and performance related settings.	• Prefer_in_Cache: 1 • CPC Stash Enable: 1 • ORP_Enable: 1 • Avoid_Blocking: 0 • Hold_Active: 0 • Force_SFDR_Allocate: 0 • ORP Restoration Window Size: 2 (corresponds to window size of 128 frames). • Class Scheduling Credit: 0

Table 123. Network Interface Egress FQs Guidelines Summary

Parameter or subject	Guideline
Maximum number of egress FQs for all network interfaces.	128 FQs
Minimum number of egress FQs per network interface.	1 FQ
Maximum number of egress FQs per work queue.	8 FQs
Egress FQ descriptor performance related settings.	• Prefer_in_Cache: 1 • CPC Stash Enable: 1 • ORP_Enable: 0 • Avoid_Blocking: 0 • Hold_Active: 0 • Force_SFDR_Allocate: 1 • Class Scheduling Credit: 0 unless a more advanced scheduling scheme is required.

Table 124. Accelerator FQs Guidelines Summary

Parameter or subject	Guideline
Assignment to high priority work queues.	Should be limited enough to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. accelerator FQs assigned to medium or low priority work queues).
Egress FQ descriptor performance related settings.	• Prefer_in_Cache: 0 • CPC Stash Enable: 1 • ORP_Enable: 0 • Avoid_Blocking: 0 • Hold_Active: 0 • Force_SFDR_Allocate: 0 unless FQ needs performance optimization . • Class Scheduling Credit: 0 unless a more advanced scheduling scheme is required .

7.4.4 Frame Manager

7.4.4.1 Frame Manager Linux Driver User Guide

7.4.4.1.1 Introduction

This part is describing the Linux implementation of the driver for the Frame Manager, or FMD.

The Linux driver for the Frame Manager is based on the NetCommSw drivers, or NCSW. The NCSW drivers are written in an OS-agnostic fashion, so what the Linux FMD does is to implement a set of standard Linux character devices that rely on the NCSW drivers to do the actual communication with the hardware. The figure below describes this best:



Figure 88. FMan-centric view of relationships between DPAA software and hardware blocks in the Linux environment.

The features of the Linux FMan Driver are the following:

- Performs initialization of the Frame Manager based on platform configuration (device tree), and on probing of the actual hardware;
- Supports Linux user space applications looking to create FMan PCD configurations;
- Attaches/detaches PCDs to/from FMan ports;
- Reports FMan and port status:
  - FMan registers

- FMan statistics
- FMan port and MAC counters

The Linux FMan driver does not handle actual network traffic. Network traffic in Linux is being handled exclusively by Linux network devices. Network traffic going through FMan can only be handled by the Linux DPAA Ethernet driver. Although the DPAA Ethernet and the Linux FMan Driver share strong links and interdependencies with the underlaying low-level FMD and with each other, their feature sets do not overlap. The DPAA Ethernet driver is described in the *Linux Ethernet Driver User Manual*.

Since the Linux FMD implementation relies heavily on the NCSW FMD, we strongly recommend *NetCommSw's Frame Manager Driver User's Guide* as prerequisite reading.

The USDPAA is a special case not detailed here.

7.4.4.1.2 The Linux FMD Devices

The Linux interface to the FMD consists in several Linux character devices:

- `/dev/fm[0,1]`, each corresponding to an actual Frame Manager;
- `/dev/fm[0,1]_pcd` are PCD devices corresponding each to a Frame Manager;
- `/dev/fm[0,1]_port_rx[0-4]`, and `/dev/fm[0,1]_port_tx[0-4]` corresponding to the physical ports of each FMan: each rx/tx device in a pair corresponds to the receive and transmit sides of a physical port;
- `/dev/fm[0,1]_port_oh[0-6]` correspond to the Offline Parsing ports.

These devices' creation and initialization is performed at boot time, based on probing of the physical hardware, as well as on the parsing of the device tree. Each of the physical ports can thus be disabled from the device tree, but also from the Reset Configuration Word (for details please consult the **Reset Configuration Word (RCW)** section of the **P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual**, or similar RM for other platform; for other useful reading please consult *Selecting Ethernet Interfaces* in the **QorIQ SDK Ethernet** manual).

NOTE

The assumption for the remainder of this document is that the device tree and the RCW are immutable, and therefore no attempt will be made to describe their formats in here. For details regarding the RCW/.dts, please consult the relevant documentation.

Therefore, depending on platform and on RCW/.dts configuration, some of these devices may be missing. The mapping of these devices to the physical ports is given in the following table:

Table 125. Mapping of Linux devices to low-level port IDs.

Linux Device	Low-Level ID	Identification
`/dev/fm0_port_rx0` `/dev/fm0_port_tx0`	0	1st FMan's 1st 1GbE Receive, Transmit
`/dev/fm0_port_rx1` `/dev/fm0_port_tx1`	1	1st FMan's 2nd GbE Receive, Transmit
`/dev/fm0_port_rx2` `/dev/fm0_port_tx2`	2	1st FMan's 3rd GbE Receive, Transmit
`/dev/fm0_port_rx3` `/dev/fm0_port_tx3`	3	1st FMan's 4th GbE Receive, Transmit

Table continues on the next page...

Table 125. Mapping of Linux devices to low-level port IDs. (continued)

Linux Device	Low-Level ID	Identification
/dev/fm0_port_rx4 /dev/fm0_port_tx4	4	1st FMan's 5th GbE[4] Receive, Transmit
/dev/fm0_port_rx5 /dev/fm0_port_tx5	5	1st FMan's 10Gb Receive, Transmit
N/A	0	1st FMan's Host Command
/dev/fm0_port_oh0	1	1st FMan's 1st Offline Parsing
/dev/fm0_port_oh1	2	1st FMan's 2nd Oflline Parsing
/dev/fm0_port_oh2	3	1st FMan's 3rd Offline Parsing
/dev/fm0_port_oh3	4	1st FMan's 4th Offline Parsing
/dev/fm0_port_oh4	5	1st FMan's 5th Offline Parsing
/dev/fm0_port_oh5	6	1st FMan's 6th Offline Parsing
/dev/fm0_port_oh6	7	1st FMan's 7th Offline Parsing
/dev/fm1_port_rx0 /dev/fm1_port_tx0	0	2nd FMan's 1st 1GbE Receive, Transmit
/dev/fm1_port_rx1 /dev/fm1_port_tx1	1	2nd FMan's 2nd 1GbE Receive, Transmit
/dev/fm1_port_rx2 /dev/fm1_port_tx2	2	2nd FMan's 3rd 1GbE Receive, Transmit
/dev/fm1_port_rx3 /dev/fm1_port_tx3	3	2nd FMan's 4th 1GbE Receive, Transmit
/dev/fm1_port_rx4 /dev/fm1_port_tx4	4[5]	N/A
/dev/fm1_port_rx5 /dev/fm1_port_tx5	5	2nd FMan's 10Gb Receive, Transmit
N/A	0	2nd[6] FMan's Host Command
/dev/fm1_port_oh0	1	2nd FMan's 1st Offline Parsing Port
/dev/fm1_port_oh1	2	2nd FMan's 2nd Offline Parsing Port
/dev/fm1_port_oh2	3	2nd FMan's 3rd Offline Parsing Port
/dev/fm1_port_oh3	4	2nd FMan's 4th Offline Parsing Port
/dev/fm1_port_oh4	5	2nd FMan's 5th Offline Parsing Port
/dev/fm1_port_oh5	6	2nd FMan's 6th Offline Parsing Port

Table continues on the next page...

[4] Only on P5020.
[5] This port & devices are not available on any of the platforms supported by this release!
[6] Only P4080 has a second FMan.

Table 125. Mapping of Linux devices to low-level port IDs. (continued)

Linux Device	Low-Level ID	Identification
/dev/fm1_port_oh6	7	2nd FMan's 7th Offline Parsing Port

The Low Level IDs are the IDs that are used by the Low Level Drivers (upon which the Linux FMan Driver is based) to distinguish between the physical ports. It is obvious from the above table that the port ID alone does not allow for uniquely identifying a single port. It has to be combined wiht the following information in order to succeesfully point to the desired port:

- FMan ID: 0 or 1 for FMan1 or 2, respectively;
- Port type: 1G, 10G or O/H (Offline Parsing/Host Command).

Although all this may seem confusing at first, the LLD API provides convenient enums/macros to deal with these aspects. Furthermore, the FMD driver API tries its best to hide these details from the userspace Linux programmer, specifically by using dedicated /dev entries for each port, etc. However, not all userspace-visible API is free of such port IDs, so this is why we even mention them here.

The FMD LLD uses no distinct port IDs for Rx and Tx, the distinction between Receive and Transmit being made by calling distinct Rx/Tx-specific functions, or by specifying the "RX" or "TX" direction as a separate argument.

The Host Command ports are invisible to the Linux application. One needs to be aware, though, of their mere existence at the least, since the LLD allocates the first physical O/H port of every FMan to this purpose ("O/H" standing for "Offline Parsing/Host Command"). There are 8 such O/H ports on each FMan that can be used for these purposes; the first of these having been dedicated by the LLD to Host Commands, while the remaining 7 being available for Offline Parsing. Host Commands are just one of the vehicles through which the LLD exercises control of the FMan hardware.

NOTE

Please note that depending on the platform, RCW, and .dts configuration not all the possible combinations of devices and ports are possible, and most certainly some will be missing from any existing configuration. For details regarding possible port & device configurations for a specific platform, please consult the Reference Manuals for that platform, as well as the relevant chapters from the SDK documentation for that platform.

Alongside these character devices, and out of the scope of this writing, are the Linux network devices, labeled using the `fm[1,2]-gb[0-4]` (e.g. `fm1-gb0`, `fm2-gb3`) and `fm[1,2]-10g` (i.e. `fm1-10g`, `fm2-10g`) schemes, which provide the means for Linux to handle actual network traffic, i.e. "traffic termination". These network devices are instances of the Linux DPAA Ethernet Driver, which is architected as a separate entity from the Linux FMan Driver, but which both make use at some point of the same Low-Level Driver FMD API. The feature sets of the DPAA Ethernet and of the Linux FMan drivers are disjunct, though, which is the main reason for their coexistence.

NOTE

There is no requirement that these are the only network devices in the system. You may find the well known `eth0`, `eth1`, etc. devices alongside e.g. `fm1-gb0`, except that these other network devices will correspond to other vendors' NICs that may be installed in the system and will be serviced by vendor-specific, non-DPAA, Ethernet drivers.

There are a few constants #defined in the headers that need to be included when working with the Linux FMD (in both kernel and user spaces) that may come in handy when having to deal with devices and port IDs:

- `FM_MAX_NUM_OF_1G_RX_PORTS`
- `FM_MAX_NUM_OF_10G_RX_PORTS`
- `FM_MAX_NUM_OF_1G_TX_PORTS`
- `FM_MAX_NUM_OF_10G_RX_PORTS`
- `FM_MAX_NUM_OF_RX_PORTS`

- FM_MAX_NUM_OF_TX_PORTS
- FM_MAX_NUM_OF_OH_PORTS
- IOC_FM_MAX_NUM_OF_VALID_PORTS

that together with INTG_MAX_NUM_OF_FM can give the programmer the essential tools to get around in a specific configuration (this list, though, is not exhaustive: please consult the relevant API Reference/header files before attempting to #define your own).

Also, the

```
$ ls /dev/fm*
```

Linux shell command can conveniently show all the FMD devices currently available in the target system.

7.4.4.1.3 Linux FMD Programming Model

Given the Linux devices presented earlier, a Linux application looking to use the FMan features can use the general Linux character device syscall interface:

- open()/close() - this is essential API when working with Linux devices.
- read()/write() - although read() and write() operations are mandatory to be implemented by all Linux devices, there are no read/write semantics associated with the FMD devices.
- ioctl() calls are used extensively as the only means to communicate with the hardware. The ioctl API does little more than delegating the ioctl() syscall to the underlying LLD API (for the actual mapping of IOCTLs to actual LLD APIs, please consult the tables available in the following sections).

We'll state here once more that the programming model is essentially that of the FMD LLD. The Linux wrapper merely adapts the LLD to the Linux interface requirements. This part of the SDK documentation focuses only on the Linux specifics. For details regarding individual API calls, please refer to the *Frame Manager Driver API Reference Manual* .

As is the case with any Linux device, the general sequence of actions when using the FMD devices is the following:

1. Linux boots: all /dev/fm* devices are being created, FMan resources initialized according to platform/RCW/.dts;
2. User launches FMD-aware application;
3. User app. performs open() on selected /dev/fm* device/s;
4. User app. performs ioctl() call/s on the fd returned by the previous successful open() call;
5. When the user app. decides it has finished working with selected /dev/fm* device, it must call close() on its fd, just like on any other Linux device.

Not all the LLD functions have a correspondent in the FMD IOCTLs. Only those functions have been selected which makes sense from an architectural standpoint. The same/other LLD functions are also being called by the Linux wrapper unrestrictedly, as needed to perform its required actions, and not only in response to ioctl() calls.

The arguments of the ioctl() calls can be quite complex, and may have complex requirements, as they are described in the **LLD API Reference** (Frame Manager Driver API Documentation).

The following required low-level initialization APIs: FM_Config(), FM_PCD_Config(), FM_PORT_Config(), and subsequently FM_Init(), FM_PCD_Init(), FM_PORT_Init() are being called from within the Linux FMD initialization code at boot time. They are therefore not accessible to the user space application. Any configuration of FMan hardware resources will be performed using Linux-specific means: device tree, kernel build configuration, etc. Code in the DPAA Ethernet driver also initializes the configured MACs using FM_MAC_Config(), then FM_MAC_Init(), as required by the *Frame Manager Driver API Reference Manual*, and as described in *The DPAA Ethernet Driver's User Manual*.

The correspondence between FMD Linux devices and DPAA ETH network devices is intuitive: there is a pair of /dev/fmX_port_(rxY|txY) devices for each fmX-gbY or fmX-10g device in the system. However, due to configuration, it is possible that at boot time not all FMan ports be probed by the DPAA Ethernet driver, hence not all /dev/fmX_port_(rxY|txY) may have a corresponding netdev. This is because the FMan port devices and the DPAA Ethernet devices are being configured in different sections of the device tree. The binding between these devices is also done in the device tree.

While Offline Parsing ports are being fully supported by the FMan Driver, currently it is not possible to inject traffic from user space to these ports, as there is no netdev being created for them, as the Linux FMD does not handle traffic. There is indeed a way for kernel space drivers or e.g. USDPAA apps. to use them, but that is out of scope here.

It is not to be expected that a FMan port device for which a corresponding DPAA Ethernet netdev has not been configured, to be fully functional. That is because port functionality is reliant also upon additional DPAA resources (i.e. frame queues, buffer pools) that are being initialized exclusively by the DPAA Ethernet driver. Therefore, even though `/dev/fmX_port_*` devices may exist for such ports, trying to access them may result in an error.

`FM_PORT_Enable()` and `FM_PORT_Disable()` are called for specific ports during `ifconfig up/down` of the corresponding network device (DPAA Ethernet-specific). They are also available as IOCTLs for the `/dev/fmX_port*` devices, but while in the DPAA Ethernet they are called for both ports of the RX/TX pair, the `/dev/fmX_port_(rxY|txY)` allow for selectively enabling/disabling of only one of the RX/TX sides, as desired.

The `ioctl()` API conforms to Linux rules for all FMD devices. However, errors originating within the LLD will invariably be reported to the user as `-EFAULT`. All such errors should be considered non-recoverable and should be immediately followed by a `close()` on the device for which they were reported. A more descriptive message should be printed on the bootup console only, identifying the LLD function, and the line in the source file where the error has occurred. One can look at the documentation for `enum e_ErrorType` in the **LLD API Reference** (Frame Manager Driver API Documentation) for details regarding all the possible LLD error codes and their general meaning.

The following sections will present a brief description of each type of Linux device, as well as their IOCTLs' mapping to the FMD LLD API.

7.4.4.1.4 The Linux FMan Device

This device corresponds to an individual Frame Manager, and is required for performing FMan-wide actions. The FMan device merely acts as a portal for the IOCTLs that are listed in the table below:

Table 126. IOCTLs for the FMan Device

IOCTL	LLD Mapping	Brief
`FM_IOC_SET_PORTS_BANDWIDTH`	`FM_SetPortsBandwidth()`	Sets ports' bandwidths as percentage of total bandwidth.
`FM_IOC_GET_REVISION`	`FM_GetRevision()`	API to get the FMan's revision.
`FM_IOC_GET_COUNTER`	`FM_GetCounter()`	API to read FMan hardware counters (also available through sysfs).
`FM_IOC_SET_COUNTER`	`FM_ModifyCounter()`	API to modify/reset FMan's counters.
`FM_IOC_FORCE_INTR`	`FM_ForceIntr()`	Forces an FMan interrupt (or exception). Dangerous! Use for debugging only!
`FM_IOC_GET_API_VERSION`	`FM_GetApiVersion()`	Reads the FMD IOCTL API version.
`FM_IOC_VSP_CONFIG`	`FM_VSP_Config()`	Creates descriptor for the FM VSP module.
`FM_IOC_VSP_INIT`	`FM_VSP_Init()`	Initializes the FM VSP module
`FM_IOC_VSP_FREE`	`FM_VSP_Free()`	Frees all resources that were assigned to FM VSP module.

Table continues on the next page...

Table 126. IOCTLs for the FMan Device (continued)

IOCTL	LLD Mapping	Brief
FM_IOC_VSP_CONFIG_POOL_DEPLETION	FM_VSP_ConfigPoolDepletion()	Calling this routine enables pause frame generation depending on the depletion status of BM pools. It also defines the conditions to activate this functionality. By default, this functionality is disabled.
FM_IOC_VSP_CONFIG_BUFFER_PREFIX_CONTENT	FM_VSP_ConfigBufferPrefixContent()	Defines the structure, size and content of the application buffer.
FM_IOC_VSP_CONFIG_NO_SG	FM_VSP_ConfigNoScatherGather()	Returns the pointer to the parse result in the data buffer. In Rx ports this is relevant after reception, if parse result is configured to be part of the data passed to the application. For non Rx ports it may be used to get the pointer of the area in the buffer where parse result should be initialized - if so configured. See FM_VSP_ConfigBufferPrefixContent for data buffer prefix configuration.
FM_IOC_CTRL_MON_START	FM_CtrlMonStart()	Start monitoring utilization of all available FM controllers.
FM_IOC_CTRL_MON_STOP	FM_CtrlMonStop()	Stop monitoring utilization of all available FM controllers.
FM_IOC_CTRL_MON_GET_COUNTERS	FM_CtrlMonGetCounters()	Obtain FM controller utilization parameters.

All the IOCTL-mapped LLD APIs are what the LLD terms as "callable at runtime", i.e. callable after the LLD `Init()` function for the corresponding entity has been called. This is so because by the time the user app. gets to invoke `ioctl()`, all the `Init()` functions have already been called by the initialization code of the Linux FMD at boot time.

7.4.4.1.5 The Linux PCD Device

There is exactly one PCD device, or `/dev/fmX_pcd`, for each Frame Manager. The reason for that is that PCDs are FMan-wide constructs, and are applied simultaneously to traffic being received on possibly more than one port.

"PCD" is a generic term designating a Parse-Classify-Distribute configuration for a group of ports, as described in detail in the **QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual**. In short, what a PCD does is to route incoming traffic from a set of RX ports onto several frame queues managed by the Queue Manager. Such frame queues may be attached to a DPAA Ethernet network device, in which case the traffic is received by the CPUs (or "terminated"), or they can be connected to a TX port, in which case the traffic is being forwarded onto that port. Also, frame queues can be further grouped into work queues & policed, etc. (please read the QMan documentation). However, one thing is not supported in the Linux environment, and that is: direct access to frame queues from user space (please note that this is not a limitation of the Linux FMD, but one enforced by design in the Linux driver for the QMan). Not in the classical meaning of "Linux environment", that is. If one needs to create complex DPAA scenarios that are not possible/cumbersome using the current Linux FMD, then USDPAA may hold the answer they're looking for.

There's still a lot that can be achieved with the Linux FMD, and the Linux PCD device is there to help. Its role is to manage the PCDs for its associated FMan. The ioctls for this device are mapped to the similarly-sounding `FM_PCD_*()` LLD APIs:

Table 127. IOCTL List for the PCD Device

IOCTL	LLD Mapping	Brief
FM_PCD_IOC_ENABLE	FM_PCD_Enable()	Should be called after PCD is initialized for enabling all PCD engines according to their existing configuration.
FM_PCD_IOC_DISABLE	FM_PCD_Disable()	Disables an existing PCD.
FM_PCD_IOC_PRS_LOAD_SW[_COMPAT]	FM_PCD_PrsLoadSw()	This routine may be called only when all ports in the system are actively using the classification plan scheme. In such cases it is recommended in order to save resources. The driver automatically saves 8 classification plans for ports that do NOT use the classification plan mechanism; to avoid this (in order to save those entries) this routine may be called.
FM_PCD_IOC_KG_SET_DFLT_VALUE	FM_PCD_KgSetDfltValue()	Sets a global default value to be used by the key generator when the parser does not recognize a required field/ header (default 0).
FM_PCD_IOC_KG_SET_ADDITIONAL_DATA_AFTER_PARSING	FM_PCD_KgSetAdditionalDataAfterParsing()	Calling this routine allows the keygen to access data past the parser finishing point.
FM_PCD_IOC_SET_EXCEPTION	FM_PCD_SetException()	Enables/disables PCD interrupts.
FM_PCD_IOC_GET_COUNTER	N/A	Unimplemented, do not use!
FM_PCD_IOC_SET_COUNTER	N/A	Placeholder, do not use!
FM_PCD_IOC_FORCE_INTR	FM_PCD_ForceIntr()	Forces a PCD interrupt (exception) of specified type. Dangerous! Use only for debugging!
FM_PCD_IOC_NET_ENV_CHARACTERISTICS_SET[_COMPAT]	FM_PCD_NetEnvCharacteristicsSet()	Establishes a minimal set of networking protocols ("Network Environment Characteristics") that can be discovered by this PCD (please refer to the Reference Manual for details).
FM_PCD_IOC_NET_ENV_CHARACTERISTICS_DELETE[_COMPAT]	FM_PCD_NetEnvCharacteristicsDelete()	Deletes a set of "Network Environment Characteristics".

Table continues on the next page...

Table 127. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
FM_PCD_IOC_KG_SCHEME_SET[_COMPAT]	FM_PCD_KgSchemeSet()	Initializes or modifies and enables a scheme for the KeyGen. This routine should be called for adding or modifying a scheme. When a scheme needs modifying, the API requires that it be rewritten. In such a case modify should be TRUE. If the routine is called for a valid scheme and modify is FALSE, it will return error.
FM_PCD_IOC_KG_SCHEME_DELETE[_COMPAT]	FM_PCD_KgSchemeDelete()	Deletes an initialized scheme.
FM_PCD_IOC_CC_ROOT_BUILD[_COMPAT]	FM_PCD_CcRootBuild()	This routine must be called to define a complete coarse classification tree. This is the way to define coarse classification to a certain flow - the KeyGen schemes may point only to trees defined in this way.
FM_PCD_IOC_CC_ROOT_DELETE[_COMPAT]	FM_PCD_CcRootDelete()	Deletes an existing coarse classification tree.
FM_PCD_IOC_MATCH_TABLE_SET[_COMPAT]	FM_PCD_MatchTableSet()	This routine should be called for each CC (coarse classification) node. The whole CC tree should be built bottom up so that each node points to already defined nodes. p_node_id returns the node Id to be used by other nodes.
FM_PCD_IOC_MATCH_TABLE_DELETE[_COMPAT]	FM_PCD_MatchTableDelete()	Deletes a built node.
FM_PCD_IOC_CC_ROOT_MODIFY_NEXT_ENGINE[_COMPAT]	FM_PCD_CcRootModifyNextEngine()	Modifies the Next Engine Parameters in the entry of the tree (allowed only after FM_PCD_CcBuildTree()).
FM_PCD_IOC_MATCH_TABLE_MODIFY_NEXT_ENGINE[_COMPAT]	FM_PCD_MatchTableModifyNextEngine()	Modifies the Next Engine Parameters in the relevant key entry of the node (possible only after a call to FM_PCD_MatchTableSet()).
FM_PCD_IOC_MATCH_TABLE_MODIFY_MISS_NEXT_ENGINE[_COMPAT]	FM_PCD_MatchTableModifyMissNextEngine()	Modifies the Next Engine Parameters of the Miss key case of the node (allowed only after a previous call to FM_PCD_MatchTableSet()).

Table continues on the next page...

Table 127. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
FM_PCD_IOC_MATCH_TABLE_REMOVE_KEY[_COMPAT]	FM_PCD_MatchTableRemoveKey()	Removes the key (including its next engine parameters) defined by the index of the relevant node (allowed only after a previous call to FM_PCD_MatchTableSet())
FM_PCD_IOC_MATCH_TABLE_ADD_KEY[_COMPAT]	FM_PCD_MatchTableAddKey()	Adds the key (including next engine parameters of this key) in the index defined by key_index (allowed only after a previous call to FM_PCD_MatchTableSet())
FM_PCD_IOC_MATCH_TABLE_MODIFY_KEY_AND_NEXT_ENGINE[_COMPAT]	FM_PCD_MatchTableModifyKeyAndNextEngine()	Modifies the key and Next Engine Parameters of this key in the index defined by key_index (allowed only after a previous call to FM_PCD_MatchTableSet()).
FM_PCD_IOC_MATCH_TABLE_MODIFY_KEY[_COMPAT]	FM_PCD_MatchTableModifyKey()	Modifies the key at the index defined by key_index (allowed only after a previous call to FM_PCD_MatchTableSet()).
FM_PCD_IOC_HASH_TABLE_SET[_COMPAT]	FM_PCD_HashTableSet()	Initializes a hash table structure.
FM_PCD_IOC_HASH_TABLE_DELETE[_COMPAT]	FM_PCD_HashTableDelete()	Deletes the provided hash table and released all its allocated resources.
FM_PCD_IOC_HASH_TABLE_ADD_KEY[_COMPAT]	FM_PCD_HashTableAddKey()	Adds the provided key (including next engine parameters of this key) to the hash table. The key is added as the last key of the bucket that it is mapped to.
FM_PCD_IOC_HASH_TABLE_REMOVE_KEY[_COMPAT]	FM_PCD_HashTableRemoveKey()	Removes the requested key (including its next engine parameters) from the hash table.
FM_PCD_IOC_PLCR_PROFILE_SET[_COMPAT]	FM_PCD_PlcrProfileSet()	Sets a profile entry in the policer profile table, overriding any existing value.
FM_PCD_IOC_PLCR_PROFILE_DELETE[_COMPAT]	FM_PCD_PlcrProfileDelete()	Deletes a profile entry in the policer profile table. It sets the entry to invalid.
FM_PCD_IOC_MANIP_NODE_SET[_COMPAT]	FM_PCD_ManipNodeSet()	This routine should be called for defining a manipulation node. A manipulation node must be defined before the CC node that precedes it.

Table continues on the next page...

Table 127. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
FM_PCD_IOC_MANIP_NODE_REPLACE[_COMPAT]	FM_PCD_ManipNodeReplace()	Change existing manipulation node to be according to new requirement.
FM_PCD_IOC_MANIP_NODE_DELETE[_COMPAT]	FM_PCD_ManipNodeDelete()	Deletes an existing manipulation node.
FM_PCD_IOC_SET_ADVANCED_OFFLOAD_SUPPORT	FM_PCD_SetAdvancedOffloadSupport()	This routine must be called in order to support the following features: IP-fragmentation, IP-reassembly, IPsec, header manipulation, frame replicator.
FM_PCD_IOC_FRM_REPLIC_GROUP_SET[_COMPAT]	FM_PCD_FrmReplicSetGroup()	Initialize a Frame Replicator group.
FM_PCD_IOC_FRM_REPLIC_GROUP_DELETE[_COMPAT]	FM_PCD_FrmReplicDeleteGroup()	Delete a Frame Replicator group.
FM_PCD_IOC_FRM_REPLIC_MEMBER_ADD[_COMPAT]	FM_PCD_FrmReplicAddMember()	Add the member in the index defined by the memberIndex.
FM_PCD_IOC_FRM_REPLIC_MEMBER_REMOVE[_COMPAT]	FM_PCD_FrmReplicRemoveMember()	Remove the member defined by the index from the relevant group.
FM_PCD_IOC_STATISTICS_SET_NODE[_COMPAT]	FM_PCD_StatisticsSetNode()	Not implemented in this release. Do not use!
FM_PCD_IOC_KG_SCHEME_GET_CNTR	FM_PCD_KgSchemeGetCounter()	Reads scheme packet counter.

NOTE

The _COMPAT variants of certain IOCTLs in the above table are required for supporting 32-bit user space apps. on 64-bit Linux kernels. The specifics of the COMPAT mappings are documented by Linux.

The programming model for defining and managing PCDs for a group of ports is the same as described in the **FMD LLD User's Guide** .

What follows is a step-by-step description of an example of ioctl() call mapping to a LLD API call.

The example chosen for this walk-through is that of FM_PCD_IOC_MATCH_TABLE_SET. Here's a reminder of the ioctl() prototype:

```
extern int ioctl (int __fd, unsigned long int __request, ...) __THROW;
```

and below is how it appears to kernel space:

```
struct file_operations {
 [...]
 long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
 [...]
};
```

The `ioctl()` function is actually a pointer to a driver-supplied function having the specified signature. The glue between the two is kernel code.

The semantics associated with the second and third function arguments are entirely the driver's business, but usually the `unsigned int` argument is used to discriminate between various ioctl commands (actually, it should obey some Linux good-behavior rules, which we are not going to detail here). In our case, it should be `FM_PCD_IOC_MATCH_TABLE_SET`.

Linux attaches no predefined semantics to the third argument, the `unsigned long` one. In some cases it is unused, or its semantics are those of an unsigned integer number, but in most cases it is treated as a (32-bit, on most platforms) pointer to a driver-defined structure in user space. The driver defines the format, but the user space allocates and fills in the data prior to invoking `ioctl()` on the open device `fd`. This is also the case with our example.

The format of the third argument of the `FM_PCD_IOC_MATCH_TABLE_SET` ioctl is (as it actually appears in the header file where it's defined):

```
/**************************************************************************//**
 @Description   A structure for defining the CC node params
*//***************************************************************************/
typedef struct ioc_fm_pcd_cc_node_params_t {
    ioc_fm_pcd_extract_entry_t extract_cc_params;
                                            /**< params which defines extraction
                                            parameters */

    ioc_keys_params_t          keys_params;    /**< params which defines Keys
                                            parameters of the extraction defined
                                            in extract_cc_params */

    void                       *id;            /**< output parameter;
                                            Returns the CC node Id to be used */
} ioc_fm_pcd_cc_node_params_t;
```

We'll detail the `ioc_*` types of the first two members later. The third member of this structure is apparently a pointer to some data structure being returned back to user space. It is not the case. This actual pointer should be handled as an opaque handle to some abstract item, in our case the "CC Node" that's being created for us by this `ioctl()` call if successful. This handle can be later passed to e.g. the `FM_PCD_IOC_MATCH_TABLE_DELETE` IOCTL for deletion. It corresponds to an actual `t_Handle`, as defined by the LLD.

NOTE

Failing to cleanup FMan resources that the LLD allocates in this manner can cause serious hardware resource leaks, which neither the Linux FMD, nor the LLD have the means to detect & cleanup automatically!

The LLD function that this IOCTL maps to has the following prototype:

```
t_Handle FM_PCD_MatchTableSet(t_Handle, t_FmPcdCcNodeParams *);
```

The first argument corresponds to the LLD resource that the Linux PCD device maps to. Most of the LLD resources are managed within the Linux FMD driver and not exposed to the user, but there are exceptions and the `FM_PCD_MatchTableSet()` function here is the best example, as it returns a `t_Handle` to such a LLD resource. This returned `t_Handle` is then passed over to the user space in the opaque `id` member of `ioctl()`'s third argument.

The second argument is a pointer to a structure of type `t_FmPcdCcNodeParams`. This maps to the `ioc_fm_pcd_cc_node_params_t` type that `ioctl()`'s third argument points to.

NOTE

Passing to `ioctl()` a pointer to something of a type other than the required one will cause the user application to segfault, or an error, at best, but may also cause undefined FMan behavior from that point onward, with errors being possibly reported only later downstream as the worst case. Linux/the FMD can do very little to prevent this worst case from occurring, so hopefully one can catch such coding errors early during the development cycle.

A side-by-side comparison of the two structures is given in the following table:

Table 128. Side-by-side comparison of IOCTL and LLD types

IOCTL Types	LLD Types

`typedef struct ioc_fm_pcd_cc_node_params_t {`
`  ioc_fm_pcd_extract_entry_t  extract_cc_params;`
`  ioc_keys_params_t  keys_params;`
`  void  *id;`
`} ioc_fm_pcd_cc_node_params_t;`
	`typedef struct t_FmPcdCcNodeParams {`
	`  t_FmPcdExtractEntry  extractCcParams;`
	`  t_KeysParams  keysParams;`
	
	`} t_FmPcdCcNodeParams;`

`typedef struct ioc_fm_pcd_extract_entry_t {`
`  ioc_fm_pcd_extract_type  type;`
`  union {`
`  struct {`
`  ioc_net_header_type  hdr;`
`  bool  ignore_protocol_validation;`
`  ioc_fm_pcd_hdr_index  hdr_index;`
`  ioc_fm_pcd_extract_by_hdr_type  type;`
`  union {`
`  ioc_fm_pcd_from_hdr_t  from_hdr;`
`  ioc_fm_pcd_from_field_t  from_field;`
`  ioc_fm_pcd_fields_u  full_field;`
`  } extract_by_hdr_type;`
`  } extract_by_hdr;`
 	
`  struct{`
`  ioc_fm_pcd_extract_from  src;`
`  ioc_fm_pcd_action  action;`
`  uint16_t  ic_indx_mask;`
`  uint8_t  offset;`
`  uint8_t  size;`
`  } extract_non_hdr;`
`  } extract_params;`
`} ioc_fm_pcd_extract_entry_t;`
	`typedef struct t_FmPcdExtractEntry {`
	`  e_FmPcdExtractType  type;`
	`  union {`
	`  struct {`
	`  e_NetHeaderType  hdr;`
	`  bool  ignoreProtocolValidation;`
	`  e_FmPcdHdrIndex  hdrIndex;`
	`  e_FmPcdExtractByHdrType type;`
	`  union {`
	`  t_FmPcdFromHdr  fromHdr;`
	`  t_FmPcdFromField  fromField;`
	`  t_FmPcdFields  fullField;`
	`  } extractByHdrType;`
	`  } extractByHdr;`
	
	`  struct {`
	`  e_FmPcdExtractFrom  src;`
	`  e_FmPcdAction  action;`
	`  uint16_t  icIndxMask;`
	`  uint8_t  offset;`
	`  uint8_t  size;`
	`  } extractNonHdr;`
	`  };`
	`} t_FmPcdExtractEntry;`

`typedef struct ioc_keys_params_t {`
`  uint16_t  max_num_of_keys;`
`  bool  mask_support;`
`  ioc_fm_pcd_cc_stats_mode  statistics_mode;`
`  uint16_t  num_of_keys;`
`  uint8_t  key_size;`
`  ioc_fm_pcd_cc_key_params_t`
`  key_params[IOC_FM_PCD_MAX_NUM_OF_KEYS];`
`  ioc_fm_pcd_cc_next_engine_params_t`
`  cc_next_engine_params_for_miss;`
`} ioc_keys_params_t;`
	`typedef struct t_KeysParams {`
	`  uint16_t  maxNumOfKeys;`
	`  bool  maskSupport;`
	`  ioc_fm_pcd_cc_stats_mode  statisticsMode;`
	`  uint16_t  numOfKeys;`
	`  uint8_t  keySize;`
	`  t_FmPcdCcKeyParams`
	`  keyParams[FM_PCD_MAX_NUM_OF_KEYS];`
	`  t_FmPcdCcNextEngineParams`
	`  ccNextEngineParamsForMiss;`
	`} t_KeysParams;`

While the structure members have resembling names on both sides, most are not identical. That's because style has prevailed over the need to port existing LLD applications to the Linux environment, when the Linux FMD was designed (there is a more complete porting guide included in the NCSW software bundle, though, that's not also included in this SDK; one may refer to that for the specific topic of porting existing NCSW apps. to various platforms). Except for the occasional `*id` pointer, there is a 1:1 mapping between the struct members on the two sides, and that is consistent throughout the FMD.

The constituent structures of the two APIs' argument types given above are for illustration only. Their semantics are documented in the Frame Manager Driver API Documentation .

NOTE

The existence of two separate definitions for otherwise two identical data structures may appear as an unfortunate design decision. However, since a *memcpy* from user space to kernel space is unavoidable, this design decision has no impact over performance. Moreover, the user space only sees one variant (i.e. the `ioc_*` one), hence the even smaller user impact. The larger impact is on code maintenance and on documentation.

7.4.4.1.6 The Linux Port Devices

There is a pair of RX/TX Linux character devices for each physical port of every Frame Manager. These devices are created irrespectively of the DPAA Ethernet network devices and they are strictly reflecting the available Frame Manager hardware on the given platform. The port Linux devices are labeled as follows:

- `/dev/fmX_port_rxY` for receive, where X=[0,1] represents the FMan number, and Y=[0-5] represents the physical port ID (0 corresponding to the first 1 Gb port, and 5 to the 10 Gb port), and
- `/dev/fmX_port_txY` correspondingly for the transmit side.

Each FMan also has a number of Offline Parsing ports. These are labeled as `/dev/fmX_port_ohY`, where Y=[0-6].

The port devices are created based on configuration information taken from the relevant Linux device tree section.

For instance, P5020 only has one FMan with 5 x 1Gb ports and one 10Gb port, while P4080 has two FMans, each having 4 x 1Gb and 1 x 10Gb ports. A side-by-side comparison of the corresponding port devices is given in the following table:

Table 129. Side-by-side comparison of port devices for P5020 and P4080.

P5020	P4080

For the Receive side:
`/dev/fm0_port_rx0`
`/dev/fm0_port_rx1`
`/dev/fm0_port_rx2`
`/dev/fm0_port_rx4`
`/dev/fm0_port_rx5`
	For the Receive side:
	`/dev/fm0_port_rx0`
	`/dev/fm0_port_rx1`
	`/dev/fm0_port_rx2`
	`/dev/fm0_port_rx3`
	`/dev/fm0_port_rx5`
	`/dev/fm1_port_rx0`
	`/dev/fm1_port_rx1`
	`/dev/fm1_port_rx2`
	`/dev/fm1_port_rx3`
	`/dev/fm1_port_rx5`

Table continues on the next page...

Table 129. Side-by-side comparison of port devices for P5020 and P4080. (continued)

P5020	P4080

For the Transmit side:
`/dev/fm0_port_tx0`
`/dev/fm0_port_tx1`
`/dev/fm0_port_tx2`
`/dev/fm0_port_tx3`
`/dev/fm0_port_tx4`
`/dev/fm0_port_tx5`
	For the Transmit side:
	`/dev/fm0_port_tx0`
	`/dev/fm0_port_tx1`
	`/dev/fm0_port_tx2`
	`/dev/fm0_port_tx3`
	`/dev/fm0_port_tx5`
	`/dev/fm1_port_tx0`
	`/dev/fm1_port_tx1`
	`/dev/fm1_port_tx2`
	`/dev/fm1_port_tx3`
	`/dev/fm1_port_tx5`

For Offline Parsing:
`/dev/fm0_port_oh0`
`/dev/fm0_port_oh1`
`/dev/fm0_port_oh2`
`/dev/fm0_port_oh3`
`/dev/fm0_port_oh4`
`/dev/fm0_port_oh5`
`/dev/fm0_port_oh6`
	For Offline Parsing:
	`/dev/fm0_port_oh0`
	`/dev/fm0_port_oh1`
	`/dev/fm0_port_oh2`
	`/dev/fm0_port_oh3`
	`/dev/fm0_port_oh4`
	`/dev/fm0_port_oh5`
	`/dev/fm0_port_oh6`
	`/dev/fm1_port_oh0`
	`/dev/fm1_port_oh1`
	`/dev/fm1_port_oh2`
	`/dev/fm1_port_oh3`
	`/dev/fm1_port_oh4`
	`/dev/fm1_port_oh5`
	`/dev/fm1_port_oh6`

NOTE

The `/dev/fmX_port_(rx4|tx4)` pairs of port devices are missing in the P4080, for numbering consistency reasons: this way the 10G ports will have the same IDs on every QorIQ platform that has them, thus greatly reducing the possibility for confusion, and adding to cross-platform code portability.

The table below summarizes the IOCTLs available for the port device.

Table 130. IOCTLs of the Port Device

IOCTLS	LLD Mapping	Brief
`FM_PORT_IOC_DISABLE`	`FM_PORT_Disable()`	Disables the port: all port settings are preserved, but all traffic stops.
`FM_PORT_IOC_ENABLE`	`FM_PORT_Enable()`	Enables the port: causes the port to start processing traffic.

Table continues on the next page...

Table 130. IOCTLs of the Port Device (continued)

IOCTLS	LLD Mapping	Brief
FM_PORT_IOC_SET_RATE_LIMIT	FM_PORT_SetRateLimit()	(TX & O/H Only) Activates the Rate Limiting Algorithm for the port.
FM_PORT_IOC_DELETE_RATE_LIMIT	FM_PORT_DeleteRateLimit()	(TX & O/H Only) Deactivates any Rate Limiting.
FM_PORT_IOC_SET_ERRORS_ROUTE	FM_PORT_SetErrorsRoute()	(RX & O/H Only) Instructs the FMD to enqueue frames w/specific errors onto the normal port queues, rather than onto the error queue (i.e. the default).
FM_PORT_IOC_ALLOC_PCD_FQIDS	N/A	For testing/debugging. Do not use!
FM_PORT_IOC_FREE_PCD_FQIDS	N/A	For testing/debugging. Do not use!
FM_PORT_IOC_SET_PCD[_COMPAT]	FM_PORT_SetPCD()	(RX & O/H Only) Defines a PCD configuration for the port.
FM_PORT_IOC_DELETE_PCD	FM_PORT_DeletePCD()	(RX & O/H Only) Deletes the port's PCD configuration.
FM_PORT_IOC_DETACH_PCD	FM_PORT_DetachPCD()	(RX & O/H Only) Disables the PCD configuration for the port (only allowed after FM_PORT_SetPCD() has been called for the port).
FM_PORT_IOC_ATTACH_PCD	FM_PORT_AttachPCD()	(RX & O/H Only) Re-enables the PCD configuration for the port (only valid after a call to FM_PORT_DetachPCD()).
FM_PORT_IOC_PCD_PLCR_ALLOC_PROFILES	FM_PORT_PcdPlcrAllocProfiles()	(RX & O/H Only) Allocates private policer profiles for the port (only allowed before a a call to FM_PORT_SetPCD()).
FM_PORT_IOC_PCD_PLCR_FREE_PROFILES	FM_PORT_PcdPlcrFreeProfiles()	(RX & O/H Only) Frees any private policer profiles allocated for the port (callable only before FM_PORT_SetPCD()).
FM_PORT_IOC_PCD_KG_MODIFY_INITIAL_SCHEME[_COMPAT]	FM_PORT_PcdKgModifyInitialScheme()	(RX & O/H Only) Modifies key generation scheme following frame parsing (callable only after FM_PORT_SetPCD()).
FM_PORT_IOC_PCD_PLCR_MODIFY_INITIAL_PROFILE[_COMPAT]	FM_PORT_PcdPlcrModifyInitialProfile()	(RX & O/H Only) Changes the initial policer profile for the port (callable only after FM_PORT_SetPCD()).

Table continues on the next page...

Table 130. IOCTLs of the Port Device (continued)

IOCTLS	LLD Mapping	Brief
FM_PORT_IOC_PCD_CC_MODIFY_TREE[_COMPAT]	FM_PORT_PcdCcModifyTree()	(RX & O/H Only) Replaces the coarse classification tree if one is used for the port (callable only after FM_PORT_DetachPCD() and before FM_PORT_AttachPCD()).
FM_PORT_IOC_PCD_KG_BIND_SCHEMES[_COMPAT]	FM_PORT_PcdKgBindSchemes()	(RX & O/H Only) Adds more KeyGen schemes for the port to be bound to (callable only after FM_PORT_SetPCD()).
FM_PORT_IOC_PCD_KG_UNBIND_SCHEMES[_COMPAT]	FM_PORT_PcdKgUnbindSchemes()	(RX & O/H Only) Prevents the port from using the specified KG schemes (callable only after FM_PORT_SetPCD())
FM_PORT_IOC_PCD_PRS_MODIFY_START_OFFSET	FM_PORT_PcdPrsModifyStartOffset()	(RX & O/H Only) Changes the frame offset at which parsing starts (callable only after FM_PORT_DetachPCD() and before FM_PORT_AttachPCD()).
FM_PORT_IOC_ADD_CONGESTION_GRPS	FM_PORT_AddCongestionGrps()	(RX & O/H Only) Should be called in order to enable pause frame transmission in case of congestion in one or more of the congestion groups relevant to this port. Each call to this routine may add one or more congestion groups to be considered relevant to this port.
FM_PORT_IOC_REMOVE_CONGESTION_GRPS	FM_PORT_RemoveCongestionGrps()	(RX & O/H Only) Should be called when congestion groups were defined for this port and are no longer relevant, or pause frames transmitting is not required on their behalf. Each call to this routine may remove one or more congestion groups to be considered relevant to this port.
FM_PORT_IOC_ADD_RX_HASH_MAC_ADDR	FM_MAC_AddHashMacAddr()	Add an Address to the hash table. This is for filter purpose only.
FM_PORT_IOC_REMOVE_RX_HASH_MAC_ADDR	FM_MAC_RemoveHashMacAddr()	Delete an Address to the hash table. This is for filter purpose only.
FM_PORT_IOC_SET_TX_PAUSE_FRAMES	FM_MAC_SetTxPauseFrames()	Enable/Disable transmission of Pause-Frames. The routine changes the default configuration: pause-time - [0xf000], threshold-time - [0]

Table continues on the next page...

Table 130. IOCTLs of the Port Device (continued)

IOCTLS	LLD Mapping	Brief
FM_PORT_IOC_GET_MAC_STATISTICS	FM_MAC_GetStatistics()	Get all MAC statistics counters.
FM_PORT_IOC_CONFIG_BUFFER_PREFIX_CONTENT	FM_PORT_ConfigBufferPrefixContent()	Defines the structure, size and content of the application buffer.
FM_PORT_IOC_VSP_ALLOC[_COMPAT]	FM_PORT_VSPAlloc()	This routine allocated VSPs per port and forces the port to work in VSP mode. Note that the port is initialized by default with the physical-storage-profile only.

NOTE

The COMPAT variants of certain IOCTLs in the above table are required for supporting 32-bit user space apps. on 64-bit Linux kernels. The specifics of the COMPAT mappings are documented by Linux.

The programming model for managing the FMan's ports is the same as described in the *Frame Manager Driver API Reference*. A few notable mentions though:

Although all the above IOCTLs are implemented by the Linux FMD, due to the asymmetry between RX and TX, not all are available for any port type. E.g. FM_PORT_IOC_SET_PCD will generate an error if called on a TX port device. Similarly, FM_PORT_IOC_SET_RATE_LIMIT will fail for an RX port. That is because the checking of the port type is being done late, inside the LLD, and not in the Linux FMD (i.e. the ioctl() calls for all port devices delegate to the same function inside the Linux kernel)!

The Offline Parsing ports have the best of both worlds. That is because conceptually, an O/H port is no different from a "regular" FMan port that has the TX side looped back internally to its RX side.

7.4.4.2 Frame Manager Linux Driver API Reference

This document describes the interface (IOCTLs) to the Frame Manager Linux Driver as apparent to user space Linux applications that need to use any of the Frame Manager's features. It describes the structure, concept, functionality, and high level API.

7.4.4.3 Frame Manager Driver User's Guide

7.4.4.3.1 Frame Manager Driver

7.4.4.3.1.1 Introduction

The Frame Manager is a hardware accelerator responsible for preprocessing and moving packets into and out of the datapath. It supports in-line/off-line packet parsing and initial classification to enable policing and flow/QoS based packet distribution to the CPUs for further processing of the packets.

The Frame Manager consists of a number of packet processing elements (also referred to as engines) and supports a flexible pipeline. Usually, the main Rx flow (simplified) follows these steps: packets are received from one of the Ethernet MACs, are temporarily stored in the FMan internal memory, then delivered to SoC memory via the FMan DMA. The packet header (max size 256 bytes) is stored and the modules common database structure is allocated. Then the packet is parsed by the parser or by the FMan controller. According to parsing results a key may be extracted by KeyGen, a destination frame-queue-id may be set, the packet may be classified by the FMan controller. in that stage, some offloads may be done like re-assembly, fragmentation, header-manipulation and frame-replication. At the end of the classification and manipulations stage, the packet may be colored by policer. At the end of this process, packets are delivered to SoC memory via the FMan DMA and then are enqueued to a frame queue or dropped. The processing order is Parse-Classify-Distribute (PCD) flow dependant, based on

user configurations. Each step is dependant on previous state results. This structure enables flexibility, which efficiently supports many flows.

On Tx the frames are transmitted via the desired MAC with optional checksum generation.

7.4.4.3.1.2 Frame Manager Features

The FMan driver aims to support the majority of the hardware features. It also includes exclusive software features designed to provides facilitation through abstraction.

Following are the features of the FMan driver:

- Simple initialization and configuration API for the following FMan blocks: DMA, FPM, IRAM, QMI, BMI, and RTC.
- Simple initialization and configuration for the following FMan PCD blocks: Parser, Keygen, Custom-Classifier (CC), Manipulations (e.g. Header-manipulations, IP-reassembly, IP-fragmentation, etc.) and Policer.
- FMan memory (MURAM) management.
- FMan-controller code loading.
- Software-Parser loading.
- Supported all FMan port types-Rx, Tx, Offline-Parsing, and Host-Command (internal use of the driver only)
- Common MAC API for dTSEC, 10G-MAC and mEMAC.
- Provides API for accessing the MII management interface.
- FMan Rx and Tx ports can run in one of the following modes:
  - Independent-Mode
  - Simple BMI-to-BMI (regular) mode
  - Advance PCD mode (using FMan PCD blocks such as parser, Keygen, CC, and Policer).
- FMan Offline ports can run in one of the following modes:
  - Simple BMI-to-BMI (regular) mode
  - Advance PCD mode (using FMan PCD blocks such as parser, Keygen, CC, and Policer)
- Internal (optional) Host-Command port initialization, based on user's parameters.
- FMan IRQ handling - events and exceptions.
- Supports both SMP and AMP operation modes.

7.4.4.3.1.3 Frame Manager Driver Components

The FMan driver contains following low-level modules, as shown in this figure.



Figure 89. FMan Driver Modules (from a partition point of view)

The modules are as follows:

- Frame Manager (common)-The FMan module is a singleton module within its partition. It is responsible for the common hardware modules: FPM, DMA, common QMI, common BMI, FMan controller's initialization, and runtime control routines. This module must always be initialized when working with any FMan module. The module will mainly be used internally by the other FMan modules except for its initialization by the user.

  This module has an instance for each partition. However, only the driver that is on the master-partition has access to the hardware registers.

- Frame Manager Parser-Classifier-Distributor (FMan-PCD)-The FMan PCD module is a singleton module within its partition. It is responsible of all common parts of the PCD, such as the hardware parser, software parser, Keygen, policer, and custom-classifier blocks. It is responsible for building the PCD graphs.

  This module has an instance for each partition. However, only the driver on the master-partition has access to the hardware registers.

- Frame Manager Memory (FMan-MURAM)-This module is responsible for the specific memory partition of the FMan Memory. Each partition may have its own FMan Memory partition that is managed by the FMan Memory driver. For example, an FMan Memory instance will be created for each partition that has its own FMan ports.

  This module has an instance for each partition.

- Frame Manager Real-Time-Clock (FMan-RTC)-This module is responsible for the FMan RTC module.

  This module is a "singleton" and should be created once only for the master-partition.

- Frame Manger Port (FMan-Port)-This module is responsible for all FMan port-related register space, such as all registers related to a port in QMI or BMI.

  This module can be run by each core or partition independently.

- Frame Manager MAC (FMan-MAC)-This module is responsible for the mEMAC dTSEC and the 10G MAC controllers.

  This module can be run by each core or partition independently.

- Frame Manager Virtual-Storage-Profile (FMan-VSP)-This module is responsible for allocating and managing virtual storage profiles that may used for virtualization purposes. More of the VSP is described in FMan VSP Driver on page 763.

  This module can be run by each core or partition independently.

7.4.4.3.1.4 Driver Modules in the System

The FMan driver is designed to support single or multi partition environment. In addition, the FMan driver is designed to support environment with multicore that are running in SMP mode.

The following figure shows a typical single-partition (maybe SMP or not) environment and its FMan driver building blocks.

NOTE

In this environment:

- All FMan driver modules are available and should be initialized by the user (unless if it is unnecessary for the user operation; for example, if PCD is not needed so it may not be called).
- The FMan driver modules have the full functionality of the hardware.
- Each module has full access to its hardware registers (i.e. each module will access its registers directly).



Figure 90. Single-Partition FM Building Blocks

7.4.4.3.1.4.1 Multicore Approach

The driver supports both Symmetric Multi-Processing (SMP) and Asymmetric Multi-Processing (AMP) operation methods.

7.4.4.3.1.4.1.1 SMP

As a rule, driver routines are not SMP safe. It is user's responsibility to lock all routines that might be in risk in his environment, for example, if `FM_PORT_Enable`/`FM_PORT_Disable` may be used by several cores, it is user's responsibility to protect the routine call using a spinlock.

An exception to this rule is the set of PCD routines. Due to the complexity of this module, and in order to support SMP and maintain coherency, PCD routines are protected using two mechanisms, spinlocks and flags.

Each PCD resource (i.e. software module such as scheme, CC Node, NetEnv, etc.) may have one or more spinlocks which are used to protect short code sections where specific resources such as hardware registers or software structures are accessed. In some cases, a spinlock of a higher level is used (i.e. CC locks the whole PCD).

The second mechanism is defined globally. The PCD global module provides a PcdLock mechanism, which is a list of lock objects containing a flag and a spinlock rotating that flag. On initialization of each PCD resource (i.e. software module such as scheme, CC Node, NetEnv, etc.), a PcdLock is allocated for this module. Critical sections that may not be protected by spinlocks (due to reasons of sections length, Host Commands and other lengthy operations) are protected by these flags. Note that this is a try-lock mechanism and the calling routine returns with `E_BUSY` error on failure. The try-locks are used by all PCD resources modification routines, in which case the application is expected to recall the routine until it is not busy.

In Addition, PCD and FM Port inter-module complex sections may be protected by try-locking all the initialized PcdLock modules in the global PCD, thus providing a safe PCD environment where influence and connections between modules may take effect.

On top of PCD routines, all FM Port PCD related routines are also protected by Port try-lock, meaning no two cores can access the same port to run a PCD routine. As in the PCD routines, these routines may return `E_BUSY` on failure and should then be recalled.

The driver SMP protection mechanism assumes the following:

- Only one core may initialize and delete a specific PCD software module (i.e. scheme x may not be initialized by two cores).
- A core should not attempt to delete a PCD software module when there is a risk of another core operating on that specific module.

7.4.4.3.1.4.1.2 AMP

FMan driver supports multi-partitioned system in a way that the common modules (`FM, FM_PCD`) are designed as "front-end" and "back-end". When configuring the FM driver module, the user should pass the driver its "guestId" ('0' identifies the "back-end"/"master-partition").

NOTE

In order to support AMP, FMan driver uses IPC calls for synchronization between the master module and the guests, i.e. user must have some IPC available in his system in order to have this functionality available in FMan driver.

There is no "hard" relation between "partitions" and "guests". A number of guests may co-exist on the same physical partition, creating a logical partitioning. Note that the FM "master"/"back-end" may run on any partition in the system.

When the FM driver is configured to run as a guest, it may access registers only if there is no race condition. If register access may end with some risk, the FM "front-end" driver access the HW registers through IPC call to the "back-end".

In the driver implementation, the FM-Port and the FM-MAC drivers are indifferent to guest/master considerations and may run on either partition. They do not have a "front-end" and both initialization and runtime routines are locally implemented. This means that a port may not be controlled from several partitions. However, a few ports may be controlled by the same partition.

The FM-MURAM module is pre-allocated and managed independently by each guest that has MURAM consumers (typically for Custom Classifier tables). It does not have a "front-end" and once allocated it may be accessed directly.

The FM-RTC module is a singleton module and is available only on the "back-end"/"master-partition" and therefore has no "front-end".

In the figure below, we can see two partition (A and B), where the FMan driver is configured as a master (the "back-end") on partition A and it is configured as a guest ("front-end") on partition B. In addition, each partition has its own FM-ports (also FM-MACs) that are working with the "front-end" and host-commands.



Figure 91. Multi-Partition FM Building Blocks with IPC (DPAA 1.0)



Figure 92. Multi-Partition FM Building Blocks with IPC (DPAA 1.1)

There may be two ways for user to work in AMP:

- AMP with IPC-In systems that has IPC mechanism (as in the figure above), the FM drivers (including FM, FM-PCD, FM-Port, etc.) will utilize the IPC mechanism to actually synchronize information between them; i.e. FM-PCD driver in guest mode will call the FM-PCD master to allocate schemes for its partition (according to user configuration). In this system, it is possible to create FM-Ports and FM-MACs local for guest partition; i.e. the HW of FM-Port and FM-MAC will be accessed and controlled locally for the partition that actually owns the resource.
- AMP without IPC-The FM driver itself doesn't support this kind of system. As shown in the figure below, it is still possible for a user to create DPA-Ports and to send and receive packets to and from MAC. In addition, user may also initialize

and access MACs from guest partitions, but, other operations like creating PCDs or accessing the FM-Ports registers may be done only by the master partition.

NOTE

Although FM-MAC driver can be initialized and accessed from guest partition even if there is no IPC present, some functions may be disabled (e.g. events of the MAC).



Figure 93. Multi-Partition FM Building Blocks without IPC (DPAA 1.0)



Figure 94. Multi-Partition FM Building Blocks without IPC (DPAA 1.1)

7.4.4.3.1.5 FMan Driver Calling Sequence

Initialization of the FMan driver is carried out by the application according to the following sequence:

1. MURAM configuration & Initialization
2. FMan (common) configuration & Initialization

3. [Optional] FMan RTC configuration & Initialization
4. For each MAC required by the user:
   a. MAC Configuration & Initialization
   b. PHY Initialization
5. For each FMan Port required by the user:
   a. FMan Port Configuration & Initialization
   b. [optional] If the FMan Port required to be virtualized, a set of VSPs need to be allocated and one of them should be set as the default.
   c. [optional] If VSPs were allocated in previous step, the default VSP need to be configured & initialized
   d. in that stage, user should configure and intialize everything that is needed for the operation of a port outside the fman; e.g. buffer-pools, frame-queues, etc.
   e. Port Enablement
   f. MAC Enablement
   g. Calling 'AdjustLink' MAC API routine with the relevant link parameters

NOTE

Now, the FMan is operational. The ports operate in independent mode or BMI-to-BMI mode. From that point, all the following steps are optional.

6. FMan PCD Configuration & Initialization
7. If a physical port is being "vitualized" into several software entities (using some classification to ditribute the traffic), user should configure and initialize the relevant buffer-pools and frame-queues.
8. If VSP is enabled, in that stage, user should configure and initialize the relevant profile.
9. FMan PCD Graph initialization:
   a. Calling restricted runtime routines (that may be called only when PCD is disabled)
   b. Calling the PCD enable routine
   c. Initialization of a all PCD Graph objects (i.e. KG-schemes, Match-Tables, etc)
10. FMan port-PCD related initialization; calling the run-time control routines to set the PCD related parameters

NOTE

In case the PCD is "set" to a FMan OP port, it should be disabled first (i.e. before calling 'FM_PORT_SetPCD' routine).

11. FMan runtime routines
12. FMan Free sequence - in reverse order from initialization

7.4.4.3.1.6 Global FMan Driver

The Global FMan driver refers to the common FMan features - i.e. functionality that is not defined per-port and does not belong to a spany of the specific modules such as PCD, RTC, MURAM, MAC etc.

7.4.4.3.1.6.1 FMan Hardware Overview

The following Frame Manager processing elements are considered general FMan components and are controlled by the FMan common driver:

- The Frame Processor Manager (FPM) schedules frames for processing by the different elements to create the appropriate pipeline.

- The BMI is intended to transfer data between network and internal FMan memory, generate frame descriptor (FD), initialize the internal context (IC), manage the internal buffers, allocate/deallocate external buffers with the help of BMan and activate the DMA to transfer data between internal and external RAMs
- The DMA is responsible for frames data transfer from and to external memory
- The queue manager interface (QMI) is responsible for transferring packet-based work assignments between the queue manager (QMan) and the frame manager (FMan). It provides an interface to the QMan for enqueuing and dequeuing new frames to/from the multicore system.

7.4.4.3.1.6.1.1 Global FMan Driver Software Abstraction

The FMan global driver covers all the logically common FMan functionality, i.e functionality which is not port related. The different hardware modules within the FMan (i.e. BMI, DMA, etc.) are encapsulated within the FMan module. The terms "BMI", "DMA" are used for resources identification such as exceptions, counters and some configuration parameters, but logically, the only module used for functional operations is the FMan.

7.4.4.3.1.6.2 How to use the Global FMan Driver?

The following sections provide practical information for using the software drivers.

7.4.4.3.1.6.2.1 Global FMan Driver Scope

This module represents the common parts of the FMan. It includes:

- FMan hardware structures configuration and enablement
- Resource allocation and management
- Interrupt handling
- Statistics support
- ECC support for the FMan RAM's
- Load balancing between ports

7.4.4.3.1.6.2.2 Global FMan Driver Sequence

- FMan config routine
- [Optional] FMan advance configuration routines
- FMan Init routine
- FMan runtime routines
- FMan free routine

7.4.4.3.1.6.2.3 Global FMan Driver Functional Description

The following sections describe main driver functionalities and their usage.

7.4.4.3.1.6.2.3.1 FMan Configuration and Initialization

On FMan driver initialization, the software configures all FMan registers and relevant memory. It supplies default values where no other values are specified, it allocates MURAM, it loads FMan controller code. It defines IRQ's and sets IRQ handles. It enables hardware mechanisms and initializes software data structures for software management.

By the time initialization is done, FMan is ready to be used and any of the FMan sub-modules (FMan-Ports, MACs, etc.) may be initialized.

7.4.4.3.1.6.2.3.2 Resource Management & Tuning

The FMan provides resources used by its sub-modules. Generally, the driver selects default resource allocation, but when initializing the global FMan module, the user may specify a different allocation for some or all of the resources.

The resources relevant for this discussion are resources used by the BMI only. These resources should be further distributed between the different ports, but the initial allocation is for the BMI in opposed to some internal use of the FMan controller. The main and most important resources of the FMan are TNUMs (i.e. the FMan "tasks"), DMAs, FIFOs and "pipeline-depth".

The total available resources may vary based on SoC. The recommended default values are designed to fit most applications but as the resource allocation depends on system configuration, it therefore may vary between applications. I.e. the default value that are being set by the driver will be sufficient in use-cases were the user utilizing most of the FMan bandwidth and the user application is mostly using the FMan. In other cases such as if user uses some advance PCD settings and/or overloads the SoC (e.g. PCI is being massively used), the resources may need some special treatment and tuning by user as the default may not be sufficient enough.

Most MURAM is used as a temporary location for data transaction. This part's size is referred to as "FIFO size". The rest of the MURAM may be used for other utilizations such as Custom Classifier and its size is effected by the use of these features, i.e. if Custom Classifier is not used, "FIFO size" may be enlarged. The user may call `FM_ConfigTotalFifoSize` in order to modify the default value of the MURAM. However, one should bear in mind that when FIFO size is enlarged - Custom Classifier space is decreased.

7.4.4.3.1.6.2.3.3 Load Balancing

The FMan provides a mechanism to optimize the internal arbitration of different ports over the shared resources of the hardware.

The driver supports this feature by providing an API for dividing the bandwidth between the different ports (`FM_SetPortsBandwidth`). The API is given in terms of percentage - i.e. for each port, the user should specify its percentage relative to the other ports. This API is optional and may be modified at runtime. If not used, or if all ports get the same bandwidth (whether its {50,50} or (25,25,25,25}), then no one port will have priority over other ports. If ports get different values, for example 3 ports used and get {25,50,25}, than the first and third ports will get the same access to shared resources but the second one will get twice as much. i.e. The numerical values given to each port are not important, but only the relation between the ports.

7.4.4.3.1.6.2.3.4 Statistics

The FMan API provides access to all the statistics gathered by the FMan hardware. The API routine `FM_GetCounter` may be called at any time after initialization to retrieve any of the FMan counters.

7.4.4.3.1.7 FMan Parse-Classify-Distribute Driver

The Parse-Classify-Distribute (PCD) driver module refers to the parts of the drivers handling the different PCD engines and services such as Parser, Keygen, Custom Classifier, Policer, Header Manipulation, Reassembly, Fragmentation and Frame Replication. It deals both with the common configuration and runtime features and the specific PCD resources such as Keygen Schemes, Custom Classifier graphs, etc.

7.4.4.3.1.7.1 FMan PCD Hardware Overview

- Parser-The parser performs protocol header parsing and validation for a wide range of frame formats with varying protocols and encapsulation. A hard-coded parser function is used for the known and stable protocols. The hardware parser capabilities can be expanded by software parser functions to support protocols not supported by the hardware parser including proprietary protocols and shim headers. The parser parses the frame according to a per-port configuration. It reads the frame header from the FMan Memory and writes the frame parse results to the Internal Context of the frame. The Lineup Confirmation Vector is a part of the parser result. It represents a list of all the protocols recognized by the hardware parser, and may be extended to contain information added by the software parser.
- Keygen-The Keygen is located on the FMan receive path, and enables high performance implementation of pre-classification. It holds a SoC dependent number of key generation schemes in internal memory. Each scheme can generate different frame queue ID (FQID), a Storage-Profile ID (SPID) and policer profile (PP). One main function of the Keygen module is to separate network data into different flows, each requiring different processing. Another function of the Keygen, is the Classification Plan. This is a mechanism provided in order to mask LCV bits according to per-port definition. The Classification Plan is implemented as a table of SoC dependent number of entries, logically divided or shared between the FMan Ports.

- Custom Classifier-The Frame Manager (FMan) Custom Classifier module performs a look-up using a specific key from the received frame or internal frame context according to Parser results. The FMan Custom Classifier logically occurs after the Keygen processing has completed and can be operational in both the MAC receive flow and the offline parsing flow. The look-up produces an action descriptor which contains the necessary information for the continuation of the frame processing in the next module or the next look-up table.
- Policer-The Policer supports implementation of differentiated services at line speed on the Frame Manager (FMan) receive or offline parsing paths. It holds a SoC dependent number of traffic profiles in internal memory, each profile implementing RFC-2698 or RFC-4115 or Pass-Through mode. Each mode can work in either color-blind or color aware mode, and pass or drop packets according to their resulting color.

7.4.4.3.1.7.1.1 FMan PCD Software Abstraction

The FMan PCD driver aims to provide a high-level, abstract, network oriented, logical interface. It is designed to allow a glue logic between the different PCD engines and the PCD "user" - the FMan port, and to define an interface to these features to be used by the application. In this process, new non-hardware modules may be created - such as "Network Environment", while existing hardware modules - such as "Classification Plan" - may be hidden from the user. The following sections makes an attempt to describe the driver design decisions in abstracting the engines' hardware and the gap between the hardware programming model and the drivers API.

7.4.4.3.1.7.1.1.1 FMan PCD Flow

The FMan opens the FPM scheduling capabilities to the application, which allows significant flexibility in defining the packet flow. At various points in the flow, the FMan user must configure the next engine to handle the packet and the next operation it will perform. The driver minimizes this flexibility by assuming a basic flow for each port. The driver can expand this flow to include all FMan PCD capabilities, but in a limited manner that will be described below.

The basic flow reflects the expected use of the FMan PCD. When a port is initialized, the default setup that received packets are passed to the port's default Rx frame queue, as configured by the user. When the PCD is linked to the port, the user chooses one of the provided PCD support options which selects which PCD engines (parser, Keygen, FMan-Controller, and Policer) are included in the frames. The selected PCD support option adds the selected engine or engines to the flow according to the following PCD organization.

- When parser is used, it is always the first PCD engine working on the received frames.
- If parser is not activated, Keygen, and FMan-Controller may not be activated.
- Keygen's first use follows the parser, but it may be used again following the Fman-Controller or the policer.
- If FMan-Controller is used, it will follow the Keygen. It may not be activated if Keygen is not used.
- Policer may be activated by itself or follow any of the engines.

In all cases, the frame returns to the buffer manager interface (BMI) for enqueueing. The application may not change the main flow at runtime.

The following figure shows the default ports flows (in terms of next invoked action (NIA) registers' initialization):



Figure 95. Default Rx Flow



Figure 96. Default Offline Parsing Flow

NOTE

In independent mode, both Tx and Rx BMI NIA are FMan Controller. Other NIAs are not applicable.

After basic initialization, the default Rx flow, as shown in Figure 95. Default Rx Flow on page 729, is the configured flow. A PCD flow is initially defined by FMan Port level, although it is effected both by the port configuration and the PCD resources configuration. Following figure shows the PCD flows supported by the driver.



Figure 97. Available flows

7.4.4.3.1.7.1.1.2 Global FMan PCD Module

The FMan PCD driver deals with the configuration initialization and runtime setting of the PCD resources. The actual use of these resources is in fact activated only when an FMan-Port is enabled and is bound to the initialized PCD resources. In this chapter we will only deal with the initialization and organization of those resources.

The PCD driver is constructed by a global FMan-PCD module that must be initialized first, and a set of optional PCD resources that can be initialized at run-time. The FMan-PCD module is responsible for enabling the different engines, loading SW parser if required, registering PCD interrupts and other general configuration.

7.4.4.3.1.7.1.1.3 Global FMan-PCD Resources

PCD driver's resources are NOT identical to PCD hardware resources and provide an abstraction layer to the hardware resources. PCD is viewed as a graph of PCD resources where FMan RX & OP Ports may be bound to subsets of the PCD graph. Refer to Port-PCD Binding on page 757.

The following are the driver's PCD resources:

- Network Environment Characteristics
- Software Parser
- Keygen Schemes

- Custom Classifier Roots
- Custom Classifier Match-Tables
- Custom Classifier Hahs-Tables
- Custom Classifier Manipulations
- Policer Profiles

The **Network Environment (NetEnv) Characteristics** are a pure SW resource. It is used in creating multiple HW PCD resources. Logically, it represents the NetEnv of a port or a number of port and supplies the glue between the parser, the Keygen, the Custom Classifier and the port. It ensures they all "speak the same language". Physically, it defines the LCV for all the participating protocols for each FMan Port.

Keygen Schemes and **Policer Profiles** are closely bound to their hardware programming model

Custom Classifier process is represented by a software graph. Each node in the graph represents a logical action. The driver defines different types of Custom Classifier nodes. One type of node is one of an Exact-Match which is a software representation of an Action-Descriptor (AD) that performs a lookup according to the key defined. Another type of node is one of Indexed-Lookup which is again a software representation of an Action-Descriptor of that type. A higher level of abstraction is performed on Hash-Table nodes, where the driver manages a hash table. Each node, may also contain a handle to a Manipulation action - which is the software abstraction for one or more AD's used for manipulating the frame by inserting and/or removing data. Generally, any Custom Classifier software node may be translated to one or more HW action descriptors.

The driver defines a notion of a Custom Classifier graph. The CC graph is the total set of lookups and manipulations performed by the Custom Classifier. The user builds the graph only after defining the CC Nodes. The finalization of the graph is done by building the root nodes and defining their grouping. This refers to the 16 entries array that functions as the entry point of the CC. Generally, the indexing into this array is performed by using 4 bits out of the LCV. This driver supports a division of this array into 2-16 unrelated groups to increase the flexibility of the programming and allow usage of more LCV bits.

7.4.4.3.1.7.1.1.4 How to Associate PCD Resources

The NetEnv is the link between the port and all the PCD resources it is using.

- Parser-The driver configures the LCV (lineup confirmation vector) in the parser configuration for every FMan Port according to the specific NetEnv it is bound to. When using SW parser, a private shim header should be added as a NetEnv unit, and may be used later as a regular unit.
- Keygen-Classification plan: The driver hides this resource from the user and configures classification plan entries to support and expand the HW parser capabilities according to the user definition of its NetEnv Characteristics
- Keygen-Schemes: The user describes the scheme in terms of NetEnv units, and the match vector is configured by the driver.
- Custom Classifier: The user describes the entry point of a CC root in terms of NetEnv units. The driver internally passes this information to the Keygen that uses it in selecting the entry point in the CC root when passing a frame from the Keygen to the Custom Classifier.

After defining PCD resources, the user may bind any FM Port to the initialized resources. A port must be bound to a single NetEnv, and may be bound to a Custom Classifier root and KeyGen schemes.

The set of figures below demonstrate a single example of the use of the driver's resources and their interaction with the hardware structures.

The following table demonstrates a NetEnv of 7 units. Unit 0, for example, is a simple unit recognizing ethernet frame, while unit 2 recognizes IP frames of either version.

Unit 0	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
Ethernet	Ethernet [Broadcast]	IPV4	IPV4	UDP	MPLS [stacked]	IPV4 [Muticast]
		IPV6		TCP		

Figure 98. Network Environment Example

When a port is bound to a NetEnv, the driver translates its units into the parser's hardware Line-up Confirmation Vector (LCV). The table below shows the LCV configured for a port that has the NetEnv above.

	LCV[0]	LCV[1]	LCV[2]	LCV[3]	LCV[4]	LCV[5]	LCV[6]	LCV[7-31]
Ethernet	1	1	0	0	0	0	0	0...0
IPv4	0	0	1	1	0	0	1	0...0
IPv6	0	0	1	0	0	0	0	0...0
UDP	0	0	0	0	1	0	0	0...0
TCP	0	0	0	0	1	0	0	0...0
MPLS	0	0	0	0	0	1	0	0...0

Figure 99. LCV Example (for a specific port that is bound to the Network environment above)

Based on the NetEnv, the driver also defines a set of Classification Plan entries to be used by each port using that NetEnv.

	Bit [0]	Bit [1]	Bit [2]	Bit [3]	Bit [4]	Bit [5]	Bit [6]	Bits [7-31]	Comments
0	1	0	1	1	1	0	0	1...1	No cls. Plan
1	1	1	1	1	1	0	0	1...1	Eth Broadcast
2	1	0	1	1	1	1	0	1...1	MPLS Stacked
3	1	1	1	1	1	1	0	1...1	1+2
4	1	0	1	1	1	0	1	1...1	IPv4 MC
5	1	1	1	1	1	0	1	1...1	1+4
6	1	0	1	1	1	1	1	1...1	2+4
7	1	1	1	1	1	1	1	1...1	1+2+4

Figure 100. Classification Plan entries for the Network environment above

When a frame is received its LCV is masked by one of the vectors in the Classification Plan. The FMan selects the entry based on the parser output and the port parameters.

To support this operation, the driver initializes the HXS plan offset field for each relevant header in the port parser parameters. The table below, is the driver's translation of the Network environment above into the port classification plan parameters. When a frame is being parsed, the classification plan offset for each header found is accumulated to construct the offset of the result classification plan. For example, a hypothetic frame of Ethernet BC/Stacked MPLS/IPv4 unicast frame, will have an LCV=0xF6000000 and a classification plan id of 2^(1-1)+ 2^(2-1) = 3, so its classification plan vector is 0xFDFFFFFF, and QLCV = 0xF4000000.

	HXS [plan offset]	Cls plan entry
Eth. BroadCast	1	2^(1-1)=1
MPLS Stacked	2	2^(2-1)=2
IPv6	0	0
UDP	-	-
TCP	-	-
IPv4 Multicast	3	2^(3-1)=4

Figure 101. Parser HXS (for a port that is bound to the Network environment above)

Given the driver's automatic initialization of the LCV and classification plan based on only the NetEnv, the user may now initialize Keygen schemes by passing as match criteria only the NetEnv unit id's. As in the other cases, the driver will translate the unit id's to the schemes' match vectors as can be seen in the figure below.

Id	Scheme Match Criteria	Units	Match vector
0	Ethernet broadcast	1	0x40000000
1	IPV4 MC+MPLS stacked	5+6	0x06000000
2	IPV4 MC	6	0x02000000
3	IPV4+(TCP or UDP)	3+4	0x18000000
4	match on IPv4 or IPv6 frames	2	0x20000000
5	Ethernet	0	0x80000000
6	Direct scheme	--	0xffffffff

Figure 102. Keygen schemes example

Finally, the driver will also take care of initializing the Keygen-to-Custom Classifier configuration registers. When initializing a Custom Classifier root, the user may create groups based on NetEnv units (in opposed to a simple group of a single entry; for more information refer to Custom Classifier Root on page 739).

When initializing a scheme, the user should only pass the handle to the Custom Classifier root. The driver will translate the group LCV dependent parameters into the scheme required register.

For example, Group 0 is a simple group that is not dependent on the NetEnv. Group 1 is based on a single unit - so a frame may be forwarded to 1 of 2 root nodes, and group 2 is based on 3 units - so a frame may be forwarded to 1 of 8 root nodes.

	CC Tree group Num of units	units	Keygen FMKG_SE_CCBS	Possible offsets within group depending on PR[LCV] AND FMKG_SE_CCBS
Group 0	0	--	0x00000000	0
Group 1	1	3	0x10000000 (Scheme 4 in the example)	0,1
Group 2	3	1,3,4	0x58000000	0-7

Figure 103. Keygen scheme configuration for CC next engine

The Policer Profiles are the one resource that does not rely on the Parser Results or the NetEnv. It is therefore managed independent of the other PCD resources.

7.4.4.3.1.7.1.1.5 FMan Header Manipulation

The FMan controller defines a set of header manipulation commands, and supports listing of these commands. The FMan driver allows limited listing by a single Manipulation node, limited to a single use of each command and to a defined order (e.g. remove + insert may be defined in a single node, but insert + remove or remove + remove may not). Alternatively, full listing and ordering is supported by chaining more than one Manipulation nodes. In such a case, the driver will unify HMCT's to optimize performance and MURAM usage unless parsing is required in between the different commands.

The following list maps each FMan controller command to the driver parameters in the Header Manipulation structure:

1. Generic removal-Set 'rmv' and use the corresponding parameters structure. Select generic enum and parameters.
2. Generic insertion-Set 'insrt' and use the corresponding parameters structure. Select generic enum and parameters.
3. Generic replace-Set 'insrt' and use the corresponding parameters structure. Select generic enum and parameters and set 'replace'.
4. Protocol specific removal-Set 'rmv' and use the corresponding parameters structure. Select byHdr enum and parameters.
5. Protocol specific insert-Set 'insrt' and use the corresponding parameters structure. Select byHdr enum and parameters.
6. Vlan priority update-Set 'fieldUpdate' and use the corresponding parameters structure. Select vlan enum and parameters.
7. IPv4 update-Set 'fieldUpdate' and use the corresponding parameters structure. Select IPv4 enum and parameters.
8. IPv6 update-Set 'fieldUpdate' and use the corresponding parameters structure. Select IPv6 enum and parameters.
9. TCP/UDP update-Set 'fieldUpdate' and use the corresponding parameters structure. Select TCP/UDP enum and parameters.
10. TCP/UDP checksum calculation-Set 'fieldUpdate' and use the corresponding parameters structure. Select TCP/UDP enum and parameters.
11. IP replace-Set 'custom' and use the corresponding parameters structure. Select TCP/UDP enum and parameters.

7.4.4.3.1.7.1.1.6 Custom Classifier Hash-Table Node

The driver provides a high level Hash-Table mechanism implemented over the FMan controller Custom Classifier structures. The driver implements the Hash-Table by using a Match-Table node of type Indexed-Hash, where each entry points to a hash bucket implemented by a Match-Table node of type Exact-Match (For more information on these nodes, refer to Custom Classifier Root on page 739). The driver uses the Keygen key and hash result as a key for the lookup. A selected part of the hash result is used to select the entry in the Indexed-Hash table (i.e. the bucket), and the full key possible values are used as the Match-Table keys in the selected bucket.



Figure 104. Hash_Table node example

7.4.4.3.1.7.2 How to use the FMan PCD Driver?

The following sections provide practical information for using the software drivers.

7.4.4.3.1.7.2.1 FMan PCD Driver Scope

- FMan Parser, Keygen, Custom Classifier & Policer configuration and initialization
- PCD Enable/Disable
- Resources allocation and management
- Interrupt handling
- Statistics support
- Support for FMan PCD operations

7.4.4.3.1.7.2.2 FMan PCD Driver Sequence

- FMan PCD Config routine
- [Optional] FMan PCD advance configuration routines
- FMan PCD Init routine
- Specific one-time pre-enable routines (e.g. load SW parser)
- FMan PCD Enable routine
- FMan PCD runtime routines
- FMan PCD specific resources runtime routines (for defining, modifying and deleting Keygen schemes, Custom Classifier nodes, etc.)
- FMan PCD Free routine

7.4.4.3.1.7.2.3 FMan PCD Driver Functional Description

The following sections describe main driver functionalities and their usage.

7.4.4.3.1.7.2.3.1 Global PCD Initialization

PCD initialization is divided into two parts. During the first part of the initialization, `FM_PCD_Config`, advance config routines, and `FM_PCD_Init` are called to configure and set all basic PCD capabilities, including pre-defining which engines are supported and may be used later. This stage is done in the kernel, and PCD is not yet enabled. During the second part of the initialization, PCD is enabled by a runtime routine (`FM_PCD_Enable`).

This division creates a gap during which some functionality may be added. The most important is the loading of the SW parser code. Note that this functionality is allowed only when PCD is disabled (i.e. between init and enable) or, with some restriction, in runtime after disable.

Once PCD basic initialization is complete (`FM_PCD_Init` and `FM_PCD_Enable` are called and returned), the PCD capabilities of the frame manager are reflected by the driver as a set of API runtime routines designed to define the PCD environment for a specific partition. PCD resources are defined per partition and may be used by all ports within a specific partition. The different PCD resources are first initialized and only later may be used by the FMan ports.

The order of PCD resources initialization is strict and relies on the PCD graph being initialized bottom up, which means that no resource may be initialized before its next engine is initialized. However, the use of port relative profiles is an exception to this rule. A scheme's next engine may be a port relative profile. In such a case, the scheme is initialized but not yet bound to a port, i.e. the actual policer profile is not yet specified. Therefore, its validity may not be verified. It is the user's responsibility to ensure that when a port using that scheme is activated (for using the PCD), its relative policer profile must be validated.

The PCD graph is partition based i.e. may be shared by ports on the same partition. Refer to Port-PCD Binding on page 757 for more details on port-PCD binding.

7.4.4.3.1.7.2.3.2 PCD Resources

The following subsections describe each of the driver's PCD resources in detail. In a single-partition environment, most resources are available and do not need explicit allocation. The port policer profiles are an exception. They must be allocated by the user, using the FMan Port API. In multipartition, some of the resources, specifically resources limited by hardware, must be first allocated by a partition and only then used by the partition's ports. The following sections specify the requirements for each of the PCD resources:

7.4.4.3.1.7.2.3.3 **Network Environment Characteristics**

The Network Environment (NetEnv) is a software entity that lists the network protocols used by the FM-PCD for classification and distribution. The total number of NetEnvs defined depends on the system configuration. A NetEnv may be defined per port or shared among some or all ports. The definition of a NetEnv must be done with care while considering the use of the FM-PCD module. The NetEnv is, in fact, the key for frames parsing, distribution, and classification.

The NetEnv is a list of distinction units. Each distinction unit consists of at least one or more headers. A header may either be one header from the list of supported headers or one of the supported headers plus an option (For more details on list and options available, refer to Supported Network Protocols on page 766).

The hardware parser implements header recognition. If the software parser is used, a distinction unit may also be one of the shim headers. The driver saves a number of units (that may be redefined in `fm_pcd_ext.h`) for private use. The user may than use this unit ID to recognize the private header by the Keygen or CC.

The following figure shows an example of a NetEnv. It has four units, two of which consist of a single header. One of the headers has an option. The final two units consist of two interchangeable headers. This example will be used throughout the following sections

.

Ethernet [Broadcast]	IPv4	IPv4	TCP	
	IPv6		UDP	

Figure 105. Network Environment Example

The distinction units list should reflect what the user wants to do with the PCD mechanisms to parse-classify-distribute incoming frames. Specifying a distinction unit means that the user wants to use that specification to either activate the parser on the specified headers or distinguish between frames with the Keygen or the Custom Classifier. Using interchangeable headers to define a unit means that the user is indifferent to which of the interchangeable headers is present in the frame, but instead wants the distinction to be based on the presence of either one of them. For example, if it is required that a selection of scheme is based on having L3 header of either IPv4 OR IPv6, but it is of no importance which of the two is present, than a unit should be defined with 2 interchangeable headers: IPv4, IPv6.

The initialization routine retunes a NetEnv handle to be used later to specify that Network Environment.

Depending on context, there are limitations to the use of NetEnvs. A port using the PCD functionality is bound to a NetEnv. Some, or even all, ports may share a NetEnv, but it is also possible to have one NetEnv per port. When initializing a scheme, a Custom Classifier root, or when binding a port to the PCD, one of the required parameters is the handle of an initialized NetEnv. The driver uses the definitions of that NetEnv to initialize that scheme or Custom Classifier root. When a port is bound to a Keygen scheme or a Custom Classifier root, it must be bound to the same NetEnv.

For the flow's definition, the different PCD modules may only rely on distinction units as defined by their environment. When initializing a scheme for example, a PCD module may not choose to select IPv4 as a match for recognizing flows unless IPv4 was defined in the relating environment. In fact, to guide the user through the configuration of the PCD, each module's characterization in terms of flows is not done using protocol names, but rather environment indices.

In terms of hardware implementation, the list of distinction units sets the Lineup Confirmation Vectors (LCVs) and are later used for match vector and CC indexing. The shim header LCVs are conventionally assigned from LSB up, so the first shim header is 0x0000_0001. For more details on the implementation, refer to Global FMan-PCD Resources on page 730.

Runtime Modifications: A Network Environment may not be changed at runtime. New NetEnvs may be set, and unused NetEnvs may be deleted anytime.

Available API:

- `FM_PCD_NetEnvCharacteristicsSet`
- `FM_PCD_NetEnvCharacteristicsDelete`

7.4.4.3.1.7.2.3.4 Software Parser

The PCD allows the extension of the hardware parser by loading the software parser code for further manipulation. When this is required, the user passes the image of the software parser code and a table of labels to the driver. This represents the entry-points in the software parser code. If more than one code piece is required for a specific protocol (for example, to be used by different ports) an index is added to the labels table. Later, when configuring a port that uses one or more software parsing attachments, each protocol header may be bound to one of the previously declared labels. This is done by setting the software parser enable indication for one or more protocols headers, and indicating the software parser index (relative to that protocol header). The software parser code will run for that port after the hardware parser recognizes that header. In other words, the specified protocol header is in fact the trigger for the software parser to be activated. It is typical for the software parser to parse a private header that was previously defined as a NetEnv unit and then mark its existence for classification and distribution.

The software parser loading routine must be called only when the PCD is disabled and no ports in the system are using the parser. On initialization this means that the routine, if needed, must be called after `FM_PCD_Init` and before `FM_PCD_Enable`.

Runtime Modifications: Software parser may not be changed at runtime.

Available API:

- `FM_PCD_PrsLoadSw`

7.4.4.3.1.7.2.3.5 Keygen Schemes

The scheme entity relies on the hardware entity. There are 32 Keygen schemes in a frame manager. When a PCD is defined in a single partition environment, it is the owner of all 32 schemes. When a PCD is defined in a multipartition environment, the user must specify how many schemes are required for this partition. Once schemes are allocated for a specific partition, it may be used only by ports on that partition.

Within a partition, the schemes order is relevant. When initializing a scheme, the user must specify the following:

- Relative index, relative to the partition's schemes.
- Network environment handle.
- Match criteria, or which frames should be processed by the scheme.
- Keygen action (such as hash, FQID mask, and manipulation).
- Distribution FQIDs.

The match criteria (if used), is based on the NetEnv characteristics units. Schemes that are to be used directly should be configured as such, by specifying a scheme ID rather than using match criteria or specifying distinction units. Upon initialization, the driver returns a handle to the initialized scheme. This handle can be used later to specify the scheme.

Keygen schemes are dependant on parser results. They may be used immediately after the parser by direct mode or by using the match criteria. Schemes may also be used after the Custom Classifier or the policer. This flow is typically used for flow control redistribution. In this case, to avoid infinite loops the scheme is reached only in direct manner and not by match criteria.

The keygen action consists of the construction of the key and the definition of the distribution. The key is constructed by a set of extract actions arranged in the driver as an array of extractions. Extractions may be done from data, from Parse Result, from default values, but most commonly - from the header. When extraction is taken from the header it may be described generically by size and offset, or it may be an extraction of the full field. For a full list of supported headers and fields, see Supported Network Protocols on page 766.

When a scheme is initialized, the user must specify the next engine to which the frame should pass after it is processed. The next specified engine must be initialized and valid at this point. Frames may pass to the Custom Classifier or the policer, or they may be directly enqueued to an FQID.

Once schemes are defined, ports may be bound to them. A port may be bound to as many schemes as needed, as long as they are from the same partition and the same NetEnv.

Following figure shows an example of scheme setting and connection to the NetEnv, as shown in Network Environment Characteristics on page 736.



Figure 106. Schemes Example

Runtime Modifications: Valid schemes may be modified at runtime by calling the scheme initialization routine for an existing scheme with the following differences:

1. Passing the scheme handle as retuned by the original initialization routine (instead of the scheme's relative ID).
2. Setting 'modify' to be 'TRUE'.

New schemes may be set and unused schemes may be deleted anytime.

Available API:

- `FM_PCD_KgSchemeSet`
- `FM_PCD_KgSchemeDelete`

7.4.4.3.1.7.2.3.6 Custom Classifier Root

A Custom Classifier root (or actually the entire CC graph) may be defined per FMan Port or shared by ports on the same partition. It is a set of lookups defined to classify, route and perform manipulation on a flow of frames. The CC graph is built bottom up by connecting CC Nodes. When a node (which is not a leaf in the graph) is set, it points to other nodes. These other nodes must already be initialized.

A CC root is defined by a set of entries that construct the root of the graph, and Custom Classifier Nodes of different types.

Once all nodes in the graph are ready and connected, the root is built by calling the `FM_PCD_CcRootBuild` routine. The root of the graph is in fact an array of up to 16 root entry nodes. The entry point for a frame is one of the CC root entries, depending on the engine that precedes the CC which is the Keygen.

According to the parser results (which is defined by the NetEnv setting) and Keygen configuration, a frame is directed to one of the entries in the CC root array.

When building the CC root, the user must specify its NetEnv id. Up to four distinction units may define the selection of one node (out of the 16), in a simple bit selection method. The following table shows the CC Root nodes selection (0 = unrecognized by parser, 1 = recognized by parser).

Table 131. CC Root Nodes Selection

Unit0	Unit1	Unit2	Unit3	Selected Node
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14

To allow more than 4 units to be involved in the selection, the 16 entries may be divided into groups. The table above demonstrates an organization of one group of 16 nodes, but other organizations are possible:

2 groups of 8 -> each group selected by 3 units (to select nodes 0-7 relative to this group's base)

4 groups of 4 -> each group selected by 2 units (to select nodes 0-3 relative to this group's base)

8 groups of 2 -> each group selected by 1 units (to select nodes 0-1 relative to this group's base)

16 groups of 1 -> indifferent to units (single node group always selected)

2-8 groups of varied sizes (8-1)



Figure 107. CC Root - 5 groups example

When building the CC Root, the user must specify the number and size of groups. Then, for each group, an array of per-root-node parameters is passed. The array is ordered according to the table above.

A simplified way of using the CC, is to define up to 16 different groups of one root-node each. In this way, all traffic from a specific Keygen scheme is going to the same group, which is a single node, and no NetEnv unit are selected. Groups 3 and 4 in figure above are an example of a single root group.

The following figure shows a combined use of the NetEnv units in Keygen and Custom Classifier, based on the previous NetEnv and Keygen scheme examples.



Figure 108. Keygen -> Custom Classifier Example

When a CC root or node is initialized, the driver returns a handle to the root or node respectively. This handle may be used later for specifying the root or node. For example, to build a root, the nodes are specified by passing their handles, and a root

handle must be passed when defining a port that uses the Custom Classifier. A port may be bound only to one root, from the same partition and NetEnv as the port.

Runtime Modifications: Custom Classifier nodes may be modified by using one of the routines listed in the "Available API" below.

Custom Classifier Roots may not be changed at runtime. New nodes and roots may be defined and unused ones may be deleted anytime.

Available API:

- `FM_PCD_MatchTableSet`
- `FM_PCD_MatchTableDelete`
- `FM_PCD_HashTableSet`
- `FM_PCD_HashTableDelete`
- `FM_PCD_CcRootBuild`
- `FM_PCD_CcRootDelete`

Specific runtime API:

- `FM_PCD_CcRootModifyNextEngine`
- `FM_PCD_MatchTableModifyNextEngine`
- `FM_PCD_MatchTableModifyMissNextEngine`
- `FM_PCD_MatchTableRemoveKey`
- `FM_PCD_MatchTableAddKey`
- `FM_PCD_MatchTableModifyKey`
- `FM_PCD_MatchTableModifyKeyAndNextEngine`
- `FM_PCD_MatchTableFindNModifyNextEngine`
- `FM_PCD_MatchTableFindNRemoveKey`
- `FM_PCD_MatchTableFindNModifyKeyAndNextEngine`
- `FM_PCD_MatchTableFindNModifyKey`
- `FM_PCD_HashTableAddKey`
- `FM_PCD_HashTableRemoveKey`
- `FM_PCD_HashTableModifyNextEngine`
- `FM_PCD_HashTableModifyMissNextEngine`

7.4.4.3.1.7.2.3.7 Match-Table Nodes

The driver defines two types of Match-Table nodes - Exact-Match nodes and Indexed-Lookup nodes. On both types of nodes a table of entries is defined where each entry leads to a selected next-engine with a selected action. The next-engines may be another CC Node, a Keygen scheme, a Policer profile or an enqueue action to a QM queue. In the last case, the queue may be either an Fqid (frame queue id) that was previously defined - typically by the Keygen, or an explicitly specified new Fqid that overrides any previous Fqid selection.

The difference between the two types of nodes is in the way an entry is selected in the node's table.

On an exact-match node, the user defines an extraction of data taken from the frame or the Internal-Context. The table of entries represent different possible values (keys) for this extraction, so that for each key a next-action is selected. An extra 'MISS' entry is also specified.



Figure 109. Exact-Match Node Example

On an Indexed-lookup node, up to 2^12 may be defined. The user selects 12 bits out of the Internal Context as an index to an entry in the table. The 12 bits may be masked to select less bits and a smaller table.



Figure 110. Indexed-Lookup node example

Two methods for CC node allocation are available: dynamic and static. Static mode was created in order to prevent runtime alloc/free of FMan memory (MURAM), which may cause fragmentation; in this mode, the driver automatically allocates the memory according to maximal number of keys, as received from the user. The driver calculates the maximal memory size that may be used for this CC node, taking into consideration whether key masks are required and node's statistics mode.

In dynamic mode, maximal number of keys is not provided (equals zero). At initialization, all required structures are allocated according to current number of keys. During runtime modification, these structures are re-allocated according to the updated number of keys.

7.4.4.3.1.7.2.3.8 Hash-Table Nodes

The Hash-Table node is a driver managed Hash table. It is defined as a next engine and may follow other CC nodes. The Hash-Table module uses driver lower level CC structures and provides an abstraction layer API consisting of AddKey/ RemoveKey routines. By using this module, the user may easily use a hash table based on Keygen key extraction and hash calculation. When initializing this node, the user should define parameters regarding the basic key used for hashing and the structure and size of the hash table (sets/ways).

7.4.4.3.1.7.2.3.9 Manipulations

On the structural aspect, Manipulation nodes are not graph nodes in the way that they do not effect the flow of a frame, and they are not in fact a graph junctions. Manipulations nodes are defined as extensions to existing CC nodes of all types. Any key on any CC node may have a manipulation characterization on top of the next engine definition. This is realized by CC node parameter `h_Manip` which is a handle to a previously initialized Manipulation node (according to the bottom-up principle). The Manipulation node itself does not have a next engine definition and the frame's flow is determined by the last CC node.



Figure 111. CC Node With Manipulation

Available API:

- `FM_PCD_ManipNodeSet`
- `FM_PCD_ManipNodeDelete`

Specific runtime API:

- `FM_PCD_ManipNodeReplace` (only available for Header-manipulation)
- `FM_PCD_ManipGetStatistics`

NOTE

- For all manipulation types below, the user must call 'FM_PCD_SetAdvancedOffloadSupport' before calling '`FM_PCD_Enable`'.
- For each RX/OP-Ports that will work with the above FM-PCD, the user should have at least 16 tnums (num of tasks). in order to set the tnums the user should call '`FM_PORT_ConfigNumOfTasks`'.
- It is also required to set the DMA transactions to be per port by calling '`FM_ConfigDmaAidOverride`' with 'FALSE' and calling '`FM_ConfigDmaAidMode`' with '`e_FM_DMA_AID_OUT_PORT_ID`'

7.4.4.3.1.7.2.3.9.1 Header Manipulation

The header manipulation is implemented by the FMan controller block, and is designed to change the incoming frame header for termination or interworking flow requirements. Header modification can be configured on a per-flow basis or for a user-determined group of flows.

The firmware defines some header manipulation structures which hold parameters for the definition of header manipulation action. It defines a basic table descriptor (Header Manipulation Table Descriptor HMTD) and a table of commands (HMCT), allowing a sequence of manipulations to be performed. The commands table may reside in either internal or external memory. The manipulation may be performed at any stage of the Custom Classifier process. As the manipulation changes the frame, the process allows an additional parsing of the processed frame once the manipulation process had ended.

The Header Manipulation (HM) mechanism is viewed by the driver as an extension to other Custom Classifier Nodes. It may take place at the beginning, the middle or the end of a CC graph, but it may not have an effect on the flow, i.e. the selection of the next action.



Figure 112. Header Manipulation CC Perspective

The HM action is represented by the driver's Manip node which is a driver sub-module (i.e. initialized by the user, its initialization routine returns an HM handle).

A Header Manipulation node is an independent unit that has no external information regarding other modules in the PCD graph, its users, its location in the flow, or the next engine it will be followed by.

A CC key or a CC root node may lead to a Header Manipulation node. The CC key/root node will define the next engine that should follow the manipulation. The next engine may be Keygen, Policer, another CC node, or PCD termination (enqueue).

In order to use the HM, the user should first create a Manip node, and than use its handle when defining the CC Node that points to this manipulation action.

A Header Manipulation action may be defined as one of the following manipulations:

- Remove
- Insert

- Fields Update
- Custom

More than one manipulation is allowed only if they are to be performed in the order above and only one manipulation of each type.

Other orders or a list of manipulations of the same type may be achieved by chaining some manipulation nodes by using the `h_NextManip` handle of the Manipulation parameters structure.

HM nodes may be shared, so that the same HM handle can be passed to more than one CC key.

By default, each frame goes back to the parser to be re-parsed after the manipulation. However this behavior may be disabled and may have an effect on performance as will be explained in the restrictions note below. It is controlled by the Header Manipulation node parameters.

The parsing option applies to whatever the user initialize as a Manip node - i.e. if the node contains a number of commands, the parsing can be done after all the commands and not between them. However, if the set of commands is initialized as a number of nodes that are chained together, the parser may be run after each node.

The driver aims to optimize performance and MURAM utilization. It does so by internally creating a single command table for chained nodes. Note that this optimization is NOT possible if parsing is required between manipulations and in this case the manip nodes are cascaded.

Note that when manipulations are chained, some restrictions apply:

1. Sharing of chained nodes is only possible on the head of the manipulation and not on inner nodes, i.e. all the manipulation is shared and not parts of it.
2. When parsing is required between manip nodes, the optimization described above is NOT possible and in this case the manip nodes are cascaded.
3. When parsing is required between manip nodes, the next engine of the last CC node may NOT be another CC node; i.e. chained nodes with parsing between them may only exist at the end (and not in the middle) of the CC graph.

7.4.4.3.1.7.2.3.9.2 IP Reassembly

The FM supports IP reassembly for both IPv4 and IPv6. The FMan accumulates IP fragments until enough have arrived to completely reconstitute the original datagram. IP Reassembly supports a maximum of 16 fragments per frame. Each fragment must reside in a single buffer (not in a Scatter/Gather frame).

The IP Reassembly driver utilizes the FMan Controller and FMan PCD resources in order to provide a full IP Reassembly solution.

The driver's interface is not identical to the hardware resources and provide an abstraction layer to the hardware resources. All IP Reassembly hardware data structures used for IP reassembly manipulation are represented by the software Custom Classifier Manipulation node. On top of the CC Manipulation, the driver internally defines the other resources needed for the full flow.

IP Reassembly flow

Fragments arriving on an Rx (or offline parsing) FMan Port that was configured to support IP Reassembly are recognized and marked by the software parser extension. These frames are steered to direct schemes the Keygen and caught by dedicated schemes that pass them to the Custom Classifier. The CC Root object is configured so that the IP fragments will reach a dedicated root entry node that contains a CC manipulation node. At this point, the IP Reassembly is performed. When a full frame is gathered, it is passed by the FMan controller back to the parser as a full reassembled frame. It is then passed to the Keygen and may be distributed and classified as any other frame.

What should the user do?

The following sequence describes the steps the user must take in order for the flow above to work.

- Initialize general DPAA (BM, BM Portal, BM Pools, QM, QM Portal, FMan and FMan PCD)
- Initialize the Rx/Offline FMan Port on which reassembly should run

- Define PCD as follows:
  - Set a Network Environment with one of the following options:
    - `HEADER_TYPE_IPV4` unit with `IPV4_FRAG_1` option for IPv4 reassembly manipulation.
    - `HEADER_TYPE_IPV6` unit with `IPV6_FRAG_1` option for IPv6 reassembly manipulation.

    Note that if the user needs IPv4 or IPv6 units for other use, the fragmentation units may not be shared and dedicated units must be defined.
  - Allocate the first one or two schemes - one if only IPv4 is used, 2 if IPv6 is also used. The user should not configure those schemes, just save these schemes from other usage. The driver will use the first scheme for IPv4, and if needed, it will use the second for IPv6.
  - Create reassembly manipulation using `FM_PCD_ManipNodeSet` routine. Pass the relative id's of the schemes allocated above (A single manipulation module should be created for both IPv4 and IPv6 fragmented frames, passing all relevant parameters).
  - If CC is used, it is user's responsibility to leave two unused entries when building the CC root nodes (i.e. the total number of entries between all groups should not exceed 14).
  - Set at least one scheme to catch regular/reassembled frames.
- When binding the Rx/Offline FMan Port to the PCD properties (i.e. calling `FM_PCD_SetPCD`), pass a handle to the created Reassembly Manipulation node.

Note that in order to perform distribution or classification on IPv4/IPv6 frames (unrelated to reassembly of IPv4/IPv6 fragments), independent IPv4/IPv6 units with no option must be explicitly defined.

What does the driver do?

In order to provide the required support for IP Reassembly, the driver performs some internal actions triggered by the user configuration. The following information describes the actions the driver internally performs and has no functional relevance to the user:

- When reassembly is required, the driver internally enables parser recognition of IPv4/IPv6 and shim2 - which is the IP Reassembly extension. This is triggered by the user defining NetEnv units with options: `IPV4_FRAG_1/IPV6_FRAG_1`.
- The driver loads the software parser that identifies IP fragments and enables its operation for the required FMan Port.
- The driver defines one or two (one for each IP version) Keygen schemes that recognize IP fragments and are programmed to generate an IP Reassembly key. When a frame is recognized as an IP fragment (by the Parser), it is steered to these Keygen schemes. The user should allocate the first one or two (for IPv4 and/or IPv6) schemes and pass their relative id's to the driver. The driver will internally initialize the relevant reassembly schemes when required.
- Each of the schemes above is programmed by the driver to point to a group in the Custom Classifier Root. If the user did not create a CC Root, the driver internally creates a new one. In both cases, the driver creates the needed group/s in the CC Root. It always uses the last two groups. It is user's responsibility to have at least two empty entries (one for a single IP version, two for both).
- The driver attaches the Manipulation sequence (created by the user) to the appropriate root entry node in the CC Root, causing the reassembly of IP fragments.

NOTE

The software parser code required to support reassembly may not coexist with user software parser code. If the user supplies IPv4 or IPv6 software parser code, it must include the code for handling IPv4/IPv6 reassembly according to the FMan controller spec.

Suggestions of how to use IPR in a system

The PCD with the IPR should identify frames up to L3; i.e. if the frame is IP or not.

In case it isn't an IP frame it should pass the desire PCD. IP frames should pass the reassembly process and than be directed to OP-Port to be classified according to their L3 and above.

7.4.4.3.1.7.2.3.9.3 IP Fragmentation

The FMan supports IP fragmentation for both IPv4 and IPv6. The fragmentation mechanism is implemented by the PCD, specifically by the Custom Classifier. IP fragmentation may be performed using an Offline Parsing FMan Port with a specific PCD configuration that will be described in this section.

The software driver provides API for initializing the IP fragmentation mechanism. driver's interface is not identical to the hardware resources and provide an abstraction layer to the hardware resources. Both of the AD (action descriptor) tables that used for IP fragmentation manipulation represented by the software Custom Classifier nodes using CC Manipulation. IP Fragmentation manipulation is used for fragmentation of IPv4 and IPv6 frames according to a specific MTU. This manipulation can be used on Offline Parsing ports only and as a part of the port's PCD definition. CC Nodes should have an IP fragmentation manipulation characterization in order to trigger this manipulation. This means that in order to create and initialize the IP fragmentation hardware, the user should create a Custom Classifier Node with Manipulation (refer to Custom Classifier Root on page 739). All relevant parameters such as MTU are defined during this module creation.

Following is the sequence that should be followed:

- Initialize general DPAA (BM, BM Portal, BM Pools, QM, QM Portal, FMan and FMan PCD)
- Initialize FMan Port of type Offline Parsing
- Define fragmentation PCD as follows:
  - Initialize an empty Network Environment (without any units)
  - Create fragmentation manipulation using `FM_PCD_ManipNodeSet` routine.
  - Create CC Node by calling `FM_PCD_MatchTableSet`/`FM_PCD_HashTableSet` and attached the fragmentation manipulation previously created to the desired key.
  - Build a CC Root with 1 group that points to the previously defined CC Node .
- Bind the Offline Parsing FMan Port to the PCD properties by calling `FM_PORT_SetPCD`

Manipulation parameters

- MTU of the fragmentation manipulation.
- Scratch Buffer Pool ID is a buffer pool that is required by the fragmentation process in order to ensure correct release operation of the frames and fragments. All IP Fragmentation Table Descriptors should use the same Scratch Buffer Pool ID. This pool must not be used by any other process or engine in the system.
- Don't Fragment Action - by setting this parameter the user can determine the action to be taken in case the IP packet is larger than the defined MTU and the 'Don't Fragment' (DF) bit of the frame is set.

NOTE

The software parser code required to support fragmentation may not coexist with user software parser code. If the user supplies IPv6 software parser code, it must include the code for handling IPv6 fragmentation according to the FMan controller spec.

Restrictions:

1. Tx confirmation is not supported.
2. Only Bman buffers shall be used for frames to be fragmented.
3. IP-Fragmentation will not work on OP-Port with VSP enabled.
4. fragmentation of IP-fragments is not supported
5. IPv4 packets containing header option field are fragments by copying all option fields to each fragment, regardless of the copy bit value.
6. Maximum number of fragments per frame is 16.

Suggestions of how to use IPF in a system:

In case one of the #1-#2 3 restrictions above is critical than it is suggested not to use IPF on OP-Ports that receive frames from the GPP and to do it on the GPP itself. We also suggest to put the IPF on a OP-Port just before the TX-Port.

7.4.4.3.1.7.2.3.9.4 IPSec Manipulation

The IPSec Manipulation is a specific instantiation of the special offload manipulation. It is designed to handle IPSec traffic in order to support the following actions:

- Support of variable outer header size

  The user should initialize a Manipulation node of this type passing the relevant parameters
- Support for both ipv4/ipv6 IP version within SA

  The user should initialize a Manipulation node of this type passing the relevant parameters.
- ECN/DSCP copying from inner/outer IP header to outer/inner.

  In order to use this functionality the user must follow the following steps:
  - Define a Manipulation node of this type passing the relevant parameters
  - For the relevant Rx/OP port, define a buffer prefix that includes at least the Keygen hash result.
  - Use SEC parameters to support this operation

7.4.4.3.1.7.2.3.10 **Frame Replicator**

The Frame Replicator (FR) is designed to duplicate incoming packets and route them to separate destinations. It is defined as a next engine and may follow other CC nodes, i.e. Match-table key, Hash-Table key or a CC-Root entry.

A Frame Replicator is realized by a group of members, where each member defines a replication of the incoming frame and a route to continue.

The next engine after FR is restricted to one of the following:

- Enqueue (PCD Termination)
- Policer
- Keygen (Direct scheme that leads to either Policer or PCD Termination)

When initializing an FR node, the user must define the maximum number of members this node may contain. The actual number of members may be modified on runtime by adding and removing FR group members.

Runtime modifications of add/remove members to/from the group can be done at any point in the system and in any location of the members group (first, middle or last). Note that runtime-modifications require the use of Host Command.

The order of the members in the group is of significance as the implementation of the replication is serial.

Manipulation may be applied to:

1. The whole group. The manipulation node should be placed before the replication group. That means that the FR is the next-engine of the Manip node. The Manip node is the next-engine of a key in a Match-table or Hash-table.
2. The last member of a FR group. That means that the manip node is the next-engine of the last member of the FR group.

NOTE

No support of Manip node after the "non-last" members.

The driver supports sharing of FR nodes means that FR group may be shared by more than one source.



Figure 113. Frame Replicator Following a CC Node

Available API:

- `FM_PCD_FrmReplicSetGroup`
- `FM_PCD_FrmReplicDeleteGroup`

Specific runtime API:

- `FM_PCD_FrmReplicAddMember`
- `FM_PCD_FrmReplicRemoveMember`

NOTE

- For all manipulation types below, the user must call `'FM_PCD_SetAdvancedOffloadSupport'` before calling `'FM_PCD_Enable'`.
- For each RX/OP-Ports that will work with the above FM-PCD, the user should have at least 16 tnums (num of tasks). In order to set the tnums, the user should call '`FM_PORT_ConfigNumOfTasks`'.
- It is also required to set the DMA transactions to be per port by calling '`FM_ConfigDmaAidOverride`' with 'FALSE' and calling '`FM_ConfigDmaAidMode`' with '`e_FM_DMA_AID_OUT_PORT_ID`'

7.4.4.3.1.7.2.3.11 Policer Profiles

The policer profile entity relies on the hardware entity. It defines rules for policing for a certain flow. There are 256 different profiles in a frame manager that may be organized in per port windows. Some profiles may be shared between ports on the same PCD. By default, the number of shared profiles is set by the driver, but the user can also configure it to a different value. Shared profiles are typically used for aggregation.

When a PCD is defined in a single partition environment, it is the owner of all 256 profiles. When a PCD is defined in a multipartition environment, it is the owner of its shared profiles along with all the profiles that will be allocated per port for ports on this partition. The user must explicitly allocate per-port profiles for each port (if required), after PCD is initialized and prior to the profile initialization. Note that per-port profiles are the only PCD resource that is explicitly allocated and initialized for a specific port.

After profiles are mapped, the user may initialize each of the profiles by stating the following:

- Type
  - Shared
  - Per-port
- Offset relative to the port or to the shared group of profiles
- Characteristics

Once initialized, a handle is assigned to the profile for later use.

The Policer may be used after the Parser, Keygen or Custom Classifier, or solely - without activating any of the other PCD engines. It is not dependant on any previous output such as parser result. The policer may be used more than once in a frame flow. The next action after a police profile is either to pass the frame to a direct Keygen scheme for a new distribution (typically for control frames coming from the Custom Classifier), to pass the frame to another profile (always a shared profile, typically an aggregators), or to enqueue the frame to an FQID.

When other engines select a policer profile as the next engine, its handle must be passed. An exception is when a per-port profile is specified as the next engine of a scheme or of a "overrideParams" CC key. In these cases a port-relative index is required instead. The reason for this is that the required Policer Profile may not be initialized at this stage and hence have no handle. This irregular behavior is because CC Roots and KG schemes may be shared by ports, and at the time of scheme/root initialization, they are not yet bound to specific ports. In this context, the profile selected may in fact be uninitialized and therefore can't be verified by the driver. It is therefor user's responsibility to make sure it is set prior to port- PCD binding.

Runtime Modifications: Valid profiles may be modified at runtime by calling the profile initialization routine for an existing profile, passing the profile handle as retuned by the original initialization routine, and specifying modify (instead of the profile's relative id). New profiles may be set and unused profiles may be deleted anytime.

Available API:

- `FM_PCD_PlcrProfileSet`
- `FM_PCD_DeleteProfilePlcr`

7.4.4.3.1.7.2.3.12 PCD Organization

By initializing PCD resources, the user creates a directed graph in which the parser is the source of the graph and the FQIDs are its endpoints. Following figure shows a generalized example of a basic PCD graph.



Figure 114. PCD Organization

7.4.4.3.1.7.2.3.13 PCD Definition Sequence

When a PCD graph is defined, its resources must be initialized bottom up when there's a dependency between them. Following figure shows the order of initialization (starting at the top of the figure) in a specific sequence.



Figure 115. Definition Sequence

7.4.4.3.1.7.2.3.14 Host Command

Some PCD functionalities may be managed by either memory-mapped registers or by the host command mechanism to allow independent programming in a multipartition environment. In a single partition environment in the FMan driver, the host command mechanism is optionally used, but in a multipartition environment, wherever available, only the host command is used to prevent a risk of racing. The host command driver is a part of the PCD driver and is initialized internally by the driver, using user parameters.

When PCD is first initialized in a single-partition environment, the user must specify whether the host command should be used, and if so, host command parameters are required. In a multipartition environment, the use of the host command is forced and all host command parameters are required. When PCD initialization routine is called by the master/single partition driver, the user parameters include host command port parameters (such as port id, virtual address, and default queues) and the FMan Port for the host command is internally initialized.

7.4.4.3.1.7.2.3.15 PCD Statistics

The FMan PCD API provides access to all the statistics gathered by the FMan PCD engines hardware. Statistics is enabled by default but may be disabled/enabled at runtime using the dedicated API.

The following API routines may be called at any time after initialization to retrieve any of the following FMan PCD counters:

- `FM_PCD_GetCounter`
- `FM_PCD_KgSchemeGetCounter`
- `FM_PCD_PlcrProfileGetCounter`

7.4.4.3.1.7.2.3.15.1 Custom Classifier Statistics

A CC node supports statistics gathering on per-key basis. In order to enable statistics gathering by a CC node (Match table or Hash table), statistics mode must be provided upon initialization of that node and this will determine the statistics mode for all keys of the CC node.

Next, statistics should be enabled per-key, meaning statistics should be enabled for every key that the user wishes to monitor.

After these steps, the following API routines may be called to retrieve the statistics:

- `FM_PCD_MatchTableGetKeyCounter`
- `FM_PCD_MatchTableGetKeyStatistics`
- `FM_PCD_MatchTableFindNGetKeyStatistics`
- `FM_PCD_HashTableFindNGetKeyStatistics`

7.4.4.3.1.8 FMan Port Driver

The FMan Port driver module refers to the per-port features of the FMan, including port configuration and initialization, runtime functionalities and PCD binding.

7.4.4.3.1.8.1 FMan Port Hardware Overview

The FMan hardware supports a SoC dependent number of inline and offline FMan Ports of the following types:

- 1G Rx Ports
- 1G Tx Ports
- 10G Rx Ports (may be eliminated on some SoCs)
- 10G Tx Ports
- Offline/Host-command ports

Port configuration is controlled through a set of per-port, type-dependent memory mapped registers. I.e. Each port has its own memory map area. In addition, some FMan common registers also effect port behavior - for example, global resources such as tasks number are declared in the common registers are.

7.4.4.3.1.8.1.1 FMan Port Driver Software Abstraction

The FMan Port module is an independent module. On port configuration, the user selects the type and the mode of each port (Tx/Rx, 1G/10G, online/offline/Host command, regular/independent), and specifies the port index relative to its type. This index is not related to the hardware port id as described in the hardware spec.

The driver provides abstraction to the common/private division of registers location in the memory map. i.e. all registers that are logically relevant to the port are handled by the FMan Port driver, even if they physically belong to the common FMan memory map.

7.4.4.3.1.8.2 How to use the FMan Port Driver?

The following sections provide practical information for using the software drivers.

7.4.4.3.1.8.2.1 FMan Port Driver Scope

- FMan Port hardware structures configuration and enablement
- Resource allocation and management
- FMan port types support
- Offline-Parsing ports
- Independent-Mode
- Simple BMI-to-BMI (regular) mode
- PCD Binding
- Rate limiting
- Interrupt handling
- Statistics support

7.4.4.3.1.8.2.2 FMan Port Driver Sequence

- FMan Port Config routine
- [Optional] FMan Port advance configuration routines
- FMan Port Init routine
- FMan Port runtime routines
- FMan Port Free routine

7.4.4.3.1.8.2.3 FMan Port Driver Functional Description

The following sections describe main driver functionalities and their usage.

7.4.4.3.1.8.2.3.1 FMan Port Configuration and Initialization

On FMan Port driver initialization, the software configures all FMan Port registers. It supplies default values where no other values are specified, it enables hardware mechanisms and initializes software data structures for software management.

By the time initialization is done, FMan is ready to be used and any of the FMan sub-modules (FMan-Ports, MAC's etc.) may be initialized.

7.4.4.3.1.8.2.3.2 FMan Port Types

The driver provides API for the initialization of the following port types/modes:

- Tx 1G port
- Tx 1G port - independent mode
- Rx 1G port
- Rx 1G port - independent mode
- Tx 10G port
- Tx 10G port - independent mode
- Rx 10G port
- Rx 10G port - independent mode
- Offline Parsing Port

The driver also holds a single host-command port internally when mandatory (multi-partition environments) or when user explicitly requires it.

7.4.4.3.1.8.2.3.3 Independent-Mode

A port may be configured to operate in independent-mode. In such case no PCD may be defined. A slightly different set of parameters is required as the FMan functions differently.

7.4.4.3.1.8.2.3.4 Resource Management

FMan Port related resources (TNUMs, DMAs, FIFOs, etc.)- These resources are used by the BMI. The driver selects default values for these resources but they may be need some tuning depending on the specific application, based on the total number of ports used and the performance requirements of the system. The driver provides an API routine `FM_PORT_AnalyzePerformanceParams` that uses performance monitoring mechanism in order to see the resources utilization at runtime.

The FMan Port driver allocates its resources by calling the FMan "front-end" driver. The FMan "front-end" allocates the resources by calling the "back-end" through IPC if its in guest-mode or through direct call if its not in master-mode. The port driver does not access those resources at run-time; the resources are being used only by the hardware of a port.

PCD related resources (Keygen-schemes, policer-profiles, etc.)-During the initialization of the FMan-PCD driver on each partition, the driver allocates all the required resources (configurable by the user) through IPC call to the "back-end" driver. From that point, all the resources are being handled locally on the partition. Note, that all access to these resources are still done through host-command and that assures proper synchronization between different partitions (i.e. one can access these resources by mistake from a different partition in the system).

PCD Custom-Classifier tables-The CC tables are being allocated on the MURAM memory. This means that upon initialization of this partition, piece of MURAM should be allocated to the partition (according to how much the partition requires). From that point, the local PCD driver will manage the MURAM allocation by itself.

7.4.4.3.1.8.2.3.5 Virtual Storage Profiles Support

An FMan Port may use the legacy Physical Storage Profile or the Virtual Storage Profiles (VSP). This section will discuss the usage of VSP by an FMan port, while more information about the VSP mechanism which is implemented by the driver as separate entity `FM_VSP`, can be found in FMan VSP Driver on page 763.

When a user wants to set an Rx or OP port to work in virtualization mode using VSP's rather than the physical SP, user should call the function which allocates a storage profile window (range of VSPs allocated in continuously manner) to a port. The user should also define which profile in this range should be used as default SP; note that the default profile should be a relative index within the allocated window. Upon calling the window allocation routine, the driver enables virtual mode (i.e. using VSPs) for this port, allocates its profiles and defines default SP. In order to redirect a packet into a certain VSP, user may set the 'relative-VSP-id' within the PCD graph nodes (e.g. in the match-table entries). The value in the PCD graph nodes is port relative so if two ports are sharing the same PCD graph node (e.g. a match-table), the actual VSP will be selected by the 'relative-VSP-id' plus the port's base VSP as shown in the figure below.



Figure 116. FMan PCD graph and the VSP selection

Rules and restrictions regarding the use of VSP:

- When called for Rx ports, the allocation routine expects also the handle of coupled Tx port as a parameter; the driver sets automatically the Tx port to work in VSP mode also and use the same default profile for this port.
- Storage Profiles windows may not overlap; i.e. sharing of VSPs between FM ports is not allowed by the driver.

- A call to the allocation routine requires that the FM port will be disabled. In the case of Rx port, coupled Tx port should also be disabled. When an FM Port (that has VSP mechanism enabled) is enabled, at least the default profile must be initialized.
- A call to the allocation routine may not be reverted, i.e. it's impossible to disable virtualization mode.
- Number of profiles to be allocated must be a power-of-2. In addition, the "base-profile" that will be allocated by the driver will be aligned to the number-of-profiles provided by the user.
- For FM-Port that works with VSP, its classification should also use VSP; i.e. classification (e.g. KG scheme or CC-node) should NOT try to revert from VSP to the FM-Port "physical" SP.
- When user frees all resources of FM port, the driver frees automatically VSP window which have been allocated for this port.

Initialization Sequence:

- Initialize FM Tx Port
- Initialize FM Rx Port
- Allocate VSP for FM Rx Port (thus enabling virtualization mode)
- Initialize default VSP (See FMan VSP Driver on page 763)
- Enable FM Ports

Free Sequence

- Disable Ports
- Free the default VSP
- Free FM Tx Port
- Free FM Rx Port

7.4.4.3.1.8.2.3.6 Rate Limiting

The driver supports the hardware mechanism of rate limiting for Tx ports. The runtime API consists of a number of parameters including a definition of the required rate (in KB/sec for Tx ports, in frame/sec for offline parsing ports) and refers to data rate rather than line-rate.

7.4.4.3.1.8.2.3.7 Simple BMI-to-BMI (regular) mode

This is the default FMan Rx/Offline Parsing Port mode. After Port initialization and prior to Port-PCD binding, all traffic will be received on the default Rx queue. This mode is called "BMI-To-BMI" as no PCD is involved in the data reception.

This mode is useful for the early state of a port as well as when major runtime PCD modification takes place. In such a case, sometimes the whole PCD functionality needs to be manipulated and the user should temporarily detach the Port from the PCD, receive all frames on the default Rx queue and only re-attach it to the PCD after the modifications have completed.

7.4.4.3.1.8.2.3.8 Port LIODN

An FMan Port LIODN is constructed out of a base and offset.

Upon FMan Port configuration, the user must specify the port's base LIODN.

For Rx ports, the user must also specify the LIODN offset for each port. No such configuration is required for Tx and Offline Parsing ports since on transmission, the offset LIODN is taken from the frames' FD. The FD is set according to the source of the frame - if transmitted by CPU, it is dynamically set by the QM SW portal. Another scenario is frames forwarded by other engines, in such a case their FD must contain the correct LIODN offset.

7.4.4.3.1.8.2.3.9 Port-PCD Binding

Ports may be linked to the PCD graph according to their PCD binding specifications and considering partition and Network Environment restrictions.

Following figure shows a schematic demonstration of possible port > PCD binding.



Figure 117. Port-To-PCD binding example

Once a set of PCD resources is set and organized as described above, a port may be bound to all or some of the resources by calling the `FM_PORT_SetPCD` routine. This routine, is referred to as the Port-PCD bind routine. It accepts a set of parameters that specify the PCD resources used by the port, configures PCD related parameters in the port, and bounds PCD resources to the port. The `FM_PORT_DeletePCD` should be called when the port no longer needs the configured PCD functionality. This action is referred to as Port-PCD unbinding.

Another possible action that affects the Port-PCD relationship is calling FM_PORT_DettachPCD for a port that is bound to PCD. This causes the port to stop using the PCD functionalities, which results in all frames being passed to the default FQID. Note that calling FM_PORT_DeletePCD unbinds the port from the PCD functionalities by removing the connections, while FM_PORT_DetachPCD does not remove them but only causes the port to stop using them. To return to using the PCD, FM_PORT_AttachPCD should be called.

Certain runtime modifications may not be done directly, but require either the unbinding of PCD functionalities or PCD detaching. This should be done by calling the required delete/detach routines, making the desired changes, and calling set or attach to return to using the PCD. These actions will be referred to as resetting/detaching the Port-PCD. In the time between the calls of the two routines, the port continues to work, but its PCD functionalities are disabled. In both cases, all frames arriving at this time are enqueued to the default receive queue.

In the sections below, the relationship between the port and each of the PCD resources will be explained in terms of initialization and runtime modifications.

General

The port-PCD binding affects the flow of received frames on that port in terms of PCD functionality. The user must first define the general PCD for the port, using the following enumeration types, which define the superset of engines that may be used.

- e_FM_PORT_PCD_SUPPORT_PRS_ONLY (Use only Parser)
- e_FM_PORT_PCD_SUPPORT_PLCR_ONLY (Use only Policer)
- e_FM_PORT_PCD_SUPPORT_PRS_AND_PLCR (Use Parser and Policer)
- e_FM_PORT_PCD_SUPPORT_PRS_AND_KG (Use Parser and Keygen)
- e_FM_PORT_PCD_SUPPORT_PRS_AND_KG_AND_CC (Use Parser, Keygen and Custom Classifier)
- e_FM_PORT_PCD_SUPPORT_PRS_AND_KG_AND_CC_AND_PLCR (Use all PCD engines)
- e_FM_PORT_PCD_SUPPORT_PRS_AND_KG_AND_PLCR (Use Parser, Keygen and Policer)

Runtime Modifications: The engines set may be changed at runtime only by resetting the Port-PCD.

Available General Port API:

- FM_PORT_SetPCD
- FM_PORT_DeletePCD

Network Environment

When calling the Port-PCD binding routine, the user must specify a single NetEnv by passing its handle. This setting is used for the port parser and affects the PCD behavior.

Runtime Modifications: The NetEnv may not be modified at runtime. If the port requires a change of its NetEnv, it must first reset its Port-PCD connection, than use the PCD routines to do the required changes, and than re-connect to the PCD.

Parser

The hardware parser port configuration is taken directly from the NetEnv specified for the port. Other parsing configurations are explicitly defined by the user at the parameter's structure.

The software parser may be used on a per-port-per-header basis. When PCD is set per port, there is an option in the parser parameters to choose additional parameters per header. One of the optional per-header additional parameters is to enable the software parser for that header. When set, an index should be declared to select the software parser code. The header and index must be specified in the labels' table of the software parser code that was loaded on PCD initialization. Software parser enablement may be done for as many headers as required.

Runtime Modifications: Only the starting point of the parser may be changed on the fly. Any other changes require PCD resetting.

Available Port API:

- FM_PORT_PcdPrsModifyStartOffset

Keygen Schemes

In order for a port to use Keygen schemes, the port must be bound to those resources. The port may be bound to any number of schemes. At the port bind routine, the user passes a list of scheme handles, as returned by the server at scheme setting, for binding to the port. At least one scheme must be specified. All specified schemes must be valid at that time. If the initial scheme after the parser is used directly without using the match criteria, its id should be passed as one of the parameters to the Port-PCD binding routine.

Runtime Modifications: During runtime, new schemes may be set and then bound to an existing enabled port or existing schemes may be modified. Schemes that are not required by the port may be unbound. Note that when modifying existing schemes, all ports bound to those schemes are affected. If specific schemes are not required anymore, they must first be unbound from the port. If no other port is using them, they may be deleted. The selection of the initial scheme after parser (from direct to indirect and vice versa) may be also changed at runtime.

Available Port API:

- `FM_PORT_PcdKgBindScheme`
- `FM_PORT_PcdKgUnbindScheme`
- `FM_PORT_PcdKgModifyInitialScheme`

Custom Classifier graphs

If a port is using the Custom Classifier graph, an initialized Custom Classifier Root handle (as returned by the RootBuild routine) must be passed when calling the port bind routine.

Runtime Modifications: The CC graph (as well as the CC Root) itself may be modified at runtime, but ports binding to a CC Root may be changed only by detaching and than re-attaching the Port-PCD.

- `FM_PORT_PcdCcModifyTree`

Policer Profiles

Before any port profile is set, the profile allocation routine must be called to bind the port to the policer profile. This is required as the port's binding to the policer profile is not done using the port bind routine. It is only then that per-port profiles may be set, and the port bind routine is subsequently called. If Keygen or parser are not used (i.e. policer is reached directly after parser or from BMI), the port bind routine parameters must specify which policer profile is used (otherwise, no policer parameters are required).

Runtime Modifications: The initial profile selection may be changed during runtime. All profiles allocated to a port are in fact bound to this port, so no runtime binding/unbinding is possible. Uninitialized port profiles (profiles that were allocated for this port but not used) may also be set during runtime, or existing profiles may be modified. If specific profiles are not required anymore, they may be deleted. If a change in port profile allocation is required, follow the steps given below to reset the Port-PCD:

1. Port-PCD deleted
2. Profiles deleted and freed
3. New profiles allocated and set
4. Port-PCD set

Available Port API:

- `FM_PORT_PcdPlcrModifyInitialProfile`
- `FM_PORT_PcdPlcrFreeProfiles`
- `FM_PORT_PcdPlcrAllocProfiles`

7.4.4.3.1.8.2.3.10 Port-PCD Binding Changes

There are three levels of Port-PCD binding changes:

- **Basic Runtime Modifications-**May be invoked while PCD is active and on enabled ports using PCD.

- Port routines responsible for binding/unbinding to/from the modified resources.
  - FM_PORT_PcdKgBindScheme
  - FM_PORT_PcdKgUnbindScheme
- Port routines responsible for PCD change of behavior.
  - FM_PORT_PcdKgModifyInitialScheme
  - FM_PORT_PcdPlcrModifyInitialProfile
  - FM_PORT_PcdPrsModifyStartOfset

- **Port-PCD Detach Runtime Modifications-**For changes that require detaching the Port-PCD connection:
  - FM_PORT_PcdCcModifyTree

    For these modifications, take the following steps:
    - Detach the port from its PCD resources by calling the Detach PCD routine (FM_PORT_DettachPCD). After this action, the port continues to work enqueueing all frames to the default receive FQID.
    - Call one of the two routines above.
    - Re-attach port to PCD resources by recalling the set PCD routine (FM_PORT_AttachPCD).
- **Port-PCD Reset Runtime Modifications-**For changes that require resetting of the port-PCD binding.

  The following steps should be taken for any modification that is not listed under the last two items:
  - Unbind port from its PCD resources by calling the delete PCD routine (FM_PORT_DeletePCD). After this action the port will continue to work, enqueueing all frames to the default receive FQID.
  - Modify PCD resources-optional. The change may be only in the binding of the port and not on the resources. Note that the freeing and deleting of resources, and then allocating and setting resources, must be orderly, in the same manner as for initial PCD setting and final PCD deleting.
  - Bind port to PCD resources by recalling the set PCD routine (FM_PORT_DeletePCD)

All PCD routines listed above may be used for deleting and setting PCD resources. The following two routines below are used if a change of port profiles window is required (Other PORT routines are not needed as binding is done using SetPCD routine.):

- FM_PORT_PcdPlcrFreeProfiles
- FM_PORT_PcdPlcrAllocProfiles

7.4.4.3.1.9 FMan MAC Driver

The FMan MAC driver module refers to the FMan MAC controller functionalities including configuration and initialization as well as runtime and control.

7.4.4.3.1.9.1 FMan MAC Hardware Overview

The FMan hardware supports one or two kinds of MAC controllers - depending on SoC. All SoCs support three-speed Ethernet controller (dTSEC) interfaces to 10 Mbps, 100 Mbps, and 1 Gbps Ethernet/IEEE 802.3 networks which interfaces the media through external phy or SerDes device. Some SoCs also support 10 Gigabit Ethernet media access controller (10GEC) which interfaces to 10 Gbps Ethernet/IEEE 802.3ae networks via XAUI using the high-speed SerDes interface.

7.4.4.3.1.9.1.1 FMan MAC Software Abstraction

The driver provides a unique API serving both interfaces. If user tries to configure features that are supported only by one of the interfaces, an "unsupported" message will be displayed.

7.4.4.3.1.9.2 How To Use The FMan MAC Driver?

The following sections provide practical information for using the software drivers.

7.4.4.3.1.9.2.1 FMan MAC Driver Scope

This module represents the FMan MAC. It includes:

- FMan MAC hardware structures configuration and enablement
- FMan MAC controller runtime support
- PTP IEEE 1588 support
- MAC hash addressing
- Interrupt handling
- Statistics support

7.4.4.3.1.9.2.2 FMan MAC Driver Sequence

- FMan MAC Config routine
- [Optional] FMan MAC advance configuration routines
- FMan MAC Init routine
- FMan MAC runtime routines
- FMan MAC Free routine

7.4.4.3.1.9.2.3 FMan MAC Driver Functional Description

The following sections describe main driver functionalities and their usage.

7.4.4.3.1.9.2.3.1 FMan MAC Configuration and Initialization

On FMan MAC driver initialization, the software configures all FMan MAC registers. If required, MAC may be reset at that time. The driver supplies default values where no other values are specified, it defines IRQ's and sets IRQ handles. It enables hardware mechanisms and initializes software data structures for software management.

By the time initialization is done, FMan MAC is ready to be used and the relative FMan Ports may be initialized.

7.4.4.3.1.9.2.3.2 FMan MAC Addressing

On MAC initialization, the user must define a single MAC address. During runtime, the driver provides API for modifying this address and adding other addresses (depending on the specific MAC hardware support).

In addition, the driver supports the addition and removal of addresses to the MAC hash mechanism.

7.4.4.3.1.9.2.3.3 IEEE1588 Support

The driver provides the API to support the hardware IEEE1588 time-stamping. In order to use this feature, the user must first initialize the FM-RTC module. IEEE1588 functionality is always enabled on FM-MAC. Thus, no additional settings are required for the MAC. and the FM-MAC and only then they can enable this feature by calling `FM_MAC_Enable1588TimeStamp` routine. Once enabled, the user may also set the exception for receiving 1588 relevant interrupts on the MAC.

7.4.4.3.1.9.2.3.4 MAC Statistics

The driver provides statistics gathering support for all the standard (MIB) counters. For some controllers, it is necessary to use an interrupt driven mechanism for accounting for counters overflow and in order to keep track on the accurate counters. This mechanism may have some influence on performance, and therefor the driver supports statistics gathering in 3 levels:

- Full statistics-provides all standard counters but may reduce performance.
- Partial statistics-provides only special event counters (errors etc.). If selected, regular counters (such as byte/packet) will be invalid and will return -1.
- No statistics gathering.

7.4.4.3.1.10 FMan VSP Driver

The FMan VSP driver module refers to the software support provided for the Virtual Storage Profile mechanism.

7.4.4.3.1.10.1 FMan VSP Hardware Overview

VSPs may be used by user for virtualization. If a user is running with a multi-partitioned (or with a multiple software entities) system where a single MAC may be used by several software partitions/entities simultaneously, except for using a different FQID (that is already available in DPAA1.0), user may use a different VSP for each SW partition/entity; that way, the buffer may be private (rather than being shared as in DPAA1.0). It allows the virtualization of the buffer pool selection for frame storage (and other parameters related to storage in external memory) from the physical hardware ports. Using this mechanism, different packets received on the same physical port may be stored in different BM pools based on the frame header, in a similar way to FQID selection. VSPs are replacing the legacy, "physical", per-port BM Pool selection. A backward compatible mode exists and it is possible to use the original BM Pool selection, now referred to as "Physical SP".



Figure 118. Vitualization Using VSPs

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan initialization, the first VSP index dedicated to this partition must be defined (it should be an absolute index), and so is the total number of VSP's for this partition. Later, for each port using VSP's, a window of entries should be defined. VSPs may not be shared among FMan ports.

Each port has a default VSP. On each PCD classification, a VSP may be selected. Received packets will be written into the destination buffer according to the VSP parameters, while the VSP is selected according to the frame headers and the PCD configuration.

The VSP is implemented by the driver as separate entity, however, other modules of the FM driver are aware of this entity and interact with it. An FM VSP module represents a single storage profile.

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan port initialization, if using VSP mode, it should allocate and bind to a range of VSP's. On the PCD, A decision is being taken by user on every node of the PCD graph whether to continue to work with previously defined VSP or to override with a new profile.

7.4.4.3.1.10.2 How To Use The FMan VSP Driver?

The VSP is implemented by the driver as separate entity, however, other modules of the FM driver are aware of this entity and interact with it. An FM VSP module represents a single storage profile.

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan port initialization, if using VSP mode, it should allocate and bind to a range of VSP's. On the PCD, A decision is being taken by user on every node of the PCD graph whether to continue to work with previously defined VSP or to override with a new profile.

7.4.4.3.1.10.2.1 FMan VSP Driver Scope

This module represents the FMan VSP driver. It includes:

- FMan VSP hardware structures configuration and enablement
- Parsing of the buffer
- Statistics

7.4.4.3.1.10.2.2 FMan VSP Driver Sequence

This sequence includes other modules required for the VSP

- Definition of general VSP parameters on global FMan initialization
- FM Port initialization
- FM Port VSP window allocation
- FM Port enablement
- FMan VSP Config routine (for specific VSP's)
- [Optional] FMan VSP advance configuration routines (for specific VSP's)
- FMan VSP Init routine (for specific VSP's)

7.4.4.3.1.10.2.3 FMan VSP Driver Functional Description

The following sections describe main driver functionalities and their usage.

7.4.4.3.1.10.2.3.1 Virtual Storage Profile Initialization

The VSP's must be initialized prior to their usage. It is user's responsibility to initialize at least the default VSP for each port before enabling it. Similarly, it is their responsibility to initialize all other VSPs before a classification that may use some VSP is enabled.

Initializing a VSP defines the destination BM Pool buffer for a specific type of packets. It also defines the structure of the buffer - i.e. the data offset, the prefix content, etc.

7.4.4.3.1.10.2.3.2 Virtual Storage Profile Parsing

On VSP initialization, the user defines the buffer prefix content. Based on these requirements, the driver then defines the buffer prefix structure, i.e. data offset, whether certain information such as parse result should be copied to the external buffer and where it will be located. On buffer reception, the user may call VSP routines in order to get the data, as well as the buffer prefix sections such as parse result, time stamp, or Keygen output.

7.4.4.3.1.11 FMan RTC (IEEE 1588) Driver

The FMan RTC driver module refers to the software support provided for the IEEE 1588 hardware of the FMan.

7.4.4.3.1.11.1 FMan RTC Hardware Overview

The 1588 timer module interfaces to up to four 10/100/1000 or one 10G Ethernet MACs, providing current time, 2 alarms, and 2 fiper periodic pulse generators.

7.4.4.3.1.11.2 How To Use The RTC Driver?

The following sections provide practical information for using the software drivers.

7.4.4.3.1.11.2.1 RTC Driver Scope

This module represents the FMan 1588 driver. It includes:

- IEEE 1588 hardware configuration and enablement
- Support for alarm mechanism
- Support for periodic pulse
- Support for external trigger
- Runtime compensation tuning
- Interrupt handling

7.4.4.3.1.11.2.2 RTC Driver Sequence

- FMan RTC Config routine
- [Optional] FMan RTC advance configuration routines
- FMan RTC Init routine
- FMan RTC Enable routine
- FMan RTC runtime routines
- FMan RTC Free routine

7.4.4.3.1.11.2.3 RTC Driver Functional Description

The following sections describe main driver functionalities and their usage.

7.4.4.3.1.11.2.3.1 FMan RTC 1588 module utilization

The driver API provides interface to the 1588 hardware module. It initializes its registers to define the clock period and it supports the definition of the alarms and periodic pulses. Note that When setting periodic pulse, the RTC module must be disabled.

7.4.4.3.1.11.2.3.2 Utilizing IEEE1588 for MAC frames time stamping

Several FMan driver modules are involved in having the 1588 time stamping functionality activated: FMan-RTC, FMan-MAC, FMan-Port and FMan-PCD.

The initialization sequence is as described below:

After the Frame Manager is initialized, the FMan-RTC needs to be initialized by calling (with the appropriate parameters):

- `FM_RTC_Config`
- `FM_RTC_Init`

Next, the following routine should be called, only after MAC is initialized.

- `FM_MAC_Enable1588TimeStamp`

From this point and on all the Ethernet frames on this MAC are time-stamped. In order to obtain the timestamp, during the FMan Port configuration, the user must call the advance config routine:

- `FM_PORT_ConfigBufferPrefixContent` (with 'passTimeStamp' parameter set).

At run-time, for each received/confirmed frame, the user should call the following routine, passing it the frame's data pointer:

- `FM_PORT_GetBufferTimeStamp`

The routine will return the pointer to the time stamp.

7.4.4.3.1.11.2.3.3 Utilizing IEEE1588 for PTP

The sequence described in the previous section causes all the frames that are being received or transmitted by FMan to be time-stamped. However, if the user wants to distinguish PTP frames from other frames on a specific port, PCD rules need to be applied on the PCD graph for this port; i.e using the parser to recognize the PTP frame and then using an appropriate scheme to distinguish PTP frames and route them to the desired destination queues.

7.4.4.3.1.12 FMan MURAM Driver

The FMan MURAM driver module refers to the memory management of the FMan Multi User RAM.

7.4.4.3.1.12.1 FMan MURAM Hardware Overview

The MURAM is the internal memory of the FMan.

7.4.4.3.1.12.1.1 FMan MURAM Driver Software Abstraction

The FMan MURAM driver is a memory manager that allows partitioning of the MURAM. Upon initialization the user receives a handle that may be used by other modules in order to allocate and de-allocate memory blocks out of that MURAM partition.

7.4.4.3.1.12.2 How To Use The FMan MURAM Driver?

The following sections provide practical information for using the software drivers.

7.4.4.3.1.12.2.1 FMan MURAM Driver Scope

This module manages the FMan MURAM. It includes MURAM allocation and de-allocation of different sizes of required memory blocks.

7.4.4.3.1.12.2.2 FMan MURAM Driver Sequence

- FMan MURAM config and init routine
- FMan MURAM allot and free runtime routines
- FMan MURAM free routine

7.4.4.3.1.12.2.3 FMan MURAM Driver Functional Description

The FMan MURAM drivers supports MURAM memory blocks allocation and de-allocation. After initializing an MURAM partition, the user is normally required to pass its handle to other FMan driver modules. In this way, these modules may allocate and de-allocate memory blocks from this partition.

7.4.4.3.1.13 Supported Network Protocols

The following sections show the protocols that may be selected when defining NetEnv characteristics.

7.4.4.3.1.13.1 L2 Protocols

The following list shows the L2 protocols:

- `HEADER_TYPE_ETH`, with the following two options
  - `ETH_BROADCAST`
  - `ETH_MULTICAST`
- `HEADER_TYPE_VLAN`, with the following option
  - `VLAN_STACKED`
- `HEADER_TYPE_MPLS`, with the following option

    - MPLS_STACKED
- HEADER_TYPE_PPPoE
- HEADER_TYPE_LLC_SNAP

7.4.4.3.1.13.2 L3 Protocols

The following list shows the L3 protocols:

- HEADER_TYPE_IPV4, with the following options
    - IPV4_BROADCAST_1
    - IPV4_MULTICAST_1
    - IPV4_UNICAST_2
    - IPV4_MULTICAST_BROADCAST_2
    - IPV4_FRAG_1
- HEADER_TYPE_IPV6, with the following options
    - IPV6_MULTICAST_1
    - IPV6_UNICAST_2
    - IPV6_MULTICAST_2
    - IPV6_FRAG_1
- HEADER_TYPE_GRE
- HEADER_TYPE_MINENCAP
- HEADER_TYPE_USER_DEFINED_L3

7.4.4.3.1.13.3 L4 Protocols

The following list shows the L4 protocols:

- HEADER_TYPE_TCP
- HEADER_TYPE_UDP
- HEADER_TYPE_SCTP
- HEADER_TYPE_DCCP
- HEADER_TYPE_IPSEC_AH
- HEADER_TYPE_IPSEC_ESP
- HEADER_TYPE_USER_DEFINED_L4

7.4.4.3.1.13.4 Private Headers

- HEADER_TYPE_USER_DEFINED_SHIM1
- HEADER_TYPE_USER_DEFINED_SHIM2

7.4.4.3.1.13.5 Fields Supported By Driver for Keygen Extraction

Fields supported as "full fields":

- HEADER_TYPE_ETH
    - NET_HEADER_FIELD_ETH_DA

  - NET_HEADER_FIELD_ETH_SA
  - NET_HEADER_FIELD_ETH_TYPE
- HEADER_TYPE_LLC_SNAP
  - NET_HEADER_FIELD_LLC_SNAP_TYPE
- HEADER_TYPE_VLAN
  - NET_HEADER_FIELD_VLAN_TCI

    (index may apply:
    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_LAST)
- HEADER_TYPE_MPLS
  - NET_HEADER_FIELD_MPLS_LABEL_STACK

    (index may apply:
    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_2,
    - e_FM_PCD_HDR_INDEX_LAST)
- HEADER_TYPE_IPv4
  - NET_HEADER_FIELD_IPv4_SRC_IP
  - NET_HEADER_FIELD_IPv4_DST_IP
  - NET_HEADER_FIELD_IPv4_PROTO
  - NET_HEADER_FIELD_IPv4_TOS

    (index may apply:
    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST)
- HEADER_TYPE_IPv6
  - NET_HEADER_FIELD_IPv6_SRC_IP
  - NET_HEADER_FIELD_IPv6_DST_IP
  - NET_HEADER_FIELD_IPv6_NEXT_HDR
  - NET_HEADER_FIELD_IPv6_VER | NET_HEADER_FIELD_IPv6_FL | NET_HEADER_FIELD_IPv6_TC (must come together!)

    (index may apply:
    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST)

    NOTE

    NET_HEADER_FIELD_IPv6_NEXT_HDR with e_FM_PCD_HDR_INDEX_LAST indication, applies to the very last next header indication, meaning the next L4, which may be present at the Ipv6 last extension. On earlier revisions this field applies to the Next-Header field of the main IPv6 header)
- HEADER_TYPE_IP
  - NET_HEADER_FIELD_IP_PROTO

    (index may apply:

    - e_FM_PCD_HDR_INDEX_LAST)
  - NET_HEADER_FIELD_IP_DCSP

    (index may apply:

    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1)
- HEADER_TYPE_GRE
  - NET_HEADER_FIELD_GRE_TYPE
- HEADER_TYPE_ETH
  - NET_HEADER_FIELD_ETH_DA
  - NET_HEADER_FIELD_ETH_SA
  - NET_HEADER_FIELD_ETH_TYPE
- HEADER_TYPE_MINENCAP
  - NET_HEADER_FIELD_MINENCAP_SRC_IP
  - NET_HEADER_FIELD_MINENCAP_DST_IP
  - NET_HEADER_FIELD_MINENCAP_TYPE
- HEADER_TYPE_TCP
  - NET_HEADER_FIELD_TCP_PORT_SRC
  - NET_HEADER_FIELD_TCP_PORT_DST
  - NET_HEADER_FIELD_TCP_FLAGS
- HEADER_TYPE_UDP
  - NET_HEADER_FIELD_UDP_PORT_SRC
  - NET_HEADER_FIELD_UDP_PORT_DST
- HEADER_TYPE_UDP_LITE (relevant only if FM_CAPWAP_SUPPORT define)
  - NET_HEADER_FIELD_UDP_LITE_PORT_SRC
  - NET_HEADER_FIELD_UDP_LITE_PORT_DST
- HEADER_TYPE_IPSEC_AH
  - NET_HEADER_FIELD_IPSEC_AH_SPI
  - NET_HEADER_FIELD_IPSEC_AH_NH
- HEADER_TYPE_IPSEC_ESP
  - NET_HEADER_FIELD_IPSEC_ESP_SPI
- HEADER_TYPE_SCTP
  - NET_HEADER_FIELD_SCTP_PORT_SRC
  - NET_HEADER_FIELD_SCTP_PORT_DST
- HEADER_TYPE_DCCP
  - NET_HEADER_FIELD_DCCP_PORT_SRC
  - NET_HEADER_FIELD_DCCP_PORT_DST
- HEADER_TYPE_PPPoE
  - NET_HEADER_FIELD_PPPoE_PID
  - NET_HEADER_FIELD_PPPoE_SID

Fields supported as "from fields":

- HEADER_TYPE_ETH (with or without validation):
  - NET_HEADER_FIELD_ETH_TYPE
- HEADER_TYPE_VLAN (with or without validation):
  - NET_HEADER_FIELD_VLAN_TCI

    (index may apply:
    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_LAST)
- HEADER_TYPE_IPv4 (without validation):
  - NET_HEADER_FIELD_IPv4_PROTO

    (index may apply:
    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST)
- HEADER_TYPE_IPv6 (without validation):
  - NET_HEADER_FIELD_IPv6_NEXT_HDR

    (index may apply:
    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST)

7.4.4.4 Frame Manager Driver API Reference

This document describes the interface (IOCTLs) to the Frame Manager Driver as apparent to user space Linux applications that need to use any of the Frame Manager's features. It describes the structure, concept, functionality, and low level API.

7.4.4.5 Frame Manager Configuration Tool User's Guide

7.4.4.5.1 Frame Manager Configuration Tool User Guide

7.4.4.5.1.1 Introduction

The Frame Manager (FMan) is part of NXP's Data Path Acceleration Architecture (DPAA), a set of logical blocks that lets multiple processors (cores) interact with multiple network interfaces and accelerators with low software overhead.

The Frame Manager Configuration Tool (FMC Tool) is a command-line program that converts Parse-Classify-Police-Distribute (PCD) descriptions of network packet flows into hardware configuration code for the FMan's KeyGen, Controller, and Policer functions.

The tool provides an abstraction layer: You define your application's PCD requirements in a high-level, XML markup language (NetPDL with NXP extensions). The tool translates these definitions into code that initializes the FMan's registers and data structures. This abstraction makes learning low-level hardware details unnecessary, allows new users to be productive more quickly, and simplifies the programming task for everyone.

7.4.4.5.1.2 Frame Manager (FMan) Overview

The FMan block handles both inbound and outbound Ethernet frames.

For inbound frames, the FMan can perform multi-layer protocol header parsing, classification, policing, and distribution. For outbound frames, the FMan transmits frames in priority order.

Note: The FMan is a modular engine designed for use by many QorIQ multicore embedded devices. This document describes the FMan hardware and software as implemented for the QorIQ P4080 chip. That said, the information contained herein applies to all FMan-equipped devices.

7.4.4.5.1.3 FMan hardware architecture

The P4080 has two, identical Frame Manager instances. The architecture of a single FMan block is shown in Figure 119. Frame manager (FMan) logical block diagram on page 771.

As the figure shows, the FMan consists of several sub blocks. While this document assumes that you are familiar with the DPAA in general and the FMan in particular, the sections below provide a high-level description of these sub blocks as a refresher. These sections should help you better understand the FMC Tool documentation that follows.



Figure 119. Frame manager (FMan) logical block diagram

Separator text

7.4.4.5.1.3.1 FMan Media Access Controllers (MACs)

Each of the P4080's FMan blocks contains the following:

(1) 10 Gigabit Ethernet media access controller (10GEC), interfacing to 10 Gbps Ethernet/IEEE 802.3ae networks via XAUI using the high-speed SerDes interface.

(4) Data Path three-speed Ethernet controllers (dTSEC), interfacing to 10 Mbps, 100 Mbps, and 1 Gbps Ethernet/IEEE 802.3 networks. Individual dTSECs can use an RGMII or an SGMII interface.

See the QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual for supported combinations of 10GEC and dTSECs per FMan and per P4080, as well as for detailed information about these MACs.

7.4.4.5.1.3.2 FMan FPM, DMA Controller, and Internal Memory

The Frame Processing Manager (FPM) coordinates tasks within the FMan block. The FPM ensures that frames arriving on a particular port (or destined for transmission on a particular port) are processed by the various FMan sub blocks in the appropriate order.

The FMan DMA controller performs the physical reads and writes required to move data between FMan memory and external memory, upon command from the FPM.

The FMan internal memory is multi-ported RAM within the FMan block that is used to buffer Ethernet frames (in both the Tx and Rx directions) and to hold data structures with both external relevance (Frame Descriptors) and internal relevance (Next Invoked Action (NIA) descriptors used by FMan sub blocks to direct the flow of frames through the FMan).

7.4.4.5.1.3.3 FMan Buffer Manager Interface (BMI)

The Buffer Manager Interface (BMI) is the FMan sub block that interacts with the DPAA's Buffer Manager (BMan) block to store and retrieve Ethernet frames to/from buffers in external memory.

On an inbound Ethernet frame, the BMI performs these operations:

1. Receives the frame from an FMan MAC
2. Stores the frame in FMan memoryuntil the entire frame has been received
3. Calculates a raw, L4 checksum and passes it to the FMan's Parser sub block
4. Requests that the BMan allocate external memory in which to store the frame
5. Creates a frame descriptor (FD) for the frame (either a single-frame FD or a scatter/gather FD for frames that require more than one buffer)
6. Initiates DMA of the frame from FMan memoryto external memory
7. Informs the FMan's Queue Manager Interface (QMI) sub block that a FD is available for enqueue to one of the Queue Manager block's frame queues (FQs)

On an outbound Ethernet frame, the BMI performs these operations:

1. Receives a FD dequeued by the QMI from one of the QMan block's FQs
2. Initiates DMA of the frame from external memory to FMan memory
3. Calculates a raw, L4 checksum for the frame
4. Transfers the frame to an FMan MAC
5. Passes the L4 checksum (offset/value) to the MAC
6. Signals the BMan to de-allocate the frame's buffer(s) once transfer to the MAC is complete

7.4.4.5.1.3.4 FMan Queue Manager Interface (QMI)

The FMan's Queue Manager Interface (QMI) sub block enqueues and dequeues work to frame queues (FQs), which are resources managed by the DPAA's QMan block. Typically, the QMI enqueues/dequeues a frame descriptor (FD) - a data structure that points to a buffer containing a frame; however, the QMI can also enqueue/dequeue other kinds of objects, such as those related to host commands/responses, offline parsing, Tx confirmation, and FMan error reporting.

7.4.4.5.1.3.5 FMan Parser

The FMan's Parser sub block parses fields in the hierarchy of protocol headers contained in a received frame.

The Parser is idle until notified by the FPM that a complete frame is available for parsing. The Parser then begins to parse the frame according to the rules specified in a user-written configuration (Policy file) that specifies a header at any offset from the beginning of the frame. The Parser can examine up to 256 bytes of a frame.

The result of parsing is a 'Next Invoked Action' descriptor that determines whether the frame is filtered out or passed to the FMan's KeyGen, Controller (also called the Classifier), or Policer sub block for further processing.

The Parser can be configured to write its 32-byte parse results to a reserved area of the buffer in which the BMI stored the frame. This feature makes it unnecessary for software to re-parse a frame in cases in which the FMan's only action is to distribute incoming frames across a number of frame queues.

The Parser performs multi-layer protocol header parsing and validation for a wide range of protocols and encapsulations. For the set of standard, stable protocols shown in Figure 120. Standard Protocols - Hard-Coded Parse Tree on page 773 , the Parser uses hard-coded logic to parse the protocol headers. Parsing normally progresses from the lowest layer to the upper layers shown in the figure.

You can supplement the hard-coded support for standard protocols with 'soft' parse routines, thereby extending the Parser's capabilities to unforeseen combinations of standard protocols, new standards, and proprietary protocols with shim headers inserted between otherwise well-known protocols.

The FMC Tool can configure the Parser to use both hard- and soft-parser protocol header definitions. Details of the FMC Tool files used to describe both standard and custom protocol headers are provided in Protocol files on page 779.



Figure 120. Standard Protocols - Hard-Coded Parse Tree

7.4.4.5.1.3.6 FMan Controller

The FMan Controller sub block (also called the Classifier) performs exact match frame classification. This capability lets your application identify control traffic (or other important subsets of traffic) that can be distinguished by a small number of exact-match rules and place these frames on a frame queue dedicated to this traffic type. All frames that do *not* meet an exact-match rule proceed to the KeyGen sub block, where they are distributed across a range of frame queues based on a hash function.

Exact-match frame classification uses rules tables stored in FMan memory. An FMan block can support classification at a line rate of up to 12 Gbps, assuming a three-level tree search. Each tree level can use up to 256 bytes of table space; however, the total rules table size for all three tree levels cannot exceed 512 bytes. For example, if classification is configured to perform an exact match on a 4-byte field and there are 32 exact-match rules defined, 128 bytes of rules table space is required.

The FMan Controller treats each rules table as an ordered list of entries and takes action on the first match.

7.4.4.5.1.3.7 FMan KeyGen

The FMan's KeyGen sub block performs line-rate packet distribution onto a range of frame queues (FQs) based on hashes of inbound frame header fields extracted by the Parser. The KeyGen sub block holds 32 key-generation schemes in internal memory, with each scheme generating a different Frame Queue ID (FQID) and Policer Profile (PP). Further, the KeyGen supports an option for a post-hash index.

By applying the same KeyGen hash scheme to all arriving frames, all frames belonging to the same coarse-grained flow are enqueued to the same frame queue (FQ). Ensuring that frames belonging to the same coarse-grained flow are enqueued to a single FQ guarantees that the fine-grained flows within the coarse-grained flow are dequeued and processed by the cores in order. Figure 121. Algorithm for Calculating a Frame Queue ID on page 774 shows the steps in the KeyGen sub block's frame queue ID calculation.



Figure 121. Algorithm for Calculating a Frame Queue ID

7.4.4.5.1.3.8 FMan Policer

The FMan's Policer sub block implements a two rate, three color marker (trTCM) traffic policing algorithm. Using the Policer, you can implement differentiated services at line speed on the FMan's receive path or its offline parse path.

The Policer holds 256 policing profiles in internal memory and can apply these profiles to the outputs of the FMan Controller and KeyGen frame processing stages to limit enqueues to frame queues.

Policer features:

- Implements RFC2698 and RFC4115
- Supports three-color traffic marking: "Green", "Yellow", and "Red"
- Supports four traffic measurements:
  - CIR - Committed Information Rate
  - CBS - Committed Burst Size
  - PIR - Peak Information Rate
  - PBS - Peak Burst Size
- Supports color-aware and color-blind metering
- Supports policing based on packet count, as well as byte count
- Includes a quick drop mode for "Red" packets

- Stores up to 256 policing profiles in internal FMan memory
- Maintains traffic statistics on a per-profile basis

7.4.4.5.1.4 FMC Tool Features

The FMC Tool can analyze input NetPDL and NetPCD XML files that define the parse, classify, police, and distribute behavior your application requires. The tool can then:

- Passes this information directly to the FMan by calling the appropriate FMan driver API functions. (See FMC Tool - Runtime Environment Mode on page 775.)
- Generate C source files containing this information that you can include in your application. (See FMC Tool - Host Mode on page 776.)

In more detail, the FMC Tool can perform the tasks listed below. The particular actions taken depend upon your application's requirements.

- Define the protocol stack
- Define a soft header examination sequence
- Configure the Policer sub block
- Configure frame distribution by defining how frames are assigned to particular frame queues
- Call hardware drivers to execute the current configuration
- Directly configure the FMan by executing on a target running embedded Linux (See FMC Tool - Runtime Environment Mode on page 775.)
- Indirectly configure the FMan by executing on a Linux or Windows host by generating C source code that configures the FMan. You include this code in your application. (See FMC Tool - Host Mode on page 776.)

7.4.4.5.1.5 FMC Tool Components and Packaging

The FMC Tool package contains these files:

- Host version of FMC Tool for desktop versions of Linux and Windows
- FMC Tool application for embedded Linux
- NetPDL file containing a description of each standard network protocol that the FMan's Hard Parser supports. This file is named `hxs_pdl_v3.xml` and is in the directory `/etc/fmc/config/`.

NOTE

For detailed information on NetPDL, go to http://www.nbee.org/doku.php?id=netpdl:index.

For documentation of NXP's customized version of NetPDL, see NXP NetPDL Reference on page 795.

7.4.4.5.1.6 FMC Tool - Runtime Environment Mode

In runtime environment mode, you run the FMC Tool on a target board from the Linux command line, passing several configuration files as arguments. The tool then calls the FMan Driver API functions required to configure the FMan block as specified in the supplied files.

When used in this way, the FMC Tool *directly* configures the FMan. In more detail, the FMC Tool passes the configuration it finds in its input files (along with compiled Soft Parser firmware) to the FMan driver which, in turn, modifies the FMan's configuration.

Note: The FMC Tool does *not* support dynamic FMan configuration; you can use the tool to configure the FMan just once, typically at application initialization.

As Figure 122. FMC Tool, Runtime Environment - Input XML Files / FMan Driver API Calls on page 776shows, you pass these files to the FMC Tool as command-line arguments:

- Standard Protocol file - Optional; included in DPAA SDK; see Standard Protocol File on page 780 for more information.
- Custom Protocol file - Optional; user written; see Custom Protocol File on page 780 for more information.
- Policy file - Required; user written; see Policy file on page 781 for more information.
- Configuration file - Required; user written; see Configuration File on page 793 for more information.



Figure 122. FMC Tool, Runtime Environment - Input XML Files / FMan Driver API Calls

See FMC Tool Command-Line Arguments on page 778 for documentation of each of the tool's command-line arguments.

Note: You should configure the FMan before you enable your Rx/Tx ports to send/receive traffic. If you do not, the FMan uses the default Rx and default Tx frame queues.

7.4.4.5.1.7 FMC Tool - Host Mode

In addition to running on a target board, the FMC Tool can execute on a host computer running Linux or Windows. When run on a host, the FMC Tool accepts the same input files as in runtime environment mode.

However, in host mode, the FMC Tool generates C source code files. This code calls the FMan driver functions required to implement the rules defined in the supplied input files. You can compile and link these files to produce a standalone executable that you can run by itself, or you can add them to your application.

Note: The FMC Tool does not support dynamic FMan configuration; you can use the tool to configure the FMan just once, typically at application initialization.

As Figure 123. FMC Tool, Host Mode - Input XML Files / Generated C Source Code Files on page 777 shows, in host mode, the FMC Tool generates C source code files from the input files listed below. (See Host Mode Output - C Source Code Files on page 777 for more information.)

- Standard Protocol File - Optional; included in DPAA SDK; see Standard Protocol File on page 780 for more information.
- Custom Protocol File - Optional; user written; see Custom Protocol File on page 780 for more information.
- Policy File - Required; user written; see Policy file on page 781 for more information.

- Configuration File - Required; user written; see Configuration File on page 793 for more information.

You pass these files to the FMC Tool as command-line arguments.



Figure 123. FMC Tool, Host Mode - Input XML Files / Generated C Source Code Files

See FMC Tool Command-Line Arguments on page 778 for documentation of each of the tool's command-line arguments.

7.4.4.5.1.7.1 Host Mode Output - C Source Code Files

When run in host mode, the FMC Tool generates C language source code files that make calls to FMan Driver API functions. These calls implement the behavior defined in the Configuration file, Policy file, and (optionally) Custom Protocol file passed to the tool from the command line. Typically, you include these source files in your project, so they are compiled and linked into your application binary. As a result, when you run your application, it automatically sets up the FMan to behave as required.

In more detail:

- When you supply a Policy file and a Configuration file, the tool generates a single source code file named "fmc_config_data.c".
- When you supply a Policy file, a Configuration file, *and* a Custom Protocol file, the tool generates two source code files: "fmc_config_data.c" and "softparse.h".

Contents of fmc_config_data.c

- #include software parser configuration "softparse.h" at the top of the file
- Initialization of FMC model structure 'fmc_model_t' with configuration data - This structure represents the data model for FMan hardware configuration according to input files

Using fmc_config_data.c

- FMC model structure must be used together with FMC model definition and FMC executer: 'fmc.h' and 'fmc_exec.c' files - These file are available in FMC source files location
- FMC model definition contains 'fmc_model' structure definition - This structure represents the FMC configuration model
- FMC executer contains 'fmc_execute' routine - This function configures the FMan hardware to behave as specified in the input files

Usage options:

- Compile and link these files together ('fmc_config_data.c', 'fmc.h', 'fmc_exec.c') and generate a standalone binary and run this binary to configure the FMan - In this case you must add a main() function that calls fmc_execute()
- Have your application call fmc_execute() - In this case you don't need to add a main() function

Contents of softparse.h

- Contains compiled firmware that controls the FMan sub blocks involved in parsing a custom protocol header
- Defines parameters such as code size, protocol to attach, and download base address

Using softparse.h - Automatically included in fmc_config.c if you pass the FMC Tool a Custom Protocol file

Note: You should configure the FMan before you enable your Rx/Tx ports to send/receive traffic. If you do not, the FMan uses the default Rx and default Tx frame queues.

7.4.4.5.1.8 FMC Tool Command-Line Arguments

The table below lists and describes the FMC Tool's command-line arguments.

Table 132. FMC Tool Command-Line Arguments

Command-Line Argument Syntax ***(Both the verbose and abbreviated command forms are shown)***	**Description**
-d <pdl_file>, --pdl <pdl_file>	Path to and name of the Standard Protocol file. *(Optional)* You can use a full path or a relative path. See Standard Protocol File on page 780 for more information.
-p <pcd_file>, --pcd <pcd_file>	Path to and name of a Policy file. *(Required unless '--sp_only' is used)* You can use a full path or a relative path. See Policy file on page 781 for more information.
-c <data_file>, --config <data_file>	Path to and name of the Configuration file. *(Required unless '--sp_only' is used)* You can use a full path or a relative path. See Configuration File on page 793 for more information.
-s <custom_protocol_file>, --custom_protocol <custom_protocol_file>	Path to and name of the Custom Protocol file. *(Optional unless the '--sp_only' flag is used, in which case, this Custom Protocol file name is required.)* You can use a full path or a relative path. See Custom Protocol File on page 780 for more information.
-f --force	Applies the new configuration, without regard for the temporary information the FMC tool stored about the previous configuration. **(Optional)** When used in runtime environment mode (that is, on the target), the FMC Tool stores information about the previous configuration that the tool uses to properly unroll this configuration before applying the new one. There are cases (such as a non-destructive reboot) when such unrolling is not desirable. In such cases, use the --force switch.

Table continues on the next page...

Table 132. FMC Tool Command-Line Arguments (continued)

Command-Line Argument Syntax ***(Both the verbose and abbreviated command forms are shown)***	**Description**
-a, --apply	Apply the supplied configuration to the FMan rather than generating C source code. *(Optional; valid only when FMC Tool is executed in runtime environment)*
--sp_only	Perform Soft Parser processing only. When this argument is supplied, the FMC Tool compiles just the Custom Protocol file, generates the file softparse.h, and exits. The file softparse.h contains C source code and custom protocol offsets. The tool creates softparse.h in the path from which the FMC Tool was executed. *(Optional)*
-w	Do not report warnings. *(Optional)*
--version	Display version information, then exit. *(Optional)*
-h, --help	Display usage information, then exit. *(Optional)*

7.4.4.5.1.9 The NetPDL and NetPCD XML Markup Languages

The Network Protocol Description Language (NetPDL) is an XML dialect that defines elements for describing protocols from OSI layer 2 to OSI layer 7. (For more information on NetPDL, see http://www.nbee.org/doku.php?id=netpdl:index.)

NXP uses NetPDL to define the standard protocols that are parsed by the FMan's Hard Parser. You cannot change these protocol descriptions. However, the SDK includes a Standard Protocol file that you can use as a reference.

In addition, you can use NetPDL (with slight semantic and syntactic differences) to define custom protocols that are parsed by the FMan's Soft Parser. This feature allows the FMan to handle any protocol that exists or that you define yourself.

Finally, NXP has extended NetPDL to create a language called NetPCD. You use the elements and attributes of NetPCD to define FMan parse, classify, police, and distribute behavior. The processing thus defined determines how frames move from block to block of the FMan.

The FMC Tool accepts files in NetPCD and NetPDL format as input.

7.4.4.5.1.10 Protocol files

For a protocol to be recognized by the FMC Tool, the protocol must be defined in one of two ways:

1. As a standard protocol within the Standard Protocol file (included in the SDK)
2. As a custom protocol within the Custom Protocol file.

Each file type is described in the sections that follow.

7.4.4.5.1.10.1 Standard Protocol File

The DPAA SDK includes a file called the Standard Protocol file. This file contains NetPDL (Network Protocol Description Language) markup that defines the fields in each standard protocol header that the FMan's Hard Parser can handle. In addition, for each standard protocol, the file includes NetPDL statements that define actions for the Hard Parser to take upon encountering an inbound instance of this protocol.

The Standard Protocol file is for the FMan's internal use only; you must therefore not change it. However, to write a Custom Protocol file and/or a Policy file, you sometimes need information the Standard Protocol file contains, such as the names of fields in a protocol's header.

For this reason, the SDK includes a copy of the Standard Protocol file in this directory: /etc/fmc/config/hxs_pdl_v3.xml.

To give you an idea what the markup in this file looks like, Figure 124. Standard Protocol File NetPDL that Defines the Ethernet Protocol on page 780 shows the NetPDL in the Standard Protocol file that defines the Ethernet protocol.



Figure 124. Standard Protocol File NetPDL that Defines the Ethernet Protocol

See the Standard Protocol File - Excerpt on page 847 topic to see a larger portion of the Standard Protocol file.

7.4.4.5.1.10.2 Custom Protocol File

(Largely, content from the Soft Parser Tool MS Word document as well as former "User Defined Protocols" sections.)

Preferred name: "FMC custom protocol file" Also previously called... - "custom protocol file"

The FMan's Hard Parser has built-in capability to handle a set of widely used, standard protocols, such as IPv4. The FMan also has a Soft Parser, which has the ability to process custom protocols.

Of course, for the Soft Parser to recognize a custom protocol, you must first provide a definition of this protocol. To do this, you create a Custom Protocol file, which consists of NetPDL markup that defines the fields in a custom protocol's header along with the actions you want the Soft Parser to take upon these fields. You then pass this file to the FMC Tool, which compiles it and passes the result to the FMan.

Note: Some elements in the NetPDL language are relevant only if used with a protocol analysis tool. The FMC Tool does *not* support these elements; instead, the tool supports only those elements that are applicable to the FMan block. Further,

although it is based on NetPDL, the markup for a custom protocol does not strictly follow NetPDL rules. As a result, it is highly recommended that the you become familiar with the NXP NetPDL Reference on page 795 topic, which fully documents the custom version of NetPDL used in custom protocol definitions.

See Custom Protocol File - GTP Protocol on page 854, for an example of a custom protocol definition file containing XML that defines the GPRS Tunneling Protocol (GTP).

Note: If your application does not use a custom protocol, you do not have to create a Custom Protocol file. Further, if your application uses *multiple* custom protocols, you can (and must) define them in a single Custom Protocol file; you can pass just one Custom Protocol file to the FMC Tool.

The general structure of a Custom Protocol file is shown below.

```
<netpdl> <!-- only one instance -->
   <protocol> <!-- one or more instances -->

      <format> <!-- only one instance -->
         <fields> <!-- only one instance -->
            <field/> <!-- one or more instances -->
         </fields>
      </format>

      <execute-code> <!-- zero or one instance -->
         <before> <!-- zero or one instance -->
         </before>

         <after> <!-- zero or one instance -->
         </after>
      </execute-code>

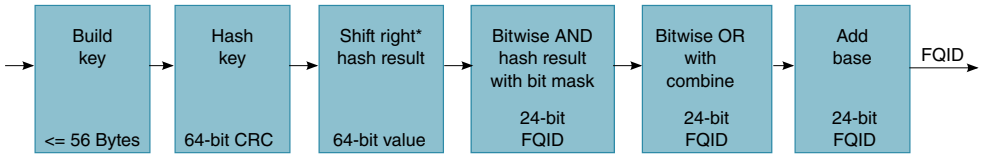
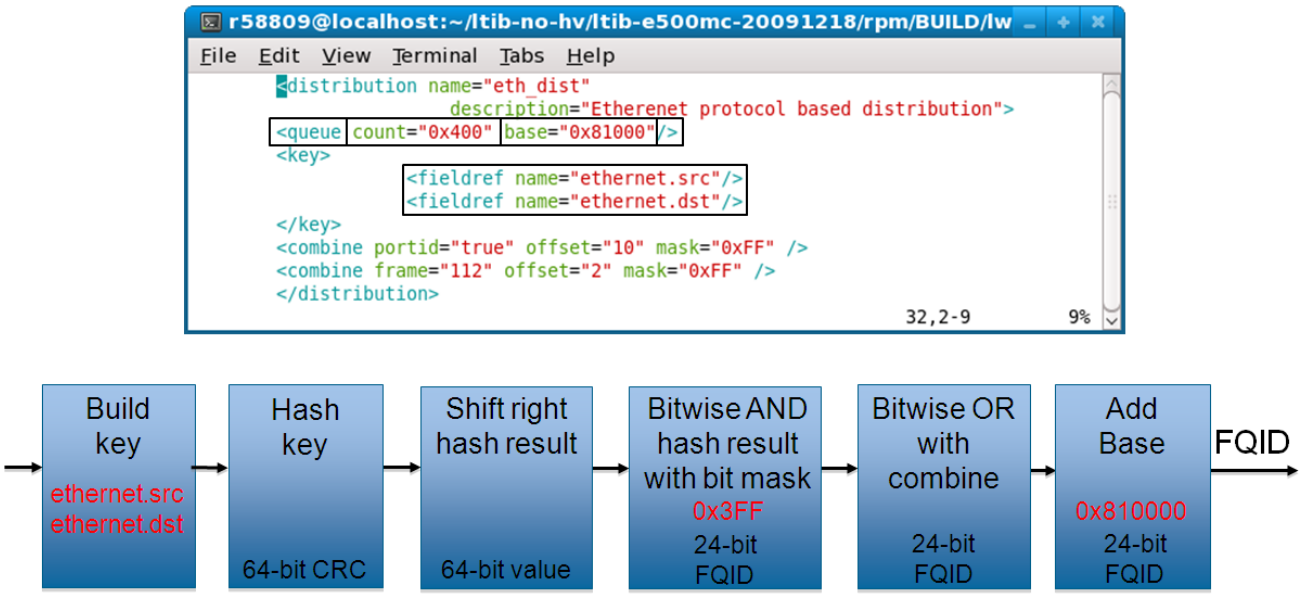
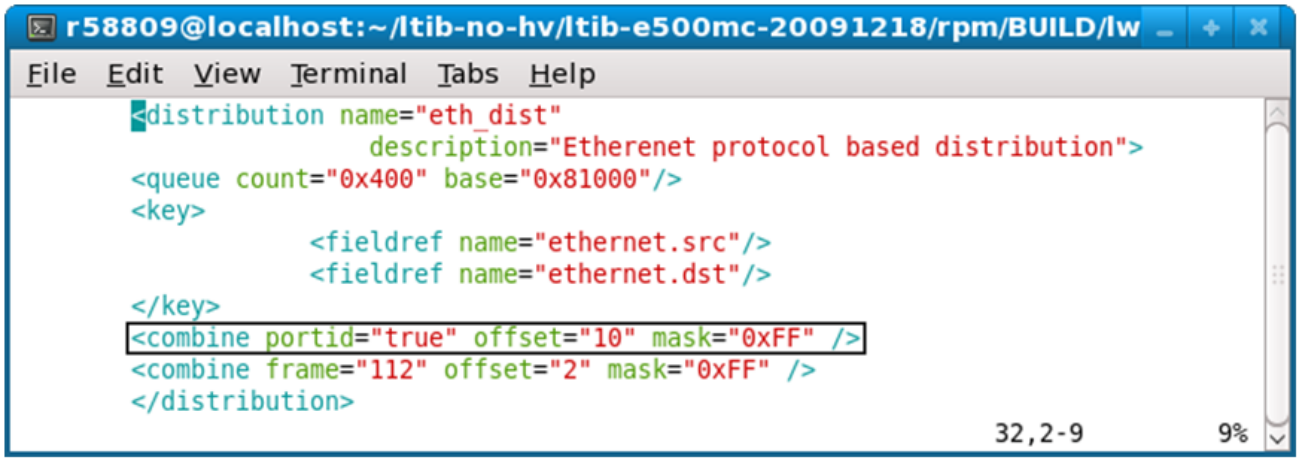
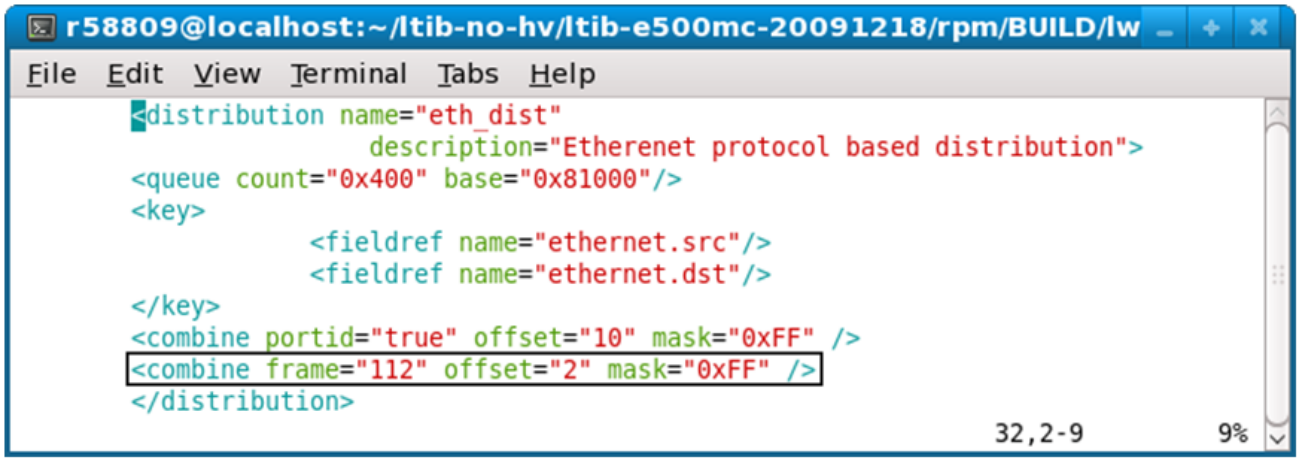
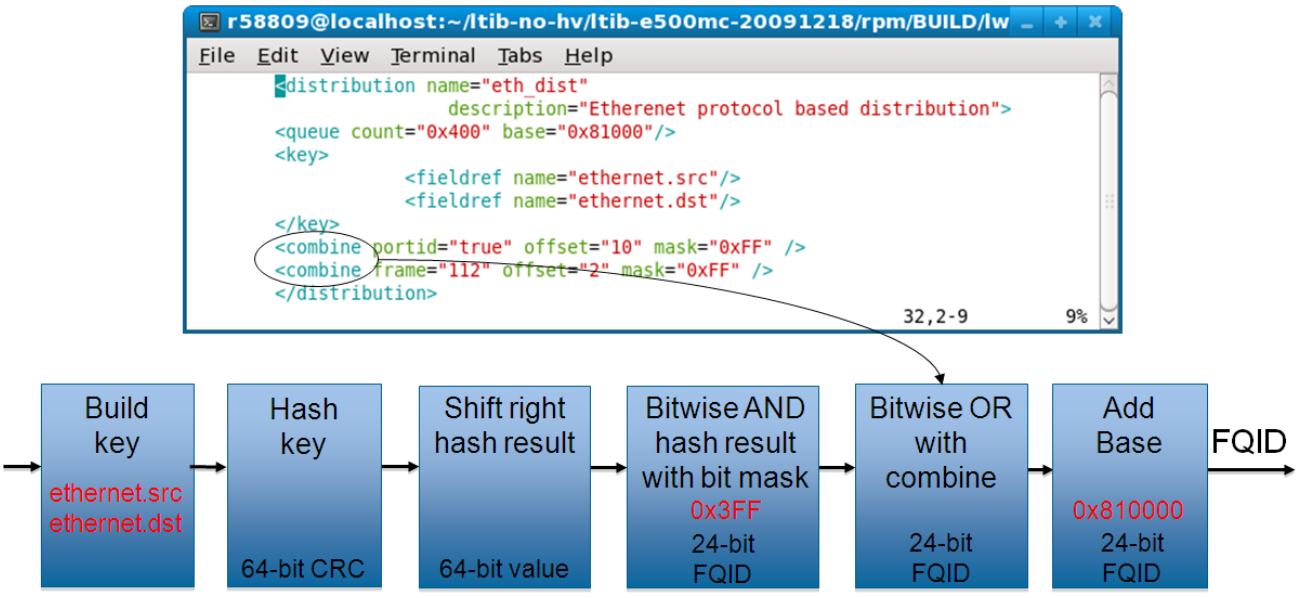
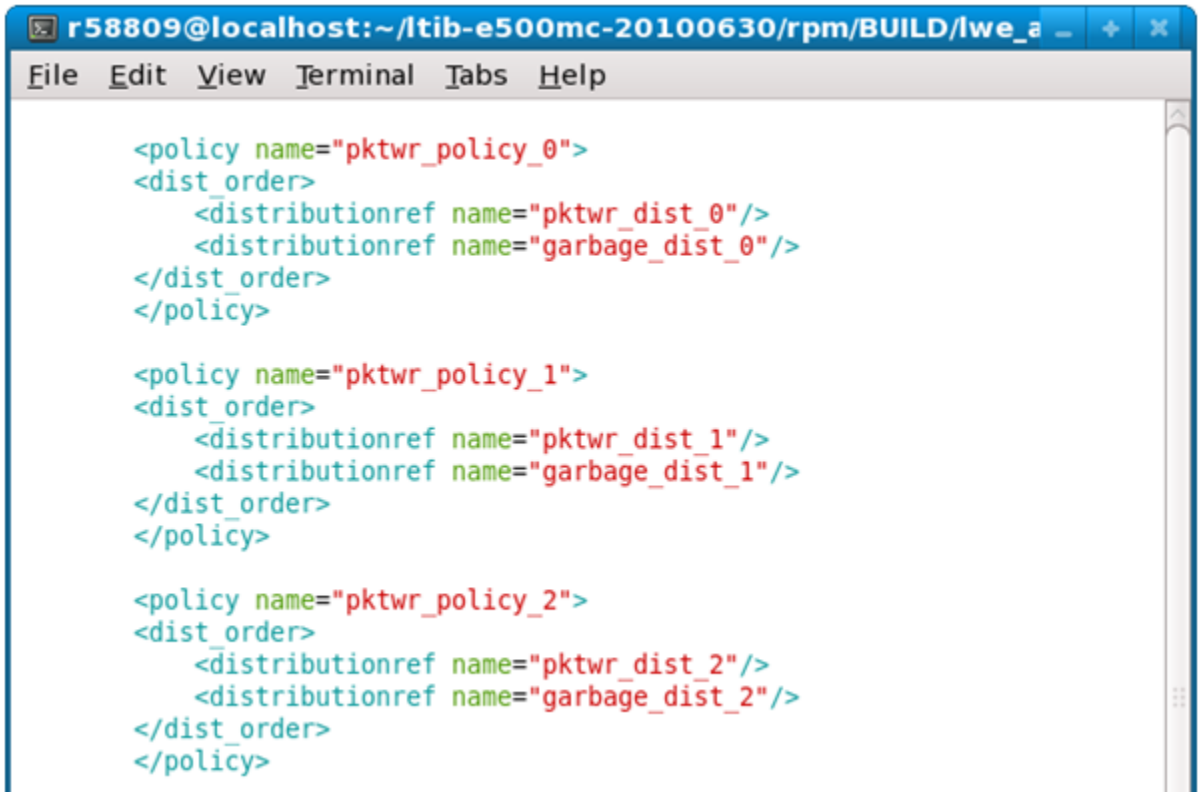
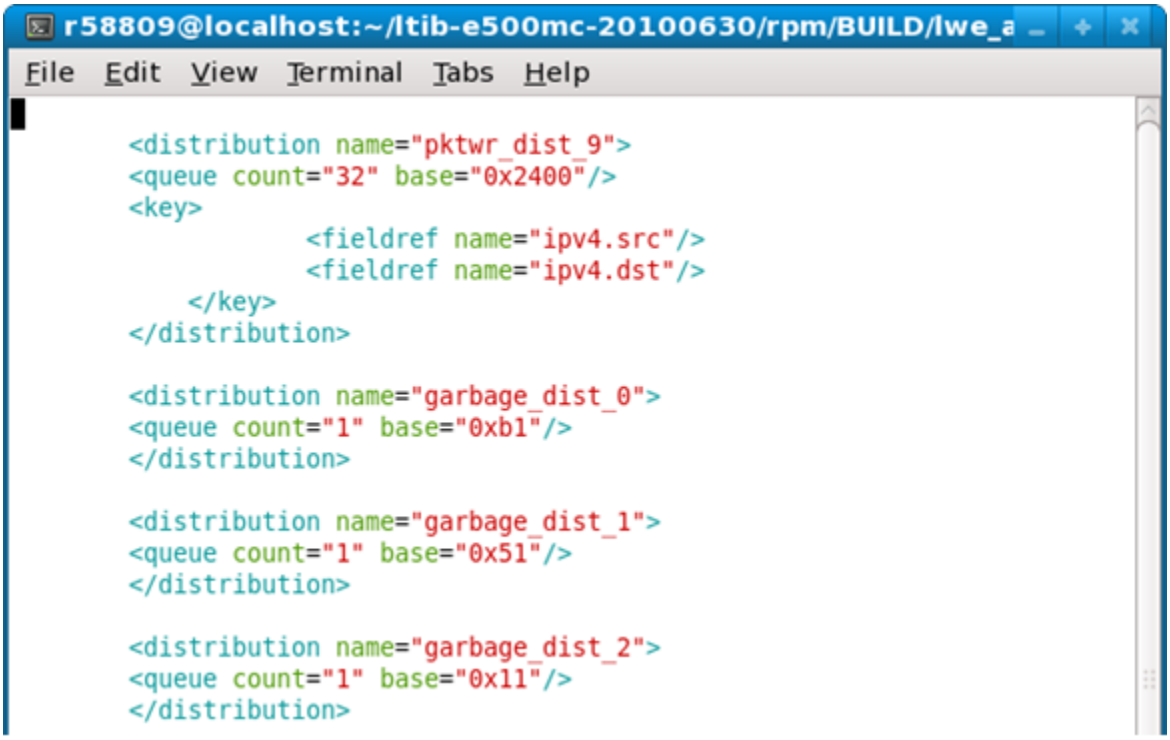
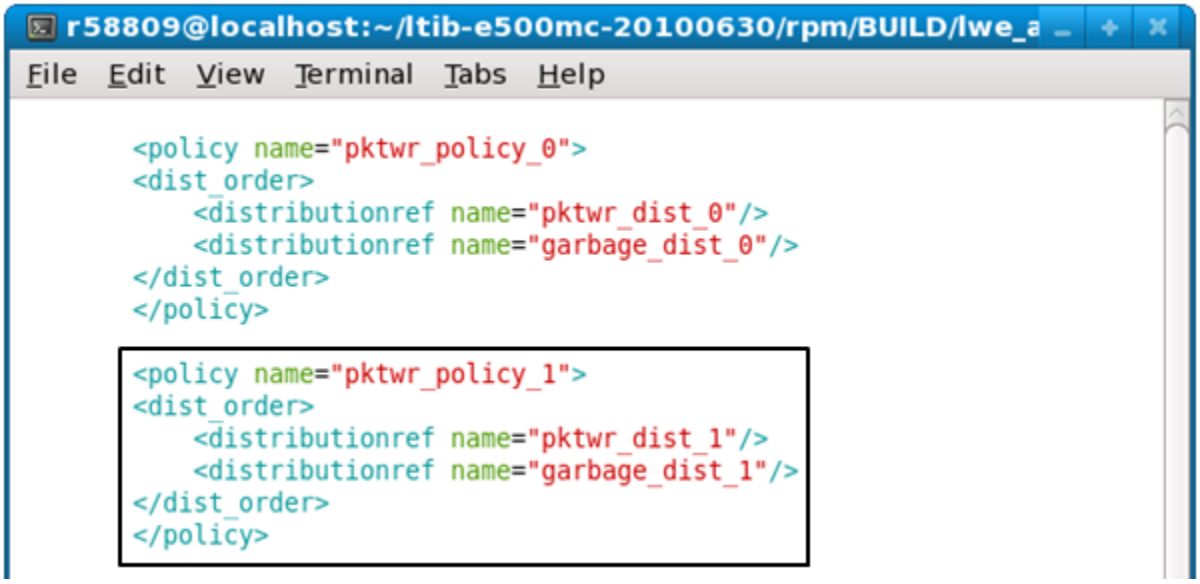
   </protocol>
</netpdl>
```

7.4.4.5.1.11 Policy file

The policy file defines how each inbound frame is parsed, classified, policed, and distributed by the various FMan sub blocks.

A policy file consists of NetPCD markup, where NetPCD is NXP's extension to NetPDL, an XML markup language for describing networking protocols. The elements and attributes of NetPCD let you define the parse, classification, policing, and distribution behavior your application requires. See NetPCD Reference on page 817 for documentation of each NetPCD element and its attributes.

A Policy file can have these sections:

- Distribution (required) - Contains one or more distribution definitions, each of which:
  - Specifies the protocol(s) a frame must contain to match the distribution
  - Defines how to handle matching frames
- Policy - (required) - Contains one or more policy definitions, each of which:
  - Is associated with an FMan port
  - Contains a prioritized list of distributions
- Classification (optional) - Contains one or more classification blocks, each of which:
  - Defines key/value/action tuples, which the FMan's Controller sub block stores in a lookup table
  - Compares the specified fields in the current frame header to each value in this table and, upon a match, takes the specified action
- Policer (optional) - Contains up to 256 policer profiles, each of which can be used to:

- Take action upon frames without regard to traffic flow rate
- Take action upon frames based on the RFC-2698 two-rate, three-color policing scheme
- Take action upon frames based on the RFC-4115 two-rate, three-color, differentiated services scheme

Note: When you run the FMC Tool, you must pass it a Policy file or the '--sp_only' flag. Otherwise, the program will exit and print an error message.

Figure 125. High-level Structure of a Policy File

```
<netpcd> <!-- only one instance -->
      <distribution> <!-- one or more instances -->
      </distribution>

      <policy> <!-- one or more instances -->
            <dist_order> <!-- one instance -->
                  <distributionref/> <!-- one or more instances -->
            </dist_order>
      </policy>

      <classification> <!-- optional, may have more than one instance -->
      </classification>

      <policer> <!-- optional, may have more than one instance -->
      </policer>
</netpcd>
```

7.4.4.5.1.11.1 Distribution Section

The Distribution *section* of the Policy file contains one or more 'distribution' *elements*. While 'distribution' elements can appear anywhere in the Policy file, they often appear at the top of the file.

Typically a 'distribution' contains child elements that define:

- Frame match rules
  - These rules define the conditions an inbound frame must meet to match (and therefore be handled by) this distribution
  - Use the 'protocols' element and/or the 'key' element to define match rules
- Frame handling rules
  - These rules determine what a distribution does with matching frames
  - Use the 'queue' and 'key' elements to hash frames, so they are evenly spread over a range of frame queues
  - Use the 'action' element to pass the frame to another element in the Policy file for further processing

Figure 126. Example Distribution Elements

```
<!-- distribution that matches all frames containing an IPv4 header -->
<!-- hashes these frames, so they are spread evenly over 32 frame queues -->
<distribution name="hash_ipv4_src_dst_dist0">
  <!-- frame match rule -->
  <key>
    <fieldref name="ipv4.src"/>
    <fieldref name="ipv4.dst"/>
  </key>

  <!-- frame handling rule -->
  <queue count="32" base="0x400"/>
</distribution>
```

```
<!-- distribution that matches frames containing Eth/VLAN/IPv4/UDP/GTP headers -->
<!-- passes all matching frames to the "dl_vlan_clasifif" classification element -->
<distribution name="dl_eth_vlan_ipv4_udp_gtp_dist">
  <!-- frame match rule -->
  <protocols>
    <protocolref name="ethernet"/>
    <protocolref name="vlan"/>
    <protocolref name="ipv4"/>
    <protocolref name="udp"/>
    <!--shim1 is custom protocol defined for GTP -->
    <protocolref name="shim1"/>
  </protocols>

  <!-- frame handling rule
  <action type="classification" name="dl_vlan_classif"/>
</distribution>
```

See The distribution element on page 819 for complete documentation of this element.

Evenly Distributing Frames over a Range of Frame Queues

One frequent use of the 'distribution' element is to distribute frames evenly over a range of frame queues. If each available core is configured to pull from the same number of queues in the range, this even spreading balances the work each core must perform.

In this scenario, the FMan's KeyGen sub block uses values in the frame's header and in the child elements of the distribution as inputs to a hash algorithm that generates a 24-bit FQID within a range of FQIDs. The KeyGen sub block then places the frame on the frame queue identified by this FQID.

Here is the KeyGen's algorithm for generating a FQID:

1. Extract and concatenate the protocol header fields specified by the 'key' child element
2. Hash the resulting string to a 64-bit CRC
3. Shift the CRC right by the number of bits specified in the 'shift' attribute of the 'key' element to move the desired bits to the 24 least significant bit positions
4. Zero-extend the bit mask specified by the 'queue' child element ('count' attribute – 1) to 24 bits
5. Bitwise AND the result with the shifted CRC
6. Bitwise OR the result with the value specified by the 'combine' child element - repeat for each 'combine' element
7. Add the result to the base FQID specified by the 'base' attribute of the 'queue' child element

Figure 127. KeyGen Algorithm for FQID Calculation on page 783 shows the algorithm the KeyGen sub block uses to calculate a FQID.



Figure 127. KeyGen Algorithm for FQID Calculation

* The 'key' element has an optional 'shift' attribute whose value defines the number of bits by which the hash result is right shifted. The default value for the shift attribute is zero.

Example KeyGen FQID Calculation

The series of figures that follow shows which child elements and attributes of a distribution block the KeyGen sub block uses in its FQID calculation.

Figure 128. FQID Calculation - Elements/Attributes Used for Key, Bit Mask, and Base FQID on page 784 shows where in the KeyGen sub block gets the inputs for the hash, shift right, bitwise AND, and "add base" parts of its FQID calculation.



Figure 128. FQID Calculation - Elements/Attributes Used for Key, Bit Mask, and Base FQID

Figure 129. FQID Calculation - A 'combine' Element that Uses the 'portid' Attribute on page 784shows a 'combine' element that includes a 'portid' attribute that is set to "true". In addition, the element's 'offset' attribute is "10", and its 'mask' is "0xFF". This markup instructs the KeyGen sub block to perform the "bitwise OR" part of the FQID calculation. In more detail, for this markup, the KeyGen does these things:

- Bitwise ANDs the 8-bit logical port ID (defined in the Configuration file) of the port on which the current frame arrived with the 8-bit mask in the 'combine' element.
- Bitwise ORs (inserts) the 8-bit result at the specified offset (10 bits) within the 24-bit FQID (where offset 0 signifies the FQID's most significant bit).

Note: Each FMan port can be assigned an 8-bit logical port ID by adding markup to the Configuration file. To do this, assign an 8-bit value to the 'portid' attribute of each 'port' element to which you want to assign a logical port ID. The Hard Parser puts this value (if defined) in the parse results array, where the a KeyGen sub block can get it.



Figure 129. FQID Calculation - A 'combine' Element that Uses the 'portid' Attribute

Figure 130. FQID Calculation - A 'combine' Element that Uses the 'frame' Attribute on page 785 shows a 'combine' element that includes a 'frame' attribute. This markup instructs the KeyGen sub block to:

- Get the 8 bits at offset 112 in the current frame header.
- Bitwise AND this value with the 8-bit mask (0xFF) specified in the 'combine' element
- Bitwise OR (insert) the 8-bit result at the specified offset within the 24-bit FQID (where offset 0 signifies the FQID's most significant bit).

Note: The value of the 'frame' attribute is an offset (in bits) from beginning of the current frame. The KeyGen sub block gets the byte at this offset for its FQID calculation. The value of 'frame' must be divisible by 8, so the bit it references is on a byte boundary.



Figure 130. FQID Calculation - A 'combine' Element that Uses the 'frame' Attribute

Finally, Figure 131. FQID Calculation - combine Elements Used in Bitwise OR on page 785 shows where the KeyGen sub block plugs the values from each of the combine elements into the bitwise OR part of the FQID calculation.



Figure 131. FQID Calculation - combine Elements Used in Bitwise OR

FQID Formula

```
FQID[0:23] = (Shifted Hash Key[0:23] & Hash Mask) |
             Data0[0:23] | Data1[0:23] | … | Data7[0:23] |
             FQID Base Address
```

In sum, use the child elements/attributes of the 'distribution' element to provide the values on the right side of the FQID equation.

7.4.4.5.1.11.2 Policy Section

The Policy *section* of the Policy *file* consists of one or more 'policy' *elements*. While 'policy' elements can appear anywhere in the Policy file, they typically follow the last 'distribution' element in the file.

Each 'policy' element defines a set of *candidate* distributions that the FMan can apply to inbound frames. The particular distribution the FMan applies to a given frame depends on these factors:

- The position of each distribution in the 'policy' element's distribution order list
- The definition of each of these distributions

Candidate distributions are listed in *priority* order. As a result, if two or more distributions in the list match the current inbound frame, the FMan applies the first matching distribution because this distribution has higher priority.

How does the FMan know which policy (that is, which prioritized list of distributions) to apply to the traffic received on a particular Ethernet port? The Configuration file provides the connection.

In a Configuration file, you must enter one 'port' element for each FMan port your application uses. Further, the port element has a required attribute - the 'policy' attribute - whose value must match the name of one of the policy elements in the Policy file, thereby defining the policy (that is, the ordered list of distributions) that the FMan will apply to all traffic received on a port. In sum, the value of a port element's policy attribute in the *Configuration* file ties the port identified by this element to a policy element in the *Policy* file.

In a Configuration file:

- A port can be assigned a single policy
- Multiple ports can be assigned the same policy
- A port can have just one active policy at a time

Typically, you assign one policy to each port your application uses.

Example 1 - Simple Use of the Policy Element

Configuration File

```
<!-- The port element assigns the dl_policy policy to the 10 Gbps port of FMan 1 -->
<!-- Policy dl_policy is defined in the Policy file - see next code snippet -->
<cfgdata>
  <config>
    <engine name="fm1">
      <port type="10G" number="0" policy="dl_policy"/>
    </engine>
  </config>
</cfgdata>
```

Policy File

```
<!-- A policy element that defines how to apply two distributions -->
<!-- These distributions are defined elsewhere in the Policy file -->
```

```
<!-- This policy is assigned to an Ethernet port by the Configuration file above -->
<policy name="dl_policy">
      <dist_order>
            <distributionref name="dl_eth_vlan_ipv4_udp_gtp_dist"/>
            <distributionref name="garbage_dist"/>
      </dist_order>
</policy>
```

In the example above, the Configuration file assigns the policy named 'dl_policy' to the 10 Gbps port of the p4080 chip's second FMan (fm1). As a result, the FMan first tries to match each frame that arrives on this port to the 'dl_eth_vlan_ipv4_udp_gtp_dist' distribution since it appears first in the 'policy' element's distribution order list. Whether the frame matches depends on the definition of the 'dl_eth_vlan_ipv4_udp_gtp_dist' distribution, which is not shown. If the frame matches, it is handled according to the rules this distribution defines. If the frame does not match, the FMan next compares it to the 'garbage_dist' distribution since it appears second in the distribution order list. Because of this distribution's definition (also not shown), it matches all frames, thereby guaranteeing that every frame is handled in one way or the other.

See The policy element on page 818 for complete documentation of this element.

Example 2 - More Complex Use of the Policy Element

Figure 132. More Complex Policy File - 1 on page 787 shows the Policy file from the pktwire application. This application requires a more complex use of policies and distributions than shown in the previous example.

This Policy file defines ten 'policy' elements - pktwr_policy_0, pktwr_policy_1, … pktwr_policy_9 - some of which are shown in the figure.

A Configuration file (not shown) assigns each of these policies to one of the p4080's ten FMan ports - five on the first FMan (fm0) and five on the second FMan (fm1).

Note: Not all QorIQ devices have two FMans. Nor does every FMan have five Ethernet ports. See the reference manual for your QorIQ device to determine the number of FMans and FMan ports this device supports.



Figure 132. More Complex Policy File - 1

The Policy file also defines ten distributions - pktwr_dist_0, pktwr_dist_1, … pktwr_dist_9 - some of which are shown in Figure 133. More Complex Policy File - 2 on page 788.

As mentioned above, each of these distributions is assigned to a policy which, in turn, is assigned to a port. A frame "matches" the distribution assigned to the port on which the frame arrived if its header contains both the ipv4.src and ipv4.dst fields.

For each frame that matches, the KeyGen sub block computes a hash result using the concatenation of the ipv4.src and ipv4.dist fields as the hash key. The KeyGen sub block then uses the hash result to compute a FQID. (See the Distribution Section on page 782 topic for detailed coverage of the KeyGen's FQID calculation algorithm.)

The resulting FQID is in the range specified by the 'queue' element. For example, for distribution “pktwr_dist_0”, the resulting FQID will be in range 0x2800 – 0x281F.

Figure 133. More Complex Policy File - 2

The Policy file also defines ten distributions - garbage_dist_0, garbage_dist_1, … garbage_dist_9 - some of which are shown in Figure 134. More Complex Policy File - 3 on page 789.

Note that these distributions do not have a 'key' element. As a result, all frames “match” these distributions. For 'garbage_dist_0', the resulting FQID is always 0xb1 since the queue element specifies just one frame queue and the base FQID value is 0xb1.



Figure 134. More Complex Policy File - 3

Let's say that an FMan port is tied to policy 'pktwr_policy_1' - highlighted in Figure 135. More Complex Policy File - 4 on page 790.

This policy instructs the FMan to first attempt to distribute frames arriving on this port using the 'pktwr_dist_1' distribution. If the current frame does not include the ipv4.src and ipv4.dst fields, the policy instructs the FMan to try the next distribution in the policy's distribution order list.

In this example, the next distribution is "garbage_dist_1" which, due to the absence of a 'key' element, matches *all* frames and enqueues them to the single frame queue defined by the 'count' and 'base' attributes of its queue element.

Note: It is common for the last distribution in a distribution order list to be a "catch all", like the default case in a C switch statement; however, this is not a requirement.



Figure 135. More Complex Policy File - 4

7.4.4.5.1.11.3 Classification Section

The Classification section of the Policy file is optional. Use it to specify exact match frame classification.

A classification specifies the action to perform on a frame when the values of the specified fields in a frame's protocol header match a predefined value. You can specify as many predefined value/action pairs as desired, as well as a default action.

A classification starts with a 'classification' element, which is a container for these child elements:

- A 'key' element that defines the header fields (in protocol.field form) to use in the exact match operation
- One or more 'entry' elements, each of which defines a value to which the specified fields are compared and a 'queue' and/or 'action' element that defines what to do with the frame upon a match
- An optional 'action' element that defines the default action to take if none of the exact match conditions is met

The FMC Tool uses the information in these child elements to populate the FMan Controller's rules table. At runtime, the Controller uses this information to extract the specified fields from the specified protocol header, compare these fields to the specified values and, upon a match, take the specified action.

See The classification element on page 828 for complete documentation of this element.

Example

The example below shows a Policy file containing a 'classification' element.

The 'policy' element named 'policy_0' lists two distributions to try, 'udp_dist' and 'non_udp_dist'.

Note: For a classification block to be applied to a frame, the frame must first match a distribution that transfers control to this classification via an 'action' element. In other words, the "source engine" of the Classifier is always a 'distribution' element.

The 'udp_classif' classification element specifies an exact-match lookup on the ipv4.dst field. If this field's value is:

- 0xC0A81402, the frame is placed on the queue whose FQID is 0x200
- 0xC0A81404, the frame is placed on the queue whose FQID is 0x400
- 0xC0A81406, the frame is placed on the queue whose FQID is 0x600
- 0xC0A81408, the frame is placed on the queue whose FQID is 0x800

Otherwise, the 'action' element passes the frame to the 'unknown_dist' distribution for handling.

```
description="Course Classification configuration">
<policy name="policy_0">
        <dist_order>
                <distributionref name="udp_dist"/>
                <distributionref name="non_udp_dist"/>
        </dist_order>
</policy>

<distribution name="udp_dist">
    <protocols>
        <protocolref name="udp"/>
    </protocols>
    <action type="classified" name="udp_classif"/>
</distribution>

<classification name="udp_classif">
    <key>
                <fieldref name="ipv4.dst">
    </key>
    <entry>
                <data>0xC0A81402</data>
                <queue base="0x200"/>
    </entry>
    <entry>
                <data>0xC0A81404</data>
                <queue base="0x400"/>
    </entry>
    <entry>
                <data>0xC0A81406</data>
                <queue base="0x600"/>
    </entry>
    <entry>
                <data>0xC0A81408</data>
                <queue base="0x800"/>
    </entry>
    <action type="distribution" condition="on-miss" name="unknown_dist"/>
<classification>
"cc_policy.xml" 108 lines --61%--
```

7.4.4.5.1.11.4 Policer Section

The Policer section of the Policy file is optional.

If used, the section consists of up to 256 policer profiles. Each profile starts with a 'policer' element, which is a container for various child elements with which you implement a particular policing behavior.

Each profile works in one of these modes:

- Pass-through – Policer performs no traffic metering
- RFC-2698 - Policer employs a two-rate, three-color marker scheme
- RFC-4115 - Policer employs a differentiated service, two-rate, three-color marker scheme that efficiently handles in-profile traffic

Each of these modes can be configured to be color-aware or color-blind.

For RFC-2698 and RFC-4115 modes, you must specify these values:

- unit, the unit to be used for the following numeric parameters. Valid values for unit are "packet" and "byte."
- CIR, Committed Information Rate[7]
- CBS, Committed Burst Size[8]
- PIR, Peak Information Rate[9]
- PBS, Peak Burst Size[10]

In all three modes, you can specify the next invoked action (NIA) for each color result (drop the frame, proceed to the specified distribution, etc.)

Example 1 - Policer Markup for RFC2698 Mode

```
<policer name="policer2">
  <algorithm>rfc2698</algorithm>

  <color_mode>color_aware</color_mode>

  <CIR>12000</CIR>
  <EIR>34000</EIR>
  <CBS>56000</CBS>
  <EBS>78000</EBS>

  <unit>byte</unit>

  <action condition="on-green"  type="distribution" name="green_dist"/>
  <action condition="on-yellow" type="distribution" name="yellow_dist"/>
  <action condition="on-red"    type="drop"/>
</policer>
```

Example 2 - Policer Markup for Pass-through Mode

```
<policer name="vlan_congestion_control_green">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>green</default_color>

  <action condition="on-green" type="distribution name="default_dist"/>
</policer>

<policer name="vlan_congestion_control_yellow">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>yellow</default_color>

  <action condition="on-yellow" type="drop"/>
</policer>
```

[7] If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second
[8] If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.
[9] If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second
[10] If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.

```
<policer name="vlan_congestion_control_red">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>red</default_color>

  <action condition="on-red" type="drop"/>
</policer>
```

7.4.4.5.1.12 Configuration File

The Configuration file contains markup that defines the FMan instances (for devices with more than one FMan) and ports that are being used.

In addition, the Configuration file "connects" each port to the parse, classification, policing, and distribution rules defined in the Policy file. How? Each 'port' element in the Configuration file has a 'policy' attribute whose value must be the name of one of the 'policy' elements in the Policy file. This information tells the FMan which distributions to compare to each frame received on a given port.

Figure 136. Example Configuration File on page 793 shows the Configuration file's elements, attributes, and element hierarchy.

Note these element and attribute requirements:

- Valid engine names are "fm0" or "fm1"
- Valid values for the port type attribute are:
  - "MAC" (1/10 Gbps Ethernet port)
- Port numbering corresponds to hardware port number (as in dts) for each port.
- The value of the 'policy' attribute of a 'port' element must match the name of a 'policy' element in the Policy file.
- portid attribute (optional) - One byte numeric value that is attached to the port and that can be used in the 'distribution' and 'combine' elements of the Policy file.

The Configuration file's general structure is shown below.

Figure 136. Example Configuration File on page 793 shows an example configuration file. It uses the optional 'portid' attribute for the 1 Gbps ports.

Figure 136. Example Configuration File

```
<cfgdata>
  <config>
    <engine name="fm0">
      <port type="MAC" number="1" policy="ipv4_policy"/>
      <port type="MAC" number="2" policy="ipv4_policy" portid="0x96"/>
      <port type="MAC" number="3" policy="ipv4_policy" portid="0x97"/>
      <port type="MAC" number="4" policy="ipv4_policy" portid="0x97"/>
    </engine>
  </config>
</cfgdata>
```

Virtual Storage Profiles

The element 'vsp' (Virtual Storage Profile) is implemented in FMC as a standalone entity or can be defined directly in the element that uses it. The element 'vsp'can be used inside distributions, classification and entries (both classification and

replicator). When used directly in the ‘classification’ element (not in ‘entry’) it counts for the on-miss action. If the 'action' of the 'entry' or on-miss goes to another 'classification' or 'replicator' the 'vsp' is ignored.

Table 133. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the virtual storage profile inside the elements that are using it.
type	optional	The type of the VSP. Values: • direct – (default) the relative profile ID is selected directly by the ‘base’ attribute. • indirect – the relative profile ID is selected base on the attributes **fqshift, vspoffset**, and **vspcount** can be used only in **distribution.**
base	required for direct.	--
fqshift	required for indirect.	Shift of KeyGen results without the FQID base.
vspoffset	optional for indirect	OR of KeyGen results without the FQID base; should indicate the storage profile offset within the port's storage profiles window.
vspcount	optional for indirect	Range of profiles starting at base.

VSP examples (standalone, defined in element, direct/indirect): The action targets of the entry are restricted to:

```
<vsp name = "storage01" base = "6"/>
<vsp name = "storage02" type = "indirect" fqshift="2" vspoffset="3"        vspcount="8"/>
<vsp name = "storage03" type = "direct" base = "7"/>

Usage:

...

<entry>
            <queue base="0x220"/>
            <vsp name=”storage01”>
</entry>

...

<distribution name="dist1">
     ...
     <queue count="8" base="0x230"/>
     <vsp type=”indirect” fqshift=”2” vspoffset=”0” vspcount=”4”/>
     ...
</distribution>

...

<classification name="eth_dest_clsf">
    <key>
         <fieldref name="ethernet.dst"/>
    </key>
...
    <vsp name=”storage03”>
```

```
    <action condition="on-miss" type="distribution" name="garbage"/>
</classification>
```

7.4.4.5.1.13 NXP NetPDL Reference

The FMan's Soft Parser can process non-standard, custom protocols that you define. To define a custom protocol, you enter NetPDL (Network Protocol Description Language) markup into a file called the Custom Protocol file. This markup defines each field in the custom protocol's header, as well as actions for the Soft Parser to take both before and after the custom header is loaded into the frame window.

Note: Although the markup used to define a custom protocol is based on NetPDL, this markup does not follow NetPDL rules strictly. As a result, you cannot rely on non-NXP documentation of NetPDL as you write your Custom Protocol file. Only the information in this appendix accurately explains how to write the NetPDL that goes in a Custom Protocol file.

You pass the name of the Custom Protocol file to the FMC Tool from the command line. The tool, in turn, passes the information in this file (directly or indirectly) to the FMan's Soft Parser.

7.4.4.5.1.13.1 Basic XML Rules

The Custom Protocol XML file follows standard XML rules.

The file is composed of several elements. Each element begins with a start tag and can contain attributes and/or child elements. If the element contains child elements, it must have a matching end tag. An element without child elements or text must end with a forward slash (/).

Note that element and attribute names are case sensitive. In the Custom Protocol file, all element and attribute names use only lower case alphabetics.

Comments always begin with "<!--" and end with "-->"

Example

```
<one-element attribute1="value"> <!-- this is a comment -->
  <child-element myattribute="4"/>
</one-element>
<another-element attribute2="value2"/>
```

7.4.4.5.1.13.2 The netpdl Element

The Custom Protocol file always begins with the <netpdl> root element. As a result, the end netpdl tag must appear at the end of the file.

Attributes: No required attributes

Child Elements: protocol

Example

```
<netpdl>
...
</netpdl>
```

7.4.4.5.1.13.3 The protocol element

Use the 'protocol' element to bracket the definition of each custom protocol in the Custom Protocol file. The 'protocol' element is a container for all the other elements required to define a custom protocol.

Attributes

name - (required) alphanumeric string; defines the unique name of the custom protocol.

longname - (optional) alphanumeric string; provides a user-friendly name for the protocol.

prevproto - (required) alphanumeric string. This attribute defines the previous protocol, that is, the protocol whose header precedes the custom protocol's header.

Table 134. Valid values for the prevproto attribute on page 796 lists the values that you can assign to the 'prevproto' attribute.

Table 134. Valid values for the prevproto attribute

Protocol	Layer
ethernet	2
llc_snap	2
vlan	2
pppoe	2
mpls	2
ipv4	3
ipv6	3
gre	3
minencap	3
otherl3 [11]	3
tcp	4
udp	4
ipsec_ah	4
ipsec_esp	4
sctp	4
dccp	4
otherl4 [1]	4

Each time the frame window contains a header for a protocol specified in the 'prevproto' attribute of one of the 'protocol' elements in the Custom Protocol file, the Hard Parser transfers control to the Soft Parser.

The Soft Parser then executes the 'before' element code of the 'protocol' element whose prevproto attribute matches the current protocol. As long as the 'before' element code is executing, the previous protocol's header remains in the frame window. As a result, the 'before' element code can reference the fields in the previous protocol header.

[11] The Custom Protocol file's NetPDL XML has a somewhat different structure and behavior if either 'otherl3' or 'otherl4' is the previous protocol. See Effect of Setting prevproto Attribute to otherl3 or otherl4 on page 797.

Typically, the 'before' element includes code that determines whether the next protocol header is an instance of the custom protocol defined by this protocol element. If it is not, the 'before' code instructs the Soft Parser to return to the Hard Parser; if it is, the Soft Parser continues to execute the 'before' code.

When the Soft Parser finishes executing the 'before' code (and if it does not return control to the Hard Parser), the Soft Parser advances the frame window to the custom protocol header and starts executing the 'after' element code (if any has been defined). Therefore, the code in the 'after' element can reference the fields in the custom protocol header.

Child Elements: format, execute-code

Example

```
<protocol name="gtpu" longname="GTP-U" prevproto="udp">
  ...
</protocol>

<protocol name="tcpExt" longname="tcp extension" prevproto="cp">
  ...
</protocol>
```

7.4.4.5.1.13.3.1 Effect of Setting prevproto Attribute to otherl3 or otherl4

When the 'prevproto' attribute of the 'protocol' element is set to otherl3 (for other layer 3 protocol) or otherl4 (for other layer 4 protocol), the first byte of the previous protocol header and the first byte of the custom protocol header are at the position in the frame window. Because they are not real protocols, neither otherl3 nor otherl4 has a real protocol header with a defined size and defined fields; these "protocols" are used just to provide the Soft Parser with an entry point (or a termination point) within the frame window. In effect, the size of the otherl3 and otherl4 "headers" is zero. Consequently, these "headers" have the same start offset in the frame window as does the custom protocol's header.

Note: Because the otherl3 and otherl4 protocols do not have real headers, they provide nothing for the Soft Parser to parse. As a result, you cannot use the 'before' element when either of these protocols is assigned to the 'prevproto' attribute. You can only use the 'after' element in these cases.

7.4.4.5.1.13.4 The format element

Use the 'format' element to bracket the definition of the structure of a custom protocol header. The 'format' element is a container for the 'fields' element which, in turn, is a container for the 'field' element. The 'field' element lets you define each field in a custom protocol's header.

Attributes: none

Child Elements: fields

7.4.4.5.1.13.4.1 The fields Element

Use the 'fields' element to define the structure of a custom protocol's header. This element is a container for the 'field' element, which lets you define each field in a custom protocol header.

Attributes: none

Child Elements: field

7.4.4.5.1.13.4.2 The field Element

Use the 'field' element to define one of the fields in a custom protocol header.

Attributes

type - (required) string; Defines the field size as either "fixed" for a byte-length field or "bit" for a bit-length field.

size - (required) integer; Defines the size of the field in bytes.

name - (required) string; Defines the unique name for the field.

longname - (optional) string; Defines the name of the field for display purposes.

mask - (required only for bit field) integer; Defines the specific bits in the current bytes which belong to this field.

The field elements appear one after the other to define a custom protocol's header frame. The first field begins in the first byte of the custom protocol's frame header and its size is determined by the size attribute. The following fields conform to the following rules:

- A fixed field or a field following a fixed field begins in the next byte, which is the previous field's offset + the previous field's size.
- A bit field following a bit field begins in the next byte only if the last bit in the previous field's mask is 1.
- If two fields share the same offset (which is possible only when both fields are bit fields and the mask of the first field does not end with 1), they should have the same value for their size attributes.

Example

```
<format>
  <fields>
    <field type="bit"   name="flags"   mask="0xE0" size="1"/>
    <field type="bit"   name="pt"      mask="0x80" size="1"/>
    <field type="bit"   name="version" mask="0x07" size="1"/>
    <field type="fixed" name="mtype"               size="1"/>
    <field type="fixed" name="length"              size="2"/>
  </fields>
</format>

<format>
  <fields>
    <field type="bit"   name="version" mask="0xE0" size="1"/>
    <field type="bit"   name="pt"      mask="0x10" size="1"/>
    <field type="bit"   name="flags"   mask="0x07" size="1"/>
    <field type="bit"   name="flags1"  mask="0x01" size="1"/>
    <field type="bit"   name="flags2"  mask="0x10" size="1"/>
    <field type="bit"   name="flags3"  mask="0x02" size="1"/>
    <field type="fixed" name="mtype"   size="1" longname="message type"/>
    <field type="fixed" name="length"  size="2"/>
  </fields>
</format>
```

The fields will, thus, be stored in the following bit offsets in the custom protocol header:

version: 0-2 pt: 3-3 flags: 5-7 flags1: 15-15 flags2: 19-19 flags3: 22-22 mtype: 24-31 length: 32-47

7.4.4.5.1.13.5 The execute-code element

Use the 'execute-code' element to define all code that should be executed for a custom protocol once the parser reaches the specified previous protocol header.

This element contains two child elements, 'before' and 'after'. At least one of these child elements must be defined. If both are defined, the 'before' element must appear before the 'after' element.

Attributes: none

Child Elements: before, after

Example

```
<execute-code>
  <before>
    ...
```

```
  </before>

  <after headersize="8">
  </after>
</execute-code>
```

7.4.4.5.1.13.5.1 The before Element

The Soft Parser executes the code in the 'before' element before it moves the frame window from the previous protocol header to the custom protocol header. Therefore, use the 'before' element to specify logic that requires access to fields in the previous protocol header. This code is often used to determine whether the next protocol header is an instance of the custom protocol this protocol block defines. If it is not, the 'before' block instructs the Soft Parser to return control to the Hard Parser; if it is, the Soft Parser continues processing.

While the code in the 'before' element is analyzed, the frame window points to the previous protocol header. Therefore, the frame window variable ($FW) references the fields in the previous protocol header and the header size variable ($headerSize) variable returns the size of the previous protocol's header.

Once the it reaches the end of the 'before' element, the Soft Parser moves the frame window to the custom protocol header. If no 'after' element has been defined, the Soft Parser then returns to the Hard Parser.

The 'before' element can only appear once in the 'execute-code' element and, if an 'after' element has been defined, the 'before' element must appear before the 'after' element.

Attributes

confirm - (optional) string; Valid values are "yes" and "no". The default value is "no" if an 'after' element has been defined. Otherwise, the default value is "yes". If confirm="yes", the Soft Parser confirms the presence of the 'prevproto' header by bitwise OR'ing the previous protocol's line-up enable confirmation mask with the current line-up confirmation vector (LCV) value.

confirmcustom - (optional) string; Valid values are "shim1", "shim2", and "no". The default value is "no". If 'confirmcustom' is set (!="no"), the Soft Parser confirms the presence of the custom protocol header by bitwise OR'ing the custom protocol's mask with the current line-up confirmation vector (LCV) value. The custom protocol can set one of the last two bits in the LCV. If "shim1" is selected, the least significant bit is set; if "shim2" is selected, the second least significant bit is set.

Child Elements: if, switch, assign, action

Note: When the previous protocol is 'otherl3' or 'otherl4', the previous protocol and the custom protocol are treated as if they are the same and each begins at the same offset within the frame window. Therefore, the 'before' element cannot be used when the 'prevproto' attribute is 'otherl3' or 'otherl4'; only an 'after' element be used when the the 'prevproto' attribute is 'otherl3' or 'otherl4'. See Effect of Setting prevproto Attribute to otherl3 or otherl4 on page 797 for more information.

7.4.4.5.1.13.5.2 The after Element

The 'after' element contains code which should be executed when a frame from the current custom protocol has been encountered. In contrast to the 'before' element, in the 'after' section, it is possible to access fields from the current protocol but not from the previous protocol. In the 'after' element the frame window variable ($FW) manipulates the current custom protocol header and the header size variable ($headerSize) returns the size of the current custom protocol header.

At the end of the 'after' element, the frame window jumps to the end of the custom protocol's header and control returns to the Hard Parser.

The 'after' element can appear only once in an 'execute-code' element and if a 'before' element has been defined, it must appear before the 'after' element.

Attributes

confirm - (optional) string; Valid values are "yes" and "no". The default value is "yes". If confirm ="yes", the Soft Parser confirms the existence of the previous protocol header by bitwise OR'ing the previous protocol's line-up enable confirmation mask with the current line-up confirmation vector (LCV) value.

confirmcustom - (optional) string; Valid values are "shim1", "shim2", and "no". The default value is "no". If 'confirmcustom' is set (!="no"), the Soft Parser confirms the presence of the custom protocol header by bitwise OR'ing the custom protocol's mask with the current line-up confirmation vector (LCV) value. The custom protocol can set one of the two last bits in the LCV. If "shim1" is selected, the least significant bit is set; if "shim2" is selected, the second least significant bit is set.

headerSize - (optional) integer; Possible values: arithmetic expression. (See Arithmetic Expressions on page 815) The default value is calculated using the fields contained by the 'format' element. You can specify the custom protocol's header size with this attribute. This information is needed so the parser returns to the right position following the custom protocol header. If header size is not specified, the FMC Tool assumes that the fields defined inside the 'format' element are the only fields in the custom protocol header and calculates the header size using these fields. The $headerSize variable in the 'after' element returns the value defined in this attribute (or the value calculated by default if the header attribute is not defined).

Child Elements: if, switch, assign, action

Example

```
<protocol name="gtp" prevproto="udp">
  <format>
    <fields>
      <field type="bit" name="version" mask="0xE0" size="1"/>
    </fields>
  </format>

  <execute-code>
    <before confirm="no">
      <assign-variable name="$GPR1" value="udp.dport"/>
      <!-- Note that this is ILLEGAL: <assign-variable name="GPR1" value="version" -->
      <assign-variable name="$shimr" value="$headerSize"/>
      <!-- shimresult now holds udp's header size -->
    </before>

    <after headersize="4" confirmcustom="shim1">
      <!-- Note that this is ILLEGAL: <assign-variable name="$GPR1" value="udp.dport"> -->
      <assign-variable name="$GPR1" value="version"/>
      <assign-variable name="$shimr" value="$headerSize"/>
      <!-- shimresult now equals 4 -->
    </after>
  </execute-code>
</protocol>
```

7.4.4.5.1.13.5.3 Child Elements of the before and after Elements

7.4.4.5.1.13.5.3.1 The assign-variable Element

The 'assign-variable' element assigns an expression to a variable.

Attributes

name - (required) string; The name of the variable to which a value will be assigned. Valid values: Variables contained in the result array.

value - (required) integer; The expression assigned to the variable. Valid values: arithmetic expressions.

Child Elements: none

Example

```
<assign-variable name="$shimoffset_2" value="$shimoffset_1+12"/>
```

7.4.4.5.1.13.5.3.2 The if Element

This element tests the specified condition. If the condition is true, control transfers to the 'if-true' element; if the condition is false, control transfers to the 'if-false' element (if one is defined).

Attributes

expr - (required) string; Defines the condition to be checked before selecting the code block to execute. Valid values: logical expressions. (See Logical Expressions on page 815 for more information.)

Child Elements: if-true (required), if-false

Example

```
<if expr="$shimoffset_1==1">
  <if-true>
    ...
  </if-true>
  <if-false>
    ...
  </if-false>
</if>
```

7.4.4.5.1.13.5.3.2.1 The if-true Element

This element defines code to execute if the expression defined in the parent 'if' element is true.

Attributes: none

Child Elements: if, switch, assign, action (the same child elements as for the 'before' and 'after' elements)

Example

```
<if expr="$shimoffset_1==1">
  <if-true>
    ...
  </if-true>
  <if-false>
    ...
  </if-false>
</if>
```

7.4.4.5.1.13.5.3.2.2 The if-false Element

This element defines the code to execute if the expression defined in the parent 'if' element is false.

Attributes: none

Child Elements: if, switch, assign, action (the same child elements as for the 'before' and 'after' elements)

Example

```
<if expr="$shimoffset_1==1">
  <if-true>
    ...
  </if-true>
  <if-false>
    ...
  </if-false>
</if>
```

7.4.4.5.1.13.5.3.3 The switch Element

This element defines an expression and a set of cases. Each case consists of a value (or set of values) and code to be executed if the value equals the switch expression. Each 'switch' element must have at least one 'case' child element.

Note: Only the code of the first case that matches the swith expression is executed. Any following cases are skipped. In C language terms, a break is automatically added after the code of each case.

Attributes

expr - (required) string; Defines the value being checked. Valid values: arithmetic expressions.

Child Elements: case, default

Example

```
<switch expr="$shimoffset_1+1">
  <case value="2">
    <assign-variable name="$GPR[1:1]" value="0"/>
  </case>

  <case value="3" maxvalue="4">
    <assign-variable name="$GPR[1:1]" value="1"/>
  </case>

  <default>
    <assign-variable name="$GPR[1:1]" value="2"/>
  </default>
</switch>
```

7.4.4.5.1.13.5.3.3.1 The case Element

This element matches a value or range of values against the switch expression.

Attributes

value - (required) integer; If the value equals the switch expression and no earlier case has been matched, the code in the 'case' element is executed.

maxvalue - (optional) integer; If the switch expression is greater than or equal to the 'value' attribute and the expression is less than or equal to the 'maxvalue' attribute (and no earlier case has been matched), the code in the 'case' element is executed.

Child Elements: if, switch, assign, action (the same child elements as for the 'before' and 'after' elements)

Example

```
<switch expr="$shimoffset_1+1">
  <case value="2">
    <assign-variable name="$GPR[1:1]" value="0"/>
  </case>

  <case value="3" maxvalue="4">
    <assign-variable name="$GPR[1:1]" value="1"/>
  </case>

  <default>
    <assign-variable name="$GPR[1:1]" value="2"/>
  </default>
</switch>
```

7.4.4.5.1.13.5.3.3.2 The default Element

The 'default' element contains code that is executed if the expression in the 'switch' element is not matched by any of the candidate cases.

Attributes: none

Child Elements: if, switch, assign, action (the same child elements as for the 'before' and 'after' elements)

Example

```
<switch expr="$shimoffset_1+1">
  <case value="2">
    <assign-variable name="$GPR[1:1]" value="0"/>
  </case>

  <case value="3" maxvalue="4">
    <assign-variable name="$GPR[1:1]" value="1"/>
  </case>

  <default>
    <assign-variable name="$GPR[1:1]" value="2"/>
  </default>
</switch>
```

7.4.4.5.1.13.5.3.4 The action Element (for use in a Custom Protocol file)

Use the 'action' element in a 'before' or 'after' block to terminate soft parsing, jump to the specified next protocol header, and continue hard parsing.

Note: This topic defines the 'action' element used in a Custom Protocol file. See The action element (for use in a policy file) on page 826 for the definition of the 'action' element used in a Policy file.

Attributes

- type - (required) string; "exit" is the only valid value for the type attribute.
- advance - (optional) string; The 'advance' attribute controls whether the Soft Parser moves the frame window to the next frame header. This attribute has different meanings in the 'before' and 'after' elements. In the 'before' element, the Soft Parser moves the frame window from the previous protocol header to the custom protocol header. In the 'after' element, the Soft Parser moves the frame window from the custom protocol header to the specified next protocol header. The frame window is advanced according to the header size. The value of 'advance' must be 'yes' or 'no'. The default is 'yes' unless 'nextproto' is set to 'end_parse', 'return', or not set at all. In these cases, the default value is 'no'.
- confirm - (optional) string; If confirm="yes", the Soft Parser bitwise OR's the previous protocol's line-up enable confirmation mask with the current line-up confirmation vector (LCV) value. Valid values are "yes" and "no"; the default value is "yes".
- confirmcustom - (optional) string; Valid values are "shim1", "shim2", or "no". The default value is "no". If confirmcustom is set to a value other than "no", the Soft Parser bitwise ORs the custom protocol's mask with the current line-up confirmation vector (LCV) value. The custom protocol can set one of the two last bits in the LCV. If shim1 is specified, the least significant bit is set; if shim2 is specified, the second least significant bit is set.
- nextproto - (optional); If used, this attribute must be one of the values in Table 135. Parse Action for each Value of the nextproto Attribute on page 804. The default value is 'return'.

Table 135. Parse Action for each Value of the nextproto Attribute

If nextproto is ...	The parse action is ...
ethernet	Jump to the Ethernet header and continue hard parsing
llc_snap	Jump to the LLC_SNAP header and continue hard parsing
vlan	Jump to the VLAN header and continue hard parsing
pppoe	Jump to the PPPoE header and continue hard parsing
mpls	Jump to the MPLS header and continue hard parsing
ipv4	Jump to the IPv4 header and continue hard parsing
ipv6	Jump to the IPV6 header and continue hard parsing
gre	Jump to the GRE header and continue hard parsing
minencap	Jump to the MinEncap header and continue hard parsing
otherl3	Jump to the otherl3 header and continue hard parsing
tcp	Jump to the TCP header and continue hard parsing
udp	Jump to the UDP header and continue hard parsing
ipsec_ah	Jump to the IPsec_ah header and continue hard parsing
ipsec_esp	Jump to the IPsec_esp header and continue hard parsing
sctp	Jump to the SCTP header and continue hard parsing
dccp	Jump to the DCCP header and continue hard parsing
otherl4	Jump to the otherl4 header and continue hard parsing
after_ethernet	Jump to the protocol that should follow the Ethernet header. The next protocol is determined from the value of the $nxtHdr variable. See Table 136. Next Protocol for each $nxtHdr Value if nextproto is 'after_ethernet' on page 805to find the next protocol for each possible value of $nxtHdr. **Note:**The 'advance' attribute must be set to 'yes' if 'nextproto' is set to 'after_ethernet'.
after_ip	Jump to the protocol that should follow the IP header. The next protocol is determined from the value of the $nxtHdr variable. See Table 136. Next Protocol for each $nxtHdr Value if nextproto is 'after_ethernet' on page 805to find the next protocol for each possible value of $nxtHdr. **Note:**The 'advance' attribute must be set to 'yes' if 'nextproto' is set to 'after_ip'.

Table continues on the next page...

Table 135. Parse Action for each Value of the nextproto Attribute (continued)

If nextproto is ...	The parse action is ...
return (default value)	Return to the Hard Parser without advancing the frame window. In this case, the Hard Parser starts parsing the frame header at the same position at which the Soft Parser began. The 'advance' attribute cannot be 'yes' when 'nextproto is set to return.
none/end_parse	Finish parsing the frame header; do not return to the Hard Parser.

Table 136. Next Protocol for each $nxtHdr Value if nextproto is 'after_ethernet'

If $nxtHdr is ...	The next protocol is ...
0x05DC or less	llc_snap
0x0800	ipv4
0x86DD	ipv6
0x8847, 0x8848	mpls
0x8100, 0x88A8, ConfigTPID1, ConfigTPID2	vlan
0x8864	pppoe
other value	otherl3

Table 137. Next Protocol for each $nxtHdr Value if nextproto is 'after_ip'

If $nxtHdr is ...	The next protocol is ...
4	ipv4
6	tcp
17	udp
33	dccp
41	ipv6
50, 51	ipsec
47	gre
55	minencap
132	sctp
other value	otherl4

Notes

- The frame window *must* be advanced when parsing jumps to the 'after_ethernet' or 'after_ip' protocols. Therefore, the 'advance' attribute cannot be set to 'no' in these cases.
- The frame window must *not* be advanced before a 'return' to the Hard Parser. Therefore, the 'advance' attribute cannot be set to 'yes' if nextproto is set to 'return' or not set at all (since 'return' is the default 'nextproto' value).

Child Elements: none

Example

```
<action type="exit"
        advance="yes"
        confirmcustom="shim2"
        confirm="no"
        nextproto="udp"/>
```

7.4.4.5.1.13.6 Expressions

Expressions are constructed of operands and operators. The simplest expression can contain just one operand. Most operators are dyadic and separate two operands (such as +, -) and some operators are monadic and operate on just the operand that follows them (such as 'not').

7.4.4.5.1.13.6.1 Operands

These are the supported types of operands: numbers, variables, fields, and expressions.

Note: The maximum size of an operand is 64 bits (8 bytes).

7.4.4.5.1.13.6.1.1 Numbers

Numbers can appear in decimal (no prefix), binary (prefixed by '0b'), or hexadecimal (prefixed by '0x') format.

All numbers are 64-bit unsigned integers. However, some operators only use the 32 LSB of a number.

Note: Immediate, primitive negative numbers are not supported. For example, the number -2 cannot appear in an expression. However, artificial negative values can be created using arithmetic expressions such as 1-3 (which returns 0xfffffffe).

7.4.4.5.1.13.6.1.2 Fields

Fields are defined with the 'format' element in a custom protocol header definition. There are two ways to access a field, by typing their name directly or by typing the name of the protocol header containing the field, followed by a period, followed by the name of the field.

In the 'before' element, it is only possible to access fields in the previous protocol header; in the 'after' element, it is only possible to access fields in the current custom protocol header.

Note: Fields longer than 8 bytes cannot be accessed individually. You can work around this limit by accessing the frame directly using the frame window ($FW) variable or by splitting the field into several shorter fields.

Example

```
<protocol name="gptu" prevproto="#ethernet">
  <format>
    <fields>
      <field type="fixed" name="example" size="2"/>
    </fields>
  </format>

  <execute-code>
    <before>
      <assign-variable name="$l2r" value="ethernet.type"/>
    </before>
```

```
    <after>
      <assign-variable name="$shimoffset_2" value="example"/>
    </after>
  </execute-code>
</protocol>
```

7.4.4.5.1.13.6.1.3 Variables

All variable names begin with the $ prefix and are case-sensitive. These variables are supported: frame window, header size, prevprotoOffset, parameter array, and result array variables.

7.4.4.5.1.13.6.1.3.1 Result Array Variables

Result array variables return values contained in the parse results array.

Syntax for accessing result array variables:

- $variableName - returns the entire variable
- $variableName[byteOffset:byteNumber] - Returns the byteNumber number of bytes in the variable starting from byteOffset. This access method is useful for accessing a subset of the bytes in the variable. In bytesNumber equals zero, the entire variable is returned, starting from byteOffset.

Example: The variable $actiondescriptor returns result array bytes 64-71. The expression $actiondescriptor[2:4] returns result array bytes 66-69 since 66 is at offset 2 of the actiondescriptor variable and the requested size is 4. The expression $actiondescriptor[3:0] returns result array bytes 67-71 since 67 is at offset 3 of the actiondescriptor variable and the requested size is 0, which means return the entire variable starting at the specified offset (3).

Other usage: In addition to expressions, result array variables can be used in the left side of 'assign-variable' elements to modify result array values.

Table 138. Result Array Variables on page 807shows the available result array variables .

Table 138. Result Array Variables

Variable Name	Result Array Bytes Referenced
gpr1	0-7
gpr2	8-15
logicalportid	16-16
shimr	17-17
l2r	18-19
l3r	20-21
l4r	22-22
classificationplanid	23-23
nxthdr	24-25
runningsum	26-27
flags	28-28

Table continues on the next page...

Table 138. Result Array Variables (continued)

Variable Name	Result Array Bytes Referenced
fragoffset	28-29
routtype	30-30
rhp	31-31
ipvalid	31-31
shimoffset_1	32-32
shimoffset_2	33-33
ip_pidoffset	34-34
ethoffset	35-35
llcs_napoffset	36-36
vlantcioffset_1	37-37
vlantcioffset_n	38-38
lastetypeoffset	39-39
pppoeoffset	40-40
mplsoffset_1	41-41
mplsoffset_n	42-42
ipoffset_1	43-43
ipoffset_n	44-44
minencapo	44-44
minencapoffset	44-44
greoffset	45-45
l4offset	46-46
nxthdroffset	47-47
framedescriptor1	48-55
framedescriptor2	56-63
actiondescriptor	64-71

Table continues on the next page...

Table 138. Result Array Variables (continued)

Variable Name	Result Array Bytes Referenced
ccbase	72-75
ks	76-76
hpnia	77-79
sperc	80-80
ipver	85-85
iplength	86-87
icp	90-91
attr	92-92
nia	93-95
ipv4sa	96-99
ipv4da	100-103
ipv6sa1	96-103
ipv6sa2	104-111
ipv6da1	112-119
ipv6da2	120-127

Note: The $GPR2 variable is used internally by the FMC Tool to calculate complex expressions, including checksum calculations. Using $GPR2 for other purposes is possible, but is not supported or recommended.

7.4.4.5.1.13.6.1.3.2 Parameter Array Variable

This variable returns data from the parameter array. Because the parameter array is more than 8 bytes long, you must specify the particular bytes needed.

Accessing parameter array variables: $PA[byteOffset:byteNumber] - returns the byteNumber number of bytes in the parameter array starting at byteOffset.

Example: The expression "$PA[4:2]" accesses the fifth and sixth bytes (indexed at PA[4] and PA[5]) of the parameter array.

7.4.4.5.1.13.6.1.3.3 Header Size Variables

Header size variables return the header size or default header size of a protocol header.

Accessing header size variables: $headerSize or $defaultHeaderSize

- In the 'before' element, the $headerSize of the previous protocol header is returned. Accessing $defaultHeaderSize is not allowed.

- In the 'after' element, the $defaultHeaderSize variable returns the number of bytes in the custom protocol's format fields. The $headerSize variable returns the headerSize as defined by the 'headersize' attribute of the 'after' element. If the user has not specified a value for the 'headersize' attribute, $headerSize returns the same value as $defaultHeaderSize.

7.4.4.5.1.13.6.1.3.4 Frame Window Variable

The frame window variable ($FW) returns data from the frame array. In the 'before' element, the frame window variable returns data from the previous protocol's header. In the 'after' element, the frame window variable returns data from the custom protocol header.

Using the frame window variable: $variableName[bitOffset:bitNumber] - Returns the bitNumber number of bits in the frame header starting from bitOffset.

Note: The frame window uses similar syntax to the parameter array and result array variables; however, the frame window variable accesses bits instead of bytes.

Examples

To access the tenth and eleventh bits in the frame array (indexed at FW[9], FW[10]), use "$FW[9:2]".

To access the entire third byte of the frame array, use "$FW[16:8]".

The conditions in the example below are always true because the same bits can be accessed using either the $FW variable or header field names.

```
<format>
  <fields>
    <field type="bit" name="first" size="1" mask="0xE0"/>
    <field type="bit" name="second" size="1" mask="0x1"/>
    <field type="bit" name="third" size="1" mask="0xF"/>
    <field type="fixed" name="fourth" size="2"/>
  </fields>
</format>
...
<after>
  <if expr="first==$FW[0:3]"> ... </if>
  <if expr="second==$FW[7:1]"> ... </if>
  <if expr="third==$FW[8:4]"> ... </if>
  <if expr="fourth==$FW[16:16]"> ... </if>
</after>
```

7.4.4.5.1.13.6.1.3.5 The prevprotoOffset Variable

This variable returns the offset of the previous protocol's frame header. This variable has the same value in the 'before' and 'after' sections and always refers to the protocol defined in the 'prevproto' attribute of the protocol element.

In the 'before' element, the frame window's current location is equal to prevprotoOffset. In the 'after' element. the frame window's current location is equal to prevprotoOffset+headerSize.

Note: This variable is actually a "shortcut" to the result array and returns or modifies values taken directly from this array.

Table 139. Previous Protocol RA Return Values

If the previous protocol is ...	The value returned from result array is ...
ethernet	$ethoffset
gre	$greoffset

Table continues on the next page...

Table 139. Previous Protocol RA Return Values (continued)

If the previous protocol is ...	The value returned from result array is ...
ipv4, ipv6	$lpoffset_n
llc_snap	$llcsnapoffset
minencap	$minencapoffset
mpls	$mplsoffset_n
pppoe	$pppoeoffset
tcp, udp, sctp, dccp, ipsec_ah, ipsec_esp	$l4offset
vlan	$vlanoffset_n
otherl3, otherl4	$NxtHdrOffset - When the previous protocol is otherl3 or other l4, the custom protocol and the previous protocol have the same offset. See Effect of Setting prevproto Attribute to otherl3 or otherl4 on page 797.

7.4.4.5.1.13.6.2 Operators

The parser supports many operators. These operators can receive arithmetic or logical operands and return an arithmetic or logical value. An arithmetic value is a number, while a logical value is true or false. (See Arithmetic Expressions on page 815 and Logical Expressions on page 815 for more information.)

Table 140. Supported Operators and their Properties on page 811describes all operators and their associated properties. All dyadic operators (operators which receive two parameters) appear between two operands. All monadic operators appear before an operand.

Table 140. Supported Operators and their Properties

Name	Parameters	Description	Symbol
Greater than	Logical (Arithmetic, Arithmetic)	Checks if the value of the first expression is greater than the second	gt
Greater equal	Logical (Arithmetic, Arithmetic)	Checks if the value of the first expression is equal to or greater than the second	ge
Less than	Logical (Arithmetic, Arithmetic)	Checks if the value of the first expression is less than the second	lt
Less equal	Logical (Arithmetic, Arithmetic)	Checks if the value of the first expression is equal to or less than the second	le
Equal	Logical (Arithmetic, Arithmetic)	Checks if the two expressions are equal	==
Not equal	Logical (Arithmetic, Arithmetic)	Checks if the two expressions are not equal	!=

Table continues on the next page...

Table 140. Supported Operators and their Properties (continued)

Name	Parameters	Description	Symbol
Logical AND	Logical (Logical, Logical)	Checks if both expressions are true	and
Logical OR	Logical (Logical, Logical)	Checks if either one of the expressions is true	or
Logical NOT	Logical (Logical)	Returns true if the expression is false; returns false otherwise	not
Add	32-bit Arithmetic (32-bit Arithmetic, 32-bit arithmetic)	Return the sum of the expressions	+
Subtract	32-bit arithmetic (32-bit Arithmetic, 32-bit arithmetic)	Return the difference between the two expressions (result of subtraction)	-
Add carry	16-bit arithmetic (16-bit arithmetic, 16-bit arithmetic)	Return the sum of the two expressions summed with the carry after 32bit	addc
Bitwise OR	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise OR operation on the two expressions	bitwor
Bitwise XOR	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise XOR operation on the two expressions	bitwxor
Bitwise AND	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise AND operation on the two expressions	bitwand
Bitwise NOT	Arithmetic (Arithmetic)	Returns the result of a bitwise NOT operation on the expression	bitwnot
Shift left	Arithmetic (Arithmetic, Integer - value up to 64 bits)	Return the left expression shifted left by the right expression	shl
Shift right	Arithmetic (Arithmetic, Integer - value up to 64 bits)	Return the left expression shifted right by the right expression	shr
Concat	Arithmetic (Arithmetic, Variable or Integer)	Special operator See The concat Operator on page 812 for full documentation	concat
Checksum	Arithmetic (Arithmetic - value up to 0xffff, Arithmetic - value up to 256, Arithmetic - value up to 256)	Special operator See The checksum Operator on page 813 for full documentation	checksum

7.4.4.5.1.13.6.2.1 The concat Operator

The concat operator shifts its first argument left and inserts its second argument to its right. The concat operation can be executed on variables or integers. If the second argument is a variable, the first argument is shifted left according to the known size of the variable. Result array variables have constant sizes and the size of the frame header's fields are set in the Custom Protocol file or the Standard Protocol file.

If the user accesses only specific bits in the second argument, the first argument is shifted left only by the number of bits specified.

If the second argument is an integer, the first argument is shifted left by the smallest word size into which the integer fits: 16, 32, 48, or 64.

Note: The second argument of a concat operation cannot be an expression because the FMC Tool does not know the size of an expression and therefore cannot shift the first argument properly. However, for expressions, you can replace the concat operation with a shift operation (as long as you know the number of bits to shift) and a bitwise OR operation.

Note: You should use concat instead of shift/bitwise OR when working with variables and integers in order to reduce code size.

For example, the following IF expression is true:

```
<assign-variable name="$shimr" value="2"/>
<assign-variable name="$GPR1[6:2]" value="3"/>
<if expr="1 concat $shimr concat $GPR1[6:2] concat 0x40000 == 0x102000300040000">
```

7.4.4.5.1.13.6.2.2 The checksum Operator

The checksum operator is a special operator with unique behavior and syntax. It appears before three operands that have parentheses around them. As a result, the concat operator looks like a function call - checksum(expression, integer, integer).

The first operand defines the initial checksum value. The second operand defines the frame window offset at which to start the checksum (relative to the current frame window location). The third operand defines the length of the data in bytes on which the checksum operation should be calculated.

Using these values, the checksum executes the add carry (addc) operation on 2-byte sized words in the frame window range specified. If the range specified contains an odd number of bytes to be checksummed, the last byte is padded on the right with zeros to form a 16-bit word for checksum purposes. The total sum is added to the initial checksum value using another addc operation. Therefore, the first argument that defined the initial sum value must be smaller than 0xffff. The result of the final addc operation is returned.

Note: Since it is only possible to access 256 bytes in the frame window, the last two arguments to the checksum operator must be less than or equal to 256.

Example

Suppose we have the following frame and the custom protocol header begins at offset 0xE (where 4500 appears):

```
FFFF FFFF FFFF 0CCB CC0D DDDD 0800 4500 002E 0000 4000 402F
2AA2 1000 0000 FFFE 0001 0308 0900 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 DA95 36D6 6F15 778C
```

The following IF conditions will always be true:

```
<after>
  <if expr="checksum(0x30A2,2,7+2)==0xDAFF">
    ...
  </if>

  <if expr="checksum(0,0,20)==0xFFFF">
    ...
  </if>
</after>
```

The first checksum operation above performs the following calculation:

```
0x30A2 + (0x002E add 0x0000 addc 0x4000 addc 0x402F addc 0x2A00)
```

The second checksum operation performs the following calculation:

```
0x0000 + (0x4500 addc 0x002E addc 0x0000 addc 0x4000 addc 0x402F addc 0x2AA2
            addc 0x1000 addc 0x0000 addc 0xFFFE addc 0x0001)
```

7.4.4.5.1.13.6.2.3 Expression Priorities

Expressions containing multiple operators perform the operation according to the following rules, in the order shown:

1. Operations in parentheses are performed
2. Operations that have a higher priority are performed
3. Multiple operations with the same priority are then executed from left to right

Note: Parentheses are recommended when several operators appear in the same expression to ensure correct calculation.

7.4.4.5.1.13.6.2.4 Operator Precendence

If several operators appear in the same expression (without separating parentheses), they are performed in the following order:

1. NOT, bitwise NOT, checksum
2. add, subtract, add carry
3. bitwise AND, bitwise OR, bitwise XOR
4. shift right, shift left, concat
5. greater than, greater equal, less than, less equal, equal, not equal
6. AND, OR

7.4.4.5.1.13.6.2.5 Variable Size

In most operations, expression size is limited to 64 bits. However, there are a few exceptions:

- When shifting variables, the shift value must be less than or equal to 64 bits since there are only 64 bits in an expression.
- The add carry operation can only be performed on 16-bit variables and always returns a 16-bit variable. The Soft Parser reports an error if an add carry operation is performed on a constant larger than 16 bits, but does not recognize a complex expression larger than 16 bits. Therefore, it is the responsibility of the user to perform the operation on 16-bit variables only.
- The subtract and add operators can only be performed on 32-bit variables and they always return a 32-bit result. If two 32-bit expressions are added and their result is larger than 32 bits, only the carry is returned, such that the returned value is a 32-bit variable. The Soft Parser reports a warning if an add carry operation is performed on a constant larger than 32 bits, but does not recognize a complex expression larger than 32 bits. Therefore, it is the responsibility of the user to perform the operation on 32-bit variables only.

For example, the following IF expressions are always true:

- ```
  <if expr="0xFFFFFFFF+2==0x1">
  ```
- ```
  <if expr="0x123456781+3==0x123456784">
  ```

The following IF expression is false (and should not be used):

- ```
  <if expr="3+0x123456781==0x123456784">
  ```

7.4.4.5.1.13.6.3 Expression Types

There are two main types of expressions: Logical expressions, which return "true" or "false", and arithmetic expressions, which return a numeric result.

7.4.4.5.1.13.6.3.1 Logical Expressions

Logical expressions appear in the 'expr' attribute of the 'if' element.

These expressions always return "true" or "false" and, therefore, must use at least one logical operator that separates arithmetic and logical operators.

Examples

The following expressions are logical expressions:

- ```
  (4+1==$shimoffset_1 or 5!=$shimoffset_2)
  ```
- ```
  not($shimoffset_2 ge $shimoffset_1 or $shimoffset_1 lt $shimoffset_2)
  ```

The following expressions are NOT logical expressions:

- ```
  (7 gt 3 and 2+7)
  ```
- ```
  (5 lt 8 or 7)
  ```

7.4.4.5.1.13.6.3.2 Arithmetic Expressions

Arithmetic expressions always have a numeric result. They can hold a single operand (a number, variable, or arithmetic expression) or more than one operand separated by arithmetic operators. Logical operators are not allowed in arithmetic expressions.

Arithmetic expressions can appear in the following places:

- The value attribute of the assign element
- The headersize attribute of the after element
- The expr attribute of the switch element

Examples

The following are arithmetic expressions:

- ```
  ($FW[0:16]+4)
  ```
- ```
  ($shimoffset_1 concat 3)
  ```
- ```
  (3+7+8+$shimoffset_2)
  ```
- ```
  4
  ```

The following is NOT an arithmetic expression:

- ```
  4==$shimoffset_2
  ```

7.4.4.5.1.13.7 Tips and Recommendations

7.4.4.5.1.13.7.1 Result Array Fields that Must be Manually Updated

The FMC Tool lets you define custom protocol headers, and the Soft Parser parses these headers. However, the Soft Parser does not update header fields for you (other than advancing the frame window and updating the line-up confirm vector (LCV) with the previous protocol). (See The before Element on page 799, The after Element on page 799, and The action Element (for use in a Custom Protocol file) on page 803 topics for more information.)

Therefore, some result array fields are left empty unless you manually update them. These fields might be needed in later stages in order for the Soft Parser to correctly interpret the custom protocol header. A list of result array fields that should be updated appears in the Frame Manager Parser section of the *QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual*. These fields include $Classificationplanid, $nxtHdr, $Runningsum, HXS offsets, Last E Type Offset, and $nxtHdrOffset. Note that the HXS offsets, $nxtHdr, and $nxtHdrOffset fields are also used internally by the Soft Parser; therefore, these fields should be modified carefully.

The $nxtHdr fields should be modified only if the custom protocol does not jump to 'after_ip' or 'after_ethernet', or if you want to change the next protocol when jumping to 'after_ip' or 'after_ethernet'. You should only modify the HXS offsets and next header offsets in the 'after' element or in the 'before' element if the parser exits without advancing the frame window.

Finally, the LCV should be manually updated when a custom protocol is being parsed. This can be done using the 'confirmcustom' attribute, which is available in the 'before', 'after', and 'action' elements.

7.4.4.5.1.13.7.2 Result Array Fields that Should Not be Modified

Some fields in the result array are for the Soft Parser's exclusive use and therefore should not be modified by the user. These fields are:

- $GPR1 is used to store temporary values in complex operations; therefore, you should not modify it.
- $nxtHdr is used to calculate the position of the next protocol header when the 'protocol' element's 'nextproto' attribute is set to 'next_ethernet' or 'next_ip'. Therefore, this variable should not be modified when 'nextproto' equals one of these values.
- $prevprotoOffset is used to advance the frame window between the 'before' and 'after' elements or when using the 'action' element with the 'advance' attribute in the 'before' element. Therefore, this variable should not be modified in the 'before' element unless the Soft Parser exits this element without advancing the frame window. In addition, $prevprotoOffset can equal these result array variables: $ethoffset, $greoffset, $ipoffset_n, $llcsnapoffset, minencapoffset, mplsoffset_n, pppoeoffset, l4offset, vlanoffset_n, and $nxtHdrOffset. As a result, these variable should also not be modified by code in the 'before' element.
- $nxtHdrOffset is used to advance the frame window between the 'before' and 'after' elements or when using the 'action' element with the 'advance' attribute in the 'before' element. Therefore, this variable should not be modified in the 'before' element unless the Soft Parser exits this element without advancing the frame window.

7.4.4.5.1.13.7.3 Setting the Next Protocol

The Soft Parser can be used to add code for an existing protocol or to define an entirely new protocol. When it is used as an extension for an existing protocol and no new frame headers are being parsed, the 'nextproto' attribute of the 'action' element should be set to 'return'. In this case, the nextproto attribute can also be left empty since 'return' is the default value. If 'return' is set, the Soft Parser will execute its code and then the Hard Parser will continue parsing at the same position in the frame header at which it stopped.

When the Soft Parser is used for a custom protocol with its own header, the Hard Parser must skip this header (since it does not know how to parse it) and, therefore, the next protocol must be set to a specific protocol. If the next protocol is unknown, the 'nextproto' attribute in the 'action' element can be set to 'after_ip' or 'after_ethernet'. In these cases, the next protocol header is determined using the value of the $nxtHdr field.

Example

1. If we want to execute the Soft Parser because when we parse the Ethernet protocol, our code will likely include an action similar to the action below, which will appear in the 'before' element.

```
<action type="exit" advance="no" next="return">
```

2. If we want to add a custom protocol after Ethernet and then jump to IPv6, our code will likely include an action similar to the action below, which will appear in the 'after' element...

```
<action type="exit" advance="yes" next="ipv6">
```

3. If we want to add a custom protocol after the Ethernet header, and we do not know where to jump next, our code will likely include an action similar to the action shown below, which will appear in the 'after' element. In this case when "after_ethernet" is used as next protocol, $nxtHdr variable but be dynamically assigned accordingly from custom protocol header by using next protocol and field names as value.

```
<assign-variable name="$nxtHdr" value="protocol.field"/>
<action type="exit" advance="yes" next="after_ethernet">
```

7.4.4.5.1.13.8 Limitations

This section discusses limitations you should consider when working with the FMC Tool's Soft Parser functionality.

7.4.4.5.1.13.8.1 Complex Expressions

Some expressions contain so many operations and parentheses that they are too complicated for the Soft Parser. If you receive an error stating that an expression is too complex, it may be necessary to simplify the expression by splitting it into multiple, smaller expressions, using parentheses, or storing temporary values in the result array variables.

Note: $GPR1 is recommended for storing temporary variables. Do not use $GPR2 for temporary variables because it is used internally by the tool).

Note that the checksum operation expressions can easily become too complex and must be simplified.

7.4.4.5.1.14 NetPCD Reference

7.4.4.5.1.14.1 The netpcd element

The 'netpcd' element is the root element of a NetPCD document (also known as a policy file). As a result, the 'netpcd' element must appear before any other NetPCD element.

7.4.4.5.1.14.1.1 netpcd Attribute Definitions

Table 141. netpcd Attribute Definitions

Attribute	Requirement	Description
name	optional	Free text. Use to describe the name and the purpose of the Policy file.
version="1.0"	optional	Version of the NetPCD DTD or XML schema. Currently there is only one version - "1.0," which is the default.
creator	optional	Author's name
date	optional	Date the document was created

7.4.4.5.1.14.1.2 netpcd Example

```
<?xml version="1.0"?>
<netpcd version="1.0" name="Example" creator="Serge Lamikhov">
```

```
  <!-- Other NetPCD elements like 'policy', 'distribution', etc -->
  <policy name="ipv4">
    <dist_order>
      <distributionref name="eth_dist"/>
      <distributionref name="default_dist"/>
    </dist_order>
  </policy>
</netpcd>
```

7.4.4.5.1.14.2 The policy element

The 'policy' element defines a prioritized list of distributions.

A policy element is assigned (via its name attribute) to a port or ports using markup in the Configuration file. Thus, the 'policy' element is the means by which specific PCD rules defined in the Policy file are applied to traffic arriving on particular FMan ports.

Upon receipt of a frame on given port, the Hard Parser tries to match this frame to the distribution listed first in the policy assigned to this port. If the frame matches, this distribution handles the frame. If the frame does not match, the Hard Parser next tries to match the frame to the second distribution in the policy list. This process continues until a distribution in the list matches or no more distributions are left in the policy element's list, in which case, the frame is placed on the FMan's default receive queue.

7.4.4.5.1.14.2.1 policy Attribute Definitions

Table 142. policy Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the policy. A port definition in the Configuration file references this name, thereby applying this policy to all frames arriving on this port.

7.4.4.5.1.14.2.2 policy Example

```
Policy File

<policy name="ipv4"> <!-- policy name is ipv4 -->
  <dist_order>
    <distributionref name="eth_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>


Configuration File

<cfgdata>
  <config>
    <engine="fm0">
      <port type="MAC" number="1" policy="ipv4"/> <!-- policy name ipv4 goes here -->
    </engine>
  </config>
</cfgdata>
```

7.4.4.5.1.14.3 The dist_order element

The 'dist_order' element is a container for a list of distribution references.

The Hard Parser chooses a particular distribution in this list at the moment when the protocol set made from the protocols participating in a distribution is a subset of the protocols found in the current network packet.

The distribution reference list contained within 'dist_order' element is processing sequentially, and the first conforming distribution is the distribution that is used. Thus, the order of distribution references is important.

7.4.4.5.1.14.3.1 dist_order Attribute Definitions

Table 143. dist_order Attribute Definitions

Attribute	Requirement	Description
none	n/a	n/a

7.4.4.5.1.14.3.2 dist_order Example

```
<policy name="ipv4">
  <dist_order>
    <distributionref name="tcp_dist"/>
    <distributionref name="udp_dist"/>
    <distributionref name="ethernet_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>
```

Note: In this example, putting "ethernet_dist" (which is supposed to process network traffic other than TCP and UDP) above "tcp_dist" will lead to all traffic be distributed according to "ethernet_dist" rule and no packets will reach "tcp_dist" or "udp_dist" rules. This is because the Ethernet protocol is a part of TCP and UDP frames as well.

7.4.4.5.1.14.4 The distributionref element

The 'distributionref' element references a 'distribution' element by its name.

The 'dist_order' element contains one or more 'distributionref' elements, thereby defining a prioritized list of distributions.

7.4.4.5.1.14.4.1 distributionref Attribute Definitions

Table 144. distributionref Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the referenced 'distribution' element

7.4.4.5.1.14.4.2 distributionref Example

```
<policy name="ipv4">
  <dist_order>
    <distributionref name="eth_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>
```

7.4.4.5.1.14.5 The distribution element

The 'distribution' element is a container for child elements that define frame match rules and frame handling rules.

Frame match rules determine whether the current frame matches (and is therefore handled by) this distribution. Frame handling rules define what action is performed on matching frames.

Use the 'protocols' element and/or the 'key' element to define frame match rules.

Use the 'action', 'key', 'queue', and 'combine' elements to define frame handling rules.

An 'action' element within a the distribution passes the frame to the specified Policy file element for further processing

The 'key', 'queue' and (optional) 'combine' elements within a distribution together provide inputs to a hash algorithm that distributes frames evenly over a range of frame queues. The 'key' element defines the protocol header fields to use as the hash key, the 'queue' element defines the base value and number of FQIDs in the frame queue range, and the optional 'combine' elements give you fine control over the exact FQIDs that the algorithm generates.

Note: You can use an 'action' element in the hash scenario described above to pass the frame to a policer profile, which may abort the enqueue operation and drop the frame if traffic conditions warrant. In the absence of an 'action' element, frame processing concludes (and the frame leaves the FMan) at the end of the 'distribution' element.

A distribution's frame queue ID calculation is performed as follows:

- A hash key is formed by extracting and concatenating the protocol header fields specified by the 'key' element.
- The result value is hashed to a 64-bit CRC.
- The number of least significant bits is taken based on the 'count' attribute of the 'queue' element.
- The resulting value is ORed with the data retrieved according to the 'combine' elements.
- The resulting value is ORed with the 'base' attribute value of the 'queue' element.

All child elements are optional. Appropriate hardware dependent default values are used in cases where a child element does not exist in the 'distribution' definition.

7.4.4.5.1.14.5.1 distribution Attribute Definitions

Table 145. distribution Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the distribution. Any references to a distribution are made using to this name.
description	optional	Free text describing the element purpose.
comment	optional	Free text providing any other information.

7.4.4.5.1.14.5.2 distribution Example

```
<distribution name="eth_dist" description="Ethernet protocol based distribution">
  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
  <combine portid="true" offset="10" mask="0xFF"/>
  <combine frame="112" offset="2" size="16" mask="0xFF"/>
  <action type="classification" name="eth_dest_clsf"/>
</distribution>
```

7.4.4.5.1.14.5.3 Default Groups

XML 'defaults' element is a container for parameters necessary for configuration of the default groups and private default registers. The element, if it exists, can be used as a child of element 'distribution'. This element contains a list of ‘default’ elements.

Table 146. 'default' Elements Attributes:

Attribute	Requirement	Description
private0	optional	The scheme default register 0.
private1	optional	The scheme default register 1.

Element 'default' attributes. This element can appear as a child to the element 'defaults':

Table 147. 'default' Element Attributes:

Attribute	Requirement	Description
type	required	Default type select. Possible values are: 1. "from_data” – any data extraction that is not one of the full fields that can be used as type. 2. "from_data_no_v” – any data extraction without validation. 3. "not_from_data” – extraction from parser result or direct use of default value. 4. "mac_addr” – MAC Address. 5. "tci” – TCI field. 6. "enet_type” – ENET Type. 7. "ppp_session_id” – PPP Session id. 8. "ppp_protocol_id” – PPP Protocol id. 9. "mpls_label” – MPLS Label. 10. "ip_addr” – IP Addr. 11. "protocol_type” – Protocol type. 12. "ip_tos_tc” – TOC or TC. 13. "ipv6_flow_label” – IPV6 flow label. 14. "ipsec_spi” – IPSEC SPI. 15. "l4_port” – L4 Port. 16. "tcp_flag” – TCP Flag
select	required	Default register select. Possible values are: 1. "gbl0” – Default selection is KG register 0. 2. "gbl1” – Default selection is KG register 1. 3. "private0” – Default selection is a per scheme register 0. 4. "private1” – Default selection is a per scheme register 1

Here is an example of possible default groups and nonheader definition:

```
<distribution name="Distribution1">
        <queue base="1" count="8"/>
        <key>
                <fieldref name="ipv4.src"/>
                <fieldref name="ipv4.dst"/>
                <fieldref name="ipv4.nextp"/>
                <nonheader source="default" offset="0" size="4"/>
        </key>
        <defaults private0="0xAAAAAAAA">
                <default type="from_data" select="private0"/>
                <default type="from_data_no_v" select="private0"/>
                <default type="not_from_data" select="private0"/>
        </defaults>
        <action type="drop"/>
</distribution>
```

7.4.4.5.1.14.6 The key element

The 'key' element contains a list of 'fieldref' elements. The 'filedref' elements define the protocol header fields whose values are concatenated to form a hash key. The Key Gen sub block hashes this key and uses a portion of the result in its frame queue ID (FQID) calculation.

7.4.4.5.1.14.6.1 key Attribute Definitions

Table 148. key Attribute Definitions

Attribute	Requirement	Description
shift	optional	Defines the amount by which the concatenation of the fields in the 'key' element are right shifted. The default value is zero. **Note:** The 'shift' attribute is ignored if the 'key' elements appears within a 'classification' element.
symmetric	optional	Generate the same hash for frames with swapped source and destination fields on all layers. If source is selected, destination must also be selected, and vice versa.

7.4.4.5.1.14.6.2 key Example

```
<key shift="16">
  <fieldref name="ethernet.src"/>
  <fieldref name="ethernet.dst"/>
</key>
```

7.4.4.5.1.14.7 The fieldref element

The 'fieldref' element refers to a protocol header field by its name.

The Standard Protocol file contains the names of the available protocols and their fields. This file is named hxs_pdl_v3.xml and is in the directory /etc/fmc/config/.

7.4.4.5.1.14.7.1 fieldref Attribute Definitions

Table 149. fieldref Attribute Definitions

Attribute	Requirement	Description
name	required	The referenced field name. The field's name should be provided in the form of "protocolname.fieldname".

7.4.4.5.1.14.7.2 fieldref Example

```
<key>
  <fieldref name="ethernet.src"/>
  <fieldref name="ethernet.dst"/>
</key>
```

7.4.4.5.1.14.8 The queue element

The 'queue' element defines the number of queues (default is one) and the base value for the FQIDs for these queues.

When used within a 'distribution' element, the 'queue' element defines a range of queues over which to evenly distribute frames.

When used within other elements, such as a 'classification' element, the 'queue' element defines the single queue on which to place a frame.

7.4.4.5.1.14.8.1 queue Attribute Definitions

Table 150. queue Attribute Definitions

Attribute	Requirement	Description
base	required	The base frame queue ID value.
count	optional	This attribute is only relevant only when a 'queue' element appears within a 'distribution' element. In this case, the 'count' attribute defines the number of frame queues over which to distribute frames. Valid values for 'count' are powers of 2. The default value is 1.

7.4.4.5.1.14.8.2 queue Example

```
<distribution name="eth_dist">
  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
</distribution>
```

7.4.4.5.1.14.9 The protocols and protocolref elements

The 'protocols' and 'protocolref' elements are used together to extend a 'distribution' element's frame match conditions.

As explained in the 'dist_order' description, a distribution is chosen based on the set of protocols specified in its 'key' element. The 'protocols' and 'protocolref' elements let you extend this set of protocols beyond those listed in the 'key' element.

7.4.4.5.1.14.9.1 protocols and protocolref Attribute Definitions

Table 151. protocols and protocolref Attribute Definitions

Attribute	Requirement	Description
name	required	The name of the protocol.
opt	optional	Applicable only for protocolref attribute Use it in a scheme for detecting protocols with the chosen options (e.g. to detect ETHERNET with BROADCAST or MULTICAST option) Table 2 contains all possible values. The values are grouped, each group being separated by a blank row. Values from different groups can be ORed

Table 152. Protocol options. Groups are separated by empty rows.

Value	Description
0x80000000	Ethernet Broadcast
0x40000000	Ethernet Multicast
0x20000000	Stacked VLAN
0x10000000	Stacked MPLS
0x08000000	IPv4 Broadcast
0x04000000	IPv4 Multicast
0x02000000	Tunneled IPv4 - Unicast
0x01000000	Tunneled IPv4 - Broadcast/Multicast
0x00000008	IPV4 reassembly option. When using this option, the IPV4 Reassembly manipulation requires network environment with IPV4 header
0x00800000	IPv6 Multicast

Table continues on the next page...

Table 152. Protocol options. Groups are separated by empty rows. (continued)

Value	Description
0x00400000	Tunneled IPv6 - Unicast
0x00200000	Tunneled IPv6 - Multicast
0x00000004	IPV6 reassembly option. When using this option, the IPV6 Reassembly manipulation requires network environment with IPV6 header.In case where fragment found, the fragment-extension offset may be found at 'shim2' (in parser-result).
0x00000008	CAPWAP reassembly option. When using this option, the CAPWAP Reassembly manipulation requires network environment with CAPWAP header. In case where fragment found, the fragment-extension offset may be found at 'shim2' (in parser-result).

7.4.4.5.1.14.9.2 protocols and protocolref Example

```
<!-- The example demonstrates the case in which -->
<!-- frame queue ID calculation is done using Ethernet header fields, -->
<!-- but the condition for matching a frame to this distribution is -->
<!-- extended by also requiring the presence of a UDP protocol header -->
<distribution name="eth_dist">
  <protocols>
    <protocolref name="udp" opt="0x00000008"/>
  </protocols>

  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
</distribution>
```

7.4.4.5.1.14.10 The combine element

The 'combine' element (like the 'key' element) is used in a 'distribution' element's frame queue ID calculation. The value built by the 'key' element is hashed, but the value of the 'combine' element is directly bitwised OR'd with the previous 24-bit FQID result.

A single 'combine' element identifies just one byte to retrieve and OR. To work around this limitation, you can have multiple 'combine' elements in a 'distribution' element.

7.4.4.5.1.14.10.1 combine Attribute Definitions

Table 153. combine Attribute Definitions

Attribute	Requirement	Description
portid	required *(in absence of frame attribute)*	Valid values: true or false If true, this attribute indicates that the logical port ID byte specified in the Configuration file should be retrieved and used in the bitwise OR part of a distribution's FQID calculation. *Note that portid and frame are mutually exclusive attributes.*
frame	required *(in absence of portid attribute)*	Valid values: numeric string This attribute identifies the byte with the frame header to extract and use in the bitwise OR part of the FQID calculation. The attribute's value indicates the bit offset from the beginning of the frame. The specified value must be divisible by 8, so it references the first bit of a byte. *Note that portid and frame are mutually exclusive attributes.*
offset	optional	This attribute controls the placement of the extracted data in the result Frame Queue ID. The offset starts at the FQID's most significant bit.
mask	optional	This attribute defines valid bits in the retrieved value. The extracted value is bitwise ANDed with the mask prior to being ORed with the previous Frame Queue ID value.

7.4.4.5.1.14.10.2 combine Example

```
<distribution name="eth_dist">
  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
  <combine portid="true" offset="10" mask="0xFF"/>
  <combine frame="64" offset="2" mask="0xFF"/>
  <action type="classification" name="eth_dest_clsf"/>
</distribution>
```

7.4.4.5.1.14.11 The action element (for use in a policy file)

The 'action' element permits you to establish a topological parse, classify, police, distribute configuration by defining the next processing element within a distribution, classification, or policer profile.

If there is no 'action' element within a distribution, classification, or policer profile, the default behavior is the completion of PCD frame processing, allowing the frame to leave the Frame Manager. Some hardware restrictions apply in the choice of the next processing element.

7.4.4.5.1.14.11.1 action Attribute Definitions

Table 154. action Attribute Definitions

Attribute	Requirement	Description
type	required	The type of the 'action' element defines the next processing element. Valid values are: • "distribution" • "classification" • "policer" • "drop" (Permitted only when the 'action' element is inside a 'policer' element.)
name	required	The name of the element of the type defined in the 'type' attribute. This attribute is not relevant if type is "drop".
condition	required *(when used within a 'policer' element)* optional *(when used within a 'distribution' or 'classification' element)*	This attribute defines the condition under which the 'action' is to be taken. This attribute is only relevant when used inside a 'policer' or a 'classification' element. Valid values are: • "on-green" • "on-yellow" • "on-red" • "on-miss"

7.4.4.5.1.14.11.2 Statistics

Attribute 'statistics' for action element of the classification and classification entries. This tells if statistics are made on that entry or on the on-miss.

Table 155. 'statistics' Element Attributes:

Attribute	Requirement	Description
statistics	optional	Enable statistics for a particular action. Possible values are: • enable/yes/true – to enable it. • disable

7.4.4.5.1.14.11.3 action Example

```
<distribution name="special_dist">
  <queue count="1" base="0xABCD"/>
  <action type="policer" name="policer2"/>
</distribution>

<policer name="policer2">
  <algorithm>rfc2698</algorithm>
  <color_mode>color_aware</color_mode>
  <CIR>1000000</CIR>
  <EIR>1400000</EIR>
  <CBS>1000000</CBS>
```

```
  <EBS>1400000</EBS>
  <unit>packet</unit>
  <action condition="on-green" type="distribution" name="special2_dist"/>
  <action condition="on-yellow" type="drop"/>
  <action condition="on-red" type="drop"/>
</policer>
```

7.4.4.5.1.14.12 The classification element

The 'classification' element allows exact match frame processing.

A classification starts with a 'classification' element, which is a container for these child elements:

- A 'key' element that defines the header fields (in protocol.field form) to use in the exact match operation
- One or more 'entry' elements, each of which defines a value to which the specified fields are compared and a 'queue' and/or 'action' element that defines what to do with the frame upon a match
- An optional 'action' element that defines the default action to take if none of the exact match conditions is met

7.4.4.5.1.14.12.1 classification Attribute Definitions

Table 156. classification Attribute Definitions

Attribute	Requirement	Description
name	required	The name of the classification

7.4.4.5.1.14.12.2 classification Statistics

The statistics are enabled on the Classification element. The parameters to setup the statistics are: - the attribute **statistics** of the element **classification**, the attribute **statistics** of the actions on entries/on-miss and the element **framelength** with attributes **index** and **value**.

Attribute ‘statistics’ for classification – this specifies the type of statistic used in the entire classification

Table 157. 'statistics' Element Attributes:

Attribute	Requirement	Description
statistics	optional	Choose statistic mode for the particular entry. Possible values are: • none • frame • byteframe • rmon

7.4.4.5.1.14.12.3 classification Example

```
<classification name="eth_dest_clsf">
  <key>
    <fieldref name="ethernet.dst"/>
  </key>

  <entry>
    <data>0x1234567890AB1234567890AB</data>
```

```
    <queue base="0x550000"/>
  </entry>

  <entry>
    <data>0xFFFFFFFFFFFFFFFFFFFFFFFF</data>
    <action type="classification" name="eth_dest_2_clsf"/>
  </entry>

  <action condition="on-miss" type="distribution" name="default_dist"/>
</classification>
```

7.4.4.5.1.14.12.4 Frame Replicators

The element **replicator** is implemented in FMC as a standalone entity.

This element can follow a Classification in the flow, as a target for one of the actions of the entries or on the on-miss. It is similar to Classification but it has no data/mask in entries, on-miss action and key element.

Table 158. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the frame replicator.
max	optional	The maximum number of entries the frame replicator can have (default and minimum is 2). If the value entered is smaller than 2 or the attribute is not set, the value is set to 2.

The element **entry** has the same syntax as the element **classification**, but the data and mask are not needed and thus are ignored. The action targets of the entry are restricted to:

- policer
- enqueue
- direct distribution

replicator example:

```
  <replicator name="frep_1" max="32">
       <entry>
           <action type="policer" name="policer_1"/>
       </entry>
       <entry>
           <queue base="0x0"/>
           <action type="distribution" name="dist_1"/>
       </entry>
       <entry>
           <queue base="0x220"/>
           <vsp name=”vsp01”>
       </entry>
       <entry>
           <queue base="0x240"/>
           <vsp base=”2”>
       </entry>
  </replicator>
```

Using the frame replicator in an action:

```
   <classification name="class_1" max="0" masks="yes">
        <key>
           <fieldref name="ethernet.type"/>
        </key>
        <entry>
           <data>0x8870</data>
           <queue base="0x01"/>
           <action type="replicator" name="frep_1"/>
        </entry>
        <action condition="on-miss" type="replicator" name="frep_1"/>
   </classification>
```

7.4.4.5.1.14.12.5 framelength Statistics

Element **framelength** attributes (there can be up to 10 values set, in ascending order and last one must be `0xFFFF`). The element **framelength** is valid only for RMON statistics.

Table 159. 'framelength' Element Attributes:

Attribute	Requirement	Description
statistics	required	The index for the frame length value specified. Possible values are from 0 to 9.
value	required	The value to be added at the specified index. Maximum value is `0xFFFF` and must be added at index 9. (FMC sets it initially by default).

7.4.4.5.1.14.12.6 Statistics Example

Statistics Example

```
<!-- Coarse classification -->
    <classification name="classif_1" max="32" masks="yes" statistics="rmon">
        <!-- Key value to be extracted from the packet -->
        <key>
            <fieldref name="ipv4.dst"/>
        </key>

        <framelength index="0" value="0x1100"/>
        <framelength index="1" value="0x1200"/>
        <framelength index="2" value="0x1300"/>
        <framelength index="3" value="0x1400"/>
        <framelength index="4" value="0x1500"/>
        <framelength index="5" value="0x1600"/>
        <framelength index="6" value="0x1700"/>
        <framelength index="7" value="0x1800"/>
        <framelength index="8" value="0x1900"/>
        <framelength index="9" value="0xFFFF"/>

        <!-- Entries in the lookup table -->
        <entry>
            <!-- 192.168.10.10 -->
            <data>0xC0A80A0A</data>
            <queue base="0x1010"/>
```

```
            <action statistics="enable"/>
        </entry>
    </classification>
```

7.4.4.5.1.14.12.7 Coarse Classification Resource Reservation

FMD API changes allow pre-allocation of MURAM memory for classification tables. This will be reflected in NetPCD XML syntax extension by introducing attibutes **max** and **masks** of the element **classification** as shown in the example below. In addition, to allow proper order of PCD elements initialization, and for the condition that not all **entry** elements are known at initialization time, the XML element **may-use** is introduced:

```
    <!-- Coarse classification -->
    <classification name="classif_1" max="32" masks="yes" statistics="mode">
        <!-- Key value to be extracted from the packet -->
        <key>
            <fieldref name="ipv4.dst"/>
        </key>

        <may-use>
            <action type="classification" name="fman_test_classif_1"/>
            <action type="distribution"   name="default_dist"/>
        </may-use>

        <!-- Entries in the lookup table -->
        <entry>
            <!-- 192.168.10.10 -->
            <data>0xC0A80A0A</data>
            <queue base="0x1010"/>
        </entry>
    </classification>
```

Resource Allocation Attributes:

Table 160. Resource Reservation Attributes:

Attribute	Requirement	Description
max	optional	If it exists, this parameter defines the maximum number of coarse classification entries allocated for this PCD element. **NOTE** The element **classification** may still contain pre-initialized entries, or, alternatively, be empty. **NOTE** For the case of empty or partially initialized element **classification**, usage of the element **may-use** might be required .
masks	optional	If provided, indicates that MURAM allocation should be done with the assumption that additional memory is required for an elements' masks. Possible values are: • no – don't allocate memory for masks (default) • yes – allocate memory for masks.

'may-use Element Description:

Table 161. 'may-use' Element Attributes:

Attribute	Requirement	Description
may-use	optional	Contains list of ‘action’ elements that may appear in the ‘classification’ entries or, be applied dynamically after partial initial configuration. **NOTE** Attention: the use of this element is required if initial ‘classification’ is empty and dynamic entries, added through FMD API, use those PCD entities

7.4.4.5.1.14.13 The entry element

The 'entry' element defines:

- the value to use in an exact match comparison with the fields specified by the 'key' element in a classification
- the action to be taken upon a match

An 'entry' element contains a 'data' element which, in turn, contains a numeric value written in hexadecimal form (that is, with a "0x" prefix). The data length of this value is determined by length of the set of 'key' fields.

In addition to the 'data' element, each 'entry' element may also contain these elements:

- queue - causes the frame to be placed on the specified queue
- action - passes the frame to the specified element within the Policy file for further processing.
- mask - a value in hexadecimal format that is applied to the data element

7.4.4.5.1.14.13.1 entry Attribute Definitions

Table 162. entry Attribute Definitions

Attribute	Requirement	Description
none	n/a	n/a

7.4.4.5.1.14.13.2 entry Example

```
<classification name="eth_dest_clsf">
  <key>
    <fieldref name="ethernet.dst"/>
  </key>

  <entry>
    <data>0x1234567890AB1234567890AB</data>
    <queue base="0x550000"/>
  </entry>
</classification>
```

7.4.4.5.1.14.14 The policer element

The 'policer' element is a container whose child elements define a policer profile that performs network bandwidth management.

7.4.4.5.1.14.14.1 policer Attribute Definitions

Table 163. policer Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the policer profile.
algorithm	required	Algorithm used for policing. Valid values: "rfc2698", "rfc4115", pass_through".
color_mode	required	Color mode used for policing. Valid values: "color_aware", "color_blind".
default_color	optional	Use when algorithm is "pass_through" and color_mode is "color_blind". In this mode, the policer re-colors incoming packets with the specified default color. Valid values: "red", "yellow", "green", or "override". If the value is override, the next invoked action is that specified for "green". The default value is "green".
unit	required	The unit to be used for numeric parameters. Valid values: "packet", "byte".
CIR	required	Committed information rate[12]
PIR	required	Peak (or excess) information rate[12]
CBS	required	Committed burst size[13]
PBS	required	Peak (or excess) burst size[13]

7.4.4.5.1.14.14.2 policer Example

```
<policer name="policer2">
  <algorithm>rfc2698</algorithm>
  <color_mode>color_aware</color_mode>
  <CIR>1000000</CIR>
  <EIR>1400000</EIR>
  <CBS>1000000</CBS>
  <EBS>1400000</EBS>
  <unit>packet</unit>
  <action condition="on-green" type="distribution" name="default_dist"/>
  <action condition="on-yellow" type="distribution" name="special2_dist"/>
  <action condition="on-red" type="drop"/>
</policer>
```

7.4.4.5.1.14.15 The nonheader element

Use the 'nonheader' element within a 'key' element to select a non-header extraction source.

Note: The 'nonheader' element can appear within a 'classification' element only. Further, the 'nonheader' element cannot be used at the same time as the 'fieldref' element.

7.4.4.5.1.14.15.1 nonheader Attribute Definitions

[12] If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second.
[13] If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.

Table 164. nonheader Attribute Definitions

Attribute	Requirement	Description
source	required	Non-header extraction source Valid values are: • "frame_start" - Extract from beginning of frame. • "key" - Extract from key value built by ‘distribution’ at preceding step (CC only). • "hash" - Extract from hash value built by ‘distribution’ at preceding step (CC only). • "parser" - Extract from parse result array. • "fqid" - Use enqueue FQID as the key value. • "flowid" - Use dequeue FQID as the key value (CC only) • "default" - Extract from a default value (distribution only). • "endofparse" - Extract from the point where parsing had finished (distribution only).
action	Required if source is "hash", "flowid" or "key". In other cases, this attribute must not be used.	The type of action for the extraction Valid values are: • "indexed_lookup" (permitted only for "hash" and "flowid" sources). The extracted value is interpreted as an entry index of classification table • "exact_match" (permitted only for "key" and "hash" sources). The extracted value is compared with ‘key’ value of the entry.
offset	required	Byte offset. Offset of key from start of frame, internal frame context or parse result array. Refer “Table 8-398. Table Descriptor (Type = 01)” of DPAA Reference Manual for full description and possible values
size	required	Size of the key in bytes.
ic_index_mask	Optional (Valid only if action is "indexed_lookup")	Internal context index mask. For the full description and possible values, refer “Table 8-399. Operation Code Description” of DPAA Reference Manual

If the action is “indexed_lookup” and the source is “hash” special checks are done in the drivers on the configured entries and maximum nuber of entries according to the internal context index mask specified. FMC is adjusting automatically the configured entries if they don’t match the provided mask: if the entry must be initialized but the user didn’t supplied it a default one is created and if the entry must be uninitialized it’s deleted by FMC. Also FMC ajdusts the maxim number of entries if it’s not configured as 0.

7.4.4.5.1.14.15.2 nonheader Example

```
<classification name="ptp_condition_class">
  <key>
    <nonheader source="hash" action="indexed_lookup" offset="2" size="2" ic_index_mask="0x01b0">
```

```
  </key>

  <entry>
    <data>0x13F</data>
    <queue base="0x01"/>
  </entry>
</classification>
```

7.4.4.5.1.14.16 Hash Tables

The element 'hashtable' can be specified inside an element 'key' of a 'classification'. The element 'hashtable' cannot appear in the same time with either elements 'fieldref' or 'nonheader' in the same 'key'. If the element 'hashtable' is used, the 'classification' may have no entries as these are supposed to be filled at runtime.

Table 165. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
mask	required	Mask that will be used on the hash-result; The number-of-sets for this hash will be calculated as (2^(number of bits set in 'mask ')); The 4 lower bits must be cleared.
hashshift	optional	Byte offset from the beginning of the KeyGen hash result to the 2-bytes to be used as hash index.(Default 0)
keysize	required	Size of the exact match keys held by the hash buckets.

Hash table example:

```
<classification name="classif_1" max="2" statistics="none">
        <key>
            <hashtable mask="0x30" hashshift="0" keysize="24"/>
        </key>
</classification>
```

7.4.4.5.1.14.17 Virtual Storage Profiles Element

The element 'vsp' (Virtual Storage Profile) is implemented in FMC as a standalone entity or can be defined directly in the element that uses it. The element 'vsp'can be used inside distributions, classification and entries (both classification and replicator). When used directly in the ‘classification’ element (not in ‘entry’) it counts for the on-miss action. If the 'action' of the 'entry' or on-miss goes to another 'classification' or 'replicator' the 'vsp' is ignored.

7.4.4.5.1.14.17.1 vsp Attributes

Table 166. 'vsp' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the virtual storage profile inside the elements that are using it.

Table continues on the next page...

Table 166. 'vsp' Element Attributes: (continued)

Attribute	Requirement	Description
type	optional	The type of the VSP. Values: • direct – (default) the relative profile ID is selected directly by the 'base' attribute. • indirect – the relative profile ID is selected base on the attributes **fqshift, vspoffset**, and **vspcount** can be used only in **distribution.**
base	required for direct.	--
fqshift	required for indirect.	Shift of KeyGen results without the FQID base.
vspoffset	optional for indirect	OR of KeyGen results without the FQID base; should indicate the storage profile offset within the port's storage profiles window.
vspcount	optional for indirect	Range of profiles starting at base.

7.4.4.5.1.14.17.2 vsp Examples

VSP examples (standalone, defined in element, direct/indirect): The action targets of the entry are restricted to:

```
<vsp name = "storage01" base = "6"/>
<vsp name = "storage02" type = "indirect" fqshift="2" vspoffset="3"        vspcount="8"/>
<vsp name = "storage03" type = "direct" base = "7"/>

Usage:

...

<entry>
            <queue base="0x220"/>
            <vsp name=”storage01”>
</entry>

...

<distribution name="dist1">
     ...
     <queue count="8" base="0x230"/>
     <vsp type=”indirect” fqshift=”2” vspoffset=”0” vspcount=”4”/>
     ...
</distribution>

...

<classification name="eth_dest_clsf">
    <key>
         <fieldref name="ethernet.dst"/>
    </key>
...
    <vsp name=”storage03”>
    <action condition="on-miss" type="distribution" name="garbage"/>
</classification>
```

7.4.4.5.1.14.18 Manipulation Parameters

Frame Manager accelerator (FMan) attaches manipulation actions as an extension to ethernet port and coarse classification ‘next engine’ dispatch activity.

To reflect the frame data processing and manipulation capabilities of the hardware, which are propagated through Frame Manager Driver (FMD) API, Frame Manager Configuration (FMC) Tool extends the syntax of the NetPCD configuration language by introducing XML entities described in this document.

Manipulation entities are diverse in their purpose and configuration parameters sets. The same manipulation entity can be referred, or attached, from/to several port or classification actions. That is why they are separated from their usage into a separate group called **manipulations**. At the moment of use, an action refers to the corresponding manipulation entity. For example:

```
<netpcd>
    <manipulations>
        <reassembly name=”name1”>
            .....
        </reassembly>
        <reassembly name=”name2”>
            .....
        </reassembly>
        <fragmentation name=”defrag1”>
            .....
        </fragmentation>
    </manipulations>

    <classification name=”clsf1”>
        .....
        <!-- 192.168.30.30 -->
        <data>0xC0A81E1E</data>
        <fragmentation name=”defrag1”/>
        .....
    </classification>

</netpcd>
```

Formal Definition:

XML element **manipulation** is a container for all types of manipulation algorithms. Configuration for each algorithm has its own XML element name.

Currently three manipulations algorithms are available:

1. IP reassembly
2. IP fragmentation
3. header manipulation

Parameters for these entities are described next.

7.4.4.5.1.14.18.1 IP Fragmentation

XML element **fragmentation** is a container for parameters necessary for configuration of the corresponding action modification. The element, if exists, can be used as a child of element **classification**.

Attention: If element **fragmentation** is present together with other ‘action’ of ‘classification’ element, the element **fragmentation** is ignored. This is a subject of FMan firmware capabilities and may change in future.

Table 167. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the manipulation algorithm.

Table 168. 'fragmentation' Child Elements:

Attribute	Requirement	Description
size	required	IP fragmentation will be executed for frames with length greater than this value.
dontFragAction	optional	If an IP packet is larger than MTU and its DF bit is set, then this field will determine the action to be taken. Possible values are: • discard - the packet (default action) • fragment – fragment the packet and continue normal processing • continue - continue normal processing without fragmenting the packet
scratchBpid	required for existing HW platforms, but not for 9164	Absolute buffer pool id according to BM configuration (DPAA 1.0 only)
sgBpid	optional	Scatter/Gather buffer pool id. If used sgBpidEn will be set to TRUE.
optionsCounterEn	optional	Enables the counter if the value is set to 'yes', 'true' or 'enable'. Disabled for other values. Default is disabled.

Here is an example of possible IP fragmentation definition:

```
<manipulations>
    <fragmentation name="frag1">
        <size>256</size>
        <dontFragAction>continue</dontFragAction>
    </fragmentation>
</manipulations>

<classification name="clsf1">
    .....
    <!-- 192.168.30.30 -->
    <data>0xC0A81E1E</data>
    <fragmentation name="frag1"/>
    .....
</classification>
```

7.4.4.5.1.14.18.2 IP Reassembly

XML element **reassembly** is a container for parameters necessary for configuration of the corresponding action modification. The element, if it exists, can be used as a child of the element **policy**.

Attention: Up to 2 additional KeyGen schemes will be constructed when using this manipulation action. Custom protocol **shim2** is reserved when element **reassembly** participates in a configuration.

Table 169. 'reassembly' Element Attributes:

Attribute	Requirement	Description
Name	required	Name of the element. The name is used to refer the manipulation algorithm

Table 170. 'reassembly' Child Elements:

Attribute	Requirement	Description
sgBpid	required	Absolute buffer pool id according to BM configuration for scatter-gather (DPAA 1.0 only)
maxInProcess	required	Number of frames which can be processed by reassembly at the same time. It has to be power of 2
dataLiodnOffset	optional	Offset of LIODN. Default value is 0
dataMemId	optional	Memory partition ID for data buffers
ipv4minFragSize	required	Minimum fragmentation size for IPv4
ipv6minFragSize	required	EMinimum fragmentation size for IPv6. The value must be equal or higher than 256
timeOutMode	optional	Expiration delay initialized by Reassembly process. Possible values are: • frame - limits the time of the reassembly process from the first fragment to the last (default) • fragment - limits the time of receiving the fragment
fqidForTimeOutFrames	required	FQID to assign for frames enqueued during Time Out Process.
numOfFramesPerHashEntry (numOfFramesPerHashEntry1)	required	Number of frames per hash entry needed for reassembly process – for ipv4. Possible values are: numeric values from 1 to 8.
numOfFramesPerHashEntry2	optional	Number of frames per hash entry needed for reassembly process – for ipv6. Possible values are: numeric values from 1 to 6.
timeoutThreshold	required	Represents the time interval in microseconds which defines if opened frame (at least one fragment was processed but not all the fragments)is found as too old
nonConsistentSpFqid	optional	Handles the case when other fragments of the frame corresponds to a different storage profile than the opening fragment. (DPAA >= 1.1 only). Default is 0

Here is an example of possible IP reassembly definition:

```
<manipulations>
    <reassembly name=”reasm1”>
        <sgBpid>2</sgBpid>
        <maxInProcess>1024</maxInProcess>
        <timeOutMode>fragment</timeOutMode>
        <fqidForTimeOutFrames>1024</fqidForTimeOutFrames>
```

```
        <numOfFramesPerHashEntry>8</numOfFramesPerHashEntry>
        <timeoutThreshold>1000000</timeoutThreshold>
        <ipv4minFragSize>0</ipv4minFragSize>
        <ipv6minFragSize>256</ipv6minFragSize>
    </reassembly>
</manipulations>

<policy name="udp_port">
   <dist_order>
       <distributionref name="custom_dist"/>
       <distributionref name="udp_port_dist"/>
       <distributionref name="default_dist"/>
   </dist_order>

   <reassembly name=”reasm1”/>
</policy>
```

7.4.4.5.1.14.18.3 Header Manipulation

XML element **header** is a container for parameters necessary for configuration of the corresponding action modification. The element, if it exists, can be used as parameter to the distribution action going to a classification or inside a classification element **entry**.

The XML element **header** may contain:

- **insert**
- **remove**
- **insert_header**
- **remove_header**
- **update**
- **custom**

Certain combinations between them are possible, for example you can have a **remove** and an **insert_header** in the same manipulation.

The header manipulation can be used inside the PCD by inserting an element **header** in the classification entry that specifies the name of the header manipulation defined in the section **manipulations**. This makes sense in a entry that goes to a policer, distribution or PCD done:

```
   <entry>
      <data>0x9100</data>
      <queue base="0x01"/>
      <action type="policer" name="plcr_01"/>
      <header name="upd_hdr"/>
   </entry>
```

Table 171. 'header' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the manipulation algorithm
parse	optional	Activate the parser a second time after completing the manipulation of the frame (if ‘yes’)

Table continues on the next page...

Table 171. 'header' Element Attributes: (continued)

Attribute	Requirement	Description
duplicate	optional	Will duplicate the header manipulation with the same setting a the specified number of times. The names of the nodes will have “_x” added at the end where x is the index of the node. For example <header name=”upd_ipv4” duplicate=”3”> will create the nodes: upd_ipv4_1, upd_ipv4_2 and upd_ipv4_3. This is only a simple tool to duplicate a header manipulation, it does not allow defining chaining between the elements created by duplication.

7.4.4.5.1.14.18.3.1 Header Manipulation - Insert

XML element **insert** is a container for parameters necessary to configure a header insert manipulation operation. The element, if it exists, can be used as a child of element **header**. There can be only one element **insert** in a header manipulation.

Table 172. 'insert' Child Elements:

Element	Requirement	Description
size	required	Size of inserted section
offset	required	Offset from beginning of header to the start location of the insertion.
replace	optional	If provided, specifies to override (replace) existing data at 'offset' (if ‘yes’), ‘no’ to insert. Possible values: • no - insert (default) • yes - replace
data	required	Data to insert

7.4.4.5.1.14.18.3.2 Header Manipulation - Remove

XML element **remove** is a container for parameters necessary to configure a header remove manipulation operation. The element, if it exists can be used as a child of element **header**. There can only be one element **remove** in a header manipulation.

Table 173. 'remove' Child Elements:

Element	Requirement	Description
size	required	Size of removed section
offset	required	Offset from beginning of header to the start location of the removal.

7.4.4.5.1.14.18.3.3 Header Manipulation - Insert-Header

XML element **insert_header** is a container for parameters necessary to configure a header insert manipulation operation of an entire header (different than generic element **insert**). The element **insert_header** ,if it exists, can be used as a child of element **header**. With some restrictions, there can be more than one element **insert_header** in one header manipulation

Table 174. 'insert_header' Element Attributes

Element	Requirement	Description
type	required	The type of the header inserted. Only ‘mpls’ is valid at this time.
header_index	optional	The header index of the header has possible values "1" and "2". The restrictions on this attribute are: • if the value is ‘2’ an ‘insert_header’ with ‘header_index’ 1 must be present in the header manipulation. • a value of **header_index** can be used only once per header manipulation

Table 175. 'insert_header' Child Elements

Element	Requirement	Description
data	optional	The data of the header to be inserted.
replace	optional	If provided, specifies to override (replace) existing data (if ‘yes’), ‘no’ to insert.

insert_header example:

```
<header name="insert_2_l2">
    <insert_header type="mpls" header_index="1">
        <data>0x00000048</data>
    </insert_header>
    <insert_header type="mpls" header_index="2">
        <data>0x00000048</data>
    </insert_header>
</header>
```

7.4.4.5.1.14.18.3.4 Header Manipulation - Remove_Header

XML element **remove_header** is a container for parameters necessary to configure a header remove manipulation operation of an entire header (different then element **remove** that is a generic one). The element, if it exists, can be used as a child of element **header**'. There can be only one instance of element **remove_header** in a manipulation and it cannot appear in the same time with the generic **remove**.

Table 176. 'remove_header' Child Elements

Element	Requirement	Description
type	required	The type of the header remove. Possible values: • "qtags" • "mpls" • "ethmpls (or "ethernet_mpls") • "eth" (or "ethernet")

remove_header example:

```
<header name="remove_l2">
    <remove_header type="qtags/>
</header>
```

7.4.4.5.1.14.18.3.5 Header Manipulation - Update

XML element **update** is a container for parameters necessary to configure a header update manipulation. The element if exists can be used as a child of element **header**. There can be only one update in a header manipulation.

update Element Attributes:

Table 177. 'remove_header' Child Elements

Element	Requirement	Description
type	required	The type of the update. Possible values: • "vlan" • "ipv4" • "ipv6" • "tcpudp"

update Child Elements:

Table 178. 'remove_header' Child Elements

Element	Requirement	Description
field	required	Specifies the field to be updated. There must be atleast one inside an update. For some types of updates the field element can appear multiple times.

Field Element Attributes:

Table 179. 'remove_header' Child Elements

Element	Requirement	Description

type	required	The type of the header remove. Possible values:
		• for 'vlan'
		  • dscp - DSCP to VLAN priority bits translation.
		  • vpri - Replace VPri of outer most VLAN tag .
		• for 'ipv4'
		  • tos - update TOS with the given value.
		  • id - update IP ID with the new 16 bit given value.
		  • ttl - Decrement TTL by 1.
		  • src - update IP source address with the given value.
		  • dst - update IP destination address with the given value.
		• for 'ipv6'
		  • tc - update Traffic Class address with the given value.
		  • hl - Decrement Hop Limit by 1.
		  • src - update IP source address with the given value.
		  • dst - update IP destination address with the given value.
		• for 'tcpudp'
		  • checksum - update TCP/UDP checksum.
		  • src - update TCP/UDP source address with the given value.
		  • dst - update TCP/UDP destination address with the given value.

value	optional	The value used for the update. It is not valid for: • hl • ttl • checksum
fill	optional	Only valid for **dscp** - fills the entire array with the given value. The fill is performed before the other **dscp** operations.
index	optional	Only valid for **dscp**. Speciefies the index in the array where that value is set. The index starts from 0.

'update' Example:

```
<header name="upd_checksum">
    <update type = "tcpudp">
        <field type="checksum"/>
    </update>
</header>

<header name="upd_ipv4src">
    <update type = "ipv4">
        <field type="src" value="0xC0A80101"/>
    </update>
```

```
</header>

<header name="upd_vpri">
    <update type = "vlan">
        <field type="dscp" fill="yes" value="4"/>
        <field type="dscp" index="20" value="2"/>
        <!--...-->
        <field type="dscp" index="30" value="2"/>
    </update>
</header>
```

7.4.4.5.1.14.18.3.6 Header Manipulation - Custom

XML element **custom** is a container for parameters necessary to configure custom header manipulation. The custom header manipulation supported by the drivers is now custom IP replace, and allows changing between ipv4 and ipv6.

'custom' Element Attributes

Table 180. 'custom' Element Attributes:

Element	Requirement	Description
type	required	The type of the custom header manipulation. Possible values are: • “ipv4byipv6” (or just “ipv4”) – Replaces ipv4 by ipv6. • -“ipv6byipv4” (or just “ipv6”) – Replaces ipv6 by ipv4.

'custom' Child Elements

Table 181. nextmanip Element Attributes:

Element	Requirement	Description
size	required	Size of the header to be inserted. (max is 256)
data	required	The header data to be inserted.
decttl	optional	Decrement TTL by 1 (ipv4). Possible values: • "yes" • "no"
dechl	optional	Decrement Hop Limit by 1 (ipv6). Possible values: • "yes" • "no"
ip (or 'ipid')	optional	16 bit New IP ID (ipv4)

'custom' Example:

```
<header name="custom_ex">
     <custom type="ipv6byipv4">
          <decttl>yes</decttl>
          <id>1</id>
          <size>0x20</size>
          <data>0x450000001234000000010000101111213141516 17</data>
     </custom>
</header>
```

7.4.4.5.1.14.18.3.7 Header Manipulation - Nextmanip

XML element **nextmanip** Can be used to setup cascading header manipulations. It relates to the header manipulation element and not sub-elements (insert, remove and update).

Table 182. Nextmanip element attributes

Element	Requirement	Description
name	required	The name of the next header manipulation

7.4.4.5.1.14.18.3.8 Header Manipulation - Example

Here is a general example of possible header manipulation definition:

```
<manipulations>
    <header name=”ins_rmv” parse=”yes”>
        <insert>
            <size>14</size>
            <offset>0</offset>
            <data>0x01020304050611121314151681 00</data>
        </insert>
        <remove>
            <size>14</size>
            <offset>0</offset>
        </remove>
    </header>

    <header name="vpri_update">
        <update type="vlan">
            <field type="vpri" fill="yes" value="0"/>
        </update>
    </header>

    <header name=”ins_vlan” parse=”no”>
        <insert>
            <size>4</size>
            <offset>12</offset>
            <data>0x81004416</data>
        </insert>
        <nextmanip name="vpri_update"/>
    </header>
</manipulations>


<classification name="clsf_1" max="0" masks="yes" statistics="none">
   <key>
      <fieldref name="ethernet.type”/>
   </key>
   <entry>
      <data>0x8847</data>
      <queue base="0x01"/>
      <action type="policer" name="plcr_1"/>
      <header name="ins_vlan"/>
   </entry>
   <entry>
      <data>0x8848</data>
      <queue base="0x02"/>
      <header name="ins_rmv"/>
```

```
   </entry>
</classification>
```

7.4.4.5.1.15 Standard Protocol File - Excerpt

The DPAA SDK includes a file called the Standard Protocol file. This file uses the NetPDL (Network Protocol Description Language) XML dialect to define the fields in each standard protocol header that the FMan can parse with its Hard Parser. In addition, for each protocol, the NetPDL statement define the actions the Hard Parser should take upon encountering this protocol header in the frame window.

For this reason, the SDK includes a copy of the Standard Protocol file here: /etc/fmc/config/hxs_pdl_v3.xml. In addition, to give you an idea what the file is like, a small portion is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<netpdl name="nbee.org NetPDL Database"
  version="0.2" creator="nbee.org" date="28-05-2008">
<!-- This file is for reference only. -->
<!-- It describes the protocols and fields supported by the FMan's Hard Parser-->

<!--
NetPDL description of the Ethernet Protocol
-->
<protocol name="ethernet" longname="Ethernet 802.3"
  comment="Ethernet DIX has been included in 802.3" showsumtemplate="ethernet">

  <execute-code>
    <!-- If we're on Ethernet IEEE 802.3, update the packet length -->
    <after when="buf2int(type) le 1500">
       <assign-variable name="$packetlength" value="buf2int(type) + 14"/>
       <!-- 14 is the size of the ethernet header -->
    </after>
  </execute-code>

  <format>
    <fields>
      <field type="fixed" name="dst" longname="MAC Destination" size="6"
        showtemplate="MACaddressEth"/>
      <field type="fixed" name="src" longname="MAC Source" size="6"
                          showtemplate="MACaddressEth"/>
      <field type="fixed" name="type" longname="Ethertype - Length" size="2"
    </fields>
  </format>

  <encapsulation>
    <!-- We have four possible encapsulations for IPX:
       - Ethernet version II
         ==> type= 0x8137
       - Novell-specific framing (raw 802.3)
         ==> directly in Ethernet; check that IPX checksum is == 0xFFFF
       - Ethernet 802.3/802.2 without SNAP
         ==> directly in SNAP; check that IPX checksum is == 0xFFFF (after SNAP hdr)
       - Ethernet 802.3/802.2 with SNAP
         ==> type= 0x8137 (in SNAP)
       See the "IPX Ethernet and FDDI Encapsulation Methods" Cisco doc, at:
       http://www.cisco.com/en/US/tech/tk389/tk224/
         technologies_q_and_a_item09186a0080093d2e.shtml
    -->
    <if expr="buf2int($packet[$currentoffset:2]) == 0xFFFF">
      <if-true>
```

```
        <nextproto proto="#ipx"/>
      </if-true>
    </if>
    <switch expr="buf2int(type)">
      <case value="0" maxvalue="1500"> <nextproto proto="#llc"/> </case>
      <case value="0x800"> <nextproto proto="#ip"/> </case>
      <case value="0x806"> <nextproto proto="#arp"/> </case>
      <case value="0x8863"> <nextproto proto="#pppoed"/> </case>
      <case value="0x8864"> <nextproto proto="#pppoe"/> </case>
      <case value="0x86DD"> <nextproto proto="#ipv6"/> </case>
      <case value="0x8100"> <nextproto proto="#vlan"/> </case>
      <case value="0x8137"> <nextproto proto="#ipx"/> </case>
      <case value="0x81FD"> <nextproto proto="#ismp"/> </case>
      <case value="0x8847" comment="mpls-unicast">
         <nextproto proto="#mpls"/>
      </case>
      <case value="0x8848" comment="mpls-multicast">
         <nextproto proto="#mpls"/>
      </case>
    </switch>
  </encapsulation>

  <visualization>
    <showsumtemplate name="ethernet">
      <section name="next"/>
      <text value="Eth: "/>
      <protofield name="src" showdata="showvalue"/>
      <text value=" => "/>
      <protofield name="dst" showdata="showvalue"/>
    </showsumtemplate>
  </visualization>

</protocol> <!-- End Ethernet protocol definition -->

<!--
NetPDL description of the VLAN Protocol
-->
<protocol name="vlan" longname="Virtual LAN (802.3ac)" showsumtemplate="vlan">
  <format>
    <fields>
      <block name="vlan" size="2" longname="Tag Control Information">
        <field type="bit" name="pri" longname="User Priority"
          mask="0xE000" size="2" showtemplate="FieldHex"/>
        <field type="bit" name="cfi" longname="CFI"
          mask="0x1000" size="2" showtemplate="FieldDec"/>
        <field type="bit" name="vlanid" longname="VLAN ID"
          mask="0x0FFF" size="2" showtemplate="FieldDec"/>
      </block>
    <field type="fixed" name="type" longname="Ethertype - Length"
      size="2" showtemplate="eth.typelength"/>
    </fields>
  </format>

  <encapsulation>
    <switch expr="buf2int(type)">
      <case value="0" maxvalue="1500"> <nextproto proto="#llc"/> </case>
      <case value="0x800"> <nextproto proto="#ip"/> </case>
      <case value="0x806"> <nextproto proto="#arp"/> </case>
      <case value="0x8863"> <nextproto proto="#pppoed"/> </case>
      <case value="0x8864"> <nextproto proto="#pppoe"/> </case>
```

```
      <case value="0x86DD"> <nextproto proto="#ipv6"/> </case>
    </switch>
  </encapsulation>

  <visualization>
    <showsumtemplate name="vlan">
      <text value=" (VLAN-ID "/>
        <protofield name="vlanid" showdata="showvalue"/>
      <text value=")"/>
    </showsumtemplate>
  </visualization>

</protocol> <!-- End VLAN protocol definition -->

<!- snip - code removed ... -->

<!--
NetPDL description of the IPv6 Protocol
-->
<protocol name="ipv6" longname="IPv6 (Internet Protocol version 6)
  showsumtemplate="ipv6">
  <!-- We should check that 'version' is equal to '6' -->
  <execute-code>
    <after>
      <!-- Store ipsrc and ipdst in a couple of variables for the sake of speed -->
      <!-- Hids differences between IPv4 and IPv6 for session tracking -->
      <assign-variable name="$ipsrc" value="src"/>
      <assign-variable name="$ipdst" value="dst"/>
      <if expr="$ipsrc lt $ipdst" >
        <if-true>
          <assign-variable name="$firstip" value="src"/>
          <assign-variable name="$secondip" value="dst"/>
        </if-true>
        <if-false>
          <assign-variable name="$firstip" value="dst"/>
          <assign-variable name="$secondip" value="src"/>
        </if-false>
      </if>
    </after>
  </execute-code>

  <format>
    <fields>
      <field type="bit" name="ver" longname="Version"
         mask="0xF0000000" size="4" showtemplate="FieldDec"/>
      <field type="bit" name="tos" longname="Type of service"
         mask="0x0F000000" size="4" showtemplate="FieldHex"/>
      <field type="bit" name="flabel" longname="Flow label"
         mask="0x00FFFFFF" size="4" showtemplate="FieldHex"/>
      <field type="fixed" name="plen" longname="Payload Length"
         size="2" showtemplate="FieldDec"/>
      <field type="fixed" name="nexthdr" longname="Next Header"
         size="1" showtemplate="ipv6.nexthdr"/>
      <field type="fixed" name="hop" longname="Hop limit"
         size="1" showtemplate="FieldDec"/>
      <field type="fixed" name="src" longname="Source address"
         size="16" showtemplate="ip6addr"/>
      <field type="fixed" name="dst" longname="Destination address"
         size="16" showtemplate="ip6addr"/>
```

```
    <loop type="while" expr="1">
      <!-- Loop until we find a 'break' -->
      <switch expr="buf2int(nexthdr)">
        <case value="0">
          <includeblk name="HBH"/>
        </case>
        <case value="43">
          <includeblk name="RH"/>
        </case>
        <case value="44">
          <includeblk name="FH"/>
        </case>
        <case value="51">
          <includeblk name="AH"/>
        </case>
        <case value="60">
          <includeblk name="DOH"/>
        </case>
        <default>
          <loopctrl type="break"/>
        </default>
      </switch>
    </loop>
  </fields>

  <block name="HBH" longname="Hop By Hop Option">
    <field type="fixed" name="nexthdr" longname="Next Header"
       size="1" showtemplate="ipv6.nexthdr"/>
    <field type="fixed" name="helen"
       longname="Length (multiple of 8 bytes, not including first 8)"
       size="1" showtemplate="ipv6.hbhlen"/>
    <loop type="size" expr="(buf2int(helen) * 8) + 6">
      <!-- '6' because the first two bytes are nexthdr and helen -->
      <includeblk name="Option"/>
    </loop>
  </block>

  <block name="FH" longname="Fragment Header">
    <field type="fixed" name="nexthdr" longname="Next Header"
      size="1" showtemplate="ipv6.nexthdr"/>
    <field type="fixed" name="reserved"
      longname="Reserved (multiple of 8 bytes)"
      comment="This is in multiple of 8 bytes"
      size="1" showtemplate="FieldDec"/>
    <field type="bit" name="fragment offset" longname="Fragment Offset"
      mask="0xFFF0" size="2" showtemplate="FieldDec"/>
    <field type="bit" name="res" longname="Res"
      mask="0x0004" size="2" showtemplate="FieldHex"/>
    <field type="bit" name="m" longname="M"
      mask="0x0001" size="2"  showtemplate="FieldBin"/>
    <field type="fixed" name="identification"
      longname="Identification" size="4" showtemplate="FieldDec"/>
  </block>

  <block name="AH" longname="Authentication Header">
    <field type="fixed" name="nexthdr" longname="Next Header"
      size="1" showtemplate="ipv6.nexthdr"/>
    <field type="fixed" name="payload len" longname="Payload Len"
      size="1" showtemplate="FieldDec"/>
    <field type="fixed" name="reserved" longname="Reserved"
```

```
      size="2" showtemplate="FieldDec"/>
    <field type="fixed" name="spi" longname="Security Parameters Index"
      size="4" showtemplate="FieldDec"/>
    <field type="fixed" name="snf" longname="Sequence Number Field"
      size="4" showtemplate="FieldDec"/>
  </block>

  <block name="DOH" longname="Destination Option Header">
    <field type="fixed" name="nexthdr" longname="Next Header"
      size="1" showtemplate="ipv6.nexthdr"/>
    <field type="fixed" name="helen"
      longname="Length (multiple of 8 bytes, not including first 8)"
      size="1" showtemplate="ipv6.hbhlen"/>
    <loop type="size" expr="(buf2int(helen) * 8)+6">
      <!-- '6' because the first two bytes are nexthdr and helen -->
      <includeblk name="Option"/>
    </loop>
  </block>

  <block name="RH" longname="Routing Header">
    <field type="fixed" name="nexthdr" longname="Next Header"
      size="1" showtemplate="ipv6.nexthdr"/>
    <field type="fixed" name="hlen"
      longname="Length (multiple of 8 bytes)"
      comment="This is in multiple of 8 bytes"
      size="1" showtemplate="FieldDec"/>
    <field type="fixed" name="rtype" longname="Routing Type"
      size="1" showtemplate="FieldDec"/>
    <field type="fixed" name="segment left" longname="Segment Left"
      size="1" showtemplate="FieldDec"/>
    <field type="variable" name="tsd" longname="Type Specific Data"
      expr="buf2int(hlen)" showtemplate="Field4BytesHex"/>
  </block>

  <block name="Option" longname="Option">
    <field type="fixed" name="opttype" longname="Option Type"
      size="1" showtemplate="ipv6.opttype">
    <field type="bit" name="act"
      longname="Action (action if Option Type is unrecognized)" mask="0xC0"
      size="1" showtemplate="ipv6.optact"/>
    <field type="bit" name="chg"
     longname="Change(whether or not option data can change while packet en-route)"
     mask="0x20" size="1" showtemplate="ipv6.optchg"/>
    <field type="bit" name="res" longname="Option Code" mask="0x1F"
      size="1"  showtemplate="FieldDec"/>
    </field>

    <switch expr="buf2int(opttype)">
      <case value="0">
                    <!-- No fields are present if the option is not 'Pad1'-->
      </case>
      <case value="5"><!-- Router Alert -->
        <field type="fixed" name="optlen" longname="Option Length"
          size="1" showtemplate="FieldDec"/>
        <field type="fixed" name="value" size="2" longname="Option Value"
          showtemplate="ipv6.optroutalert"/>
      </case>
      <default>
        <field type="fixed" name="optlen" longname="Option Length"
          size="1" showtemplate="FieldDec"/>
```

```
        <field type="variable" name="optval" longname="Option Value"
          expr="buf2int(optlen)" showtemplate="Field4BytesHex"/>
      </default>
    </switch>
  </block>
</format>

<encapsulation>
  <switch expr="buf2int(nexthdr)">
    <case value="4"> <nextproto proto="#ip"/> </case>
    <case value="6"> <nextproto proto="#tcp"/> </case>
    <case value="17"> <nextproto proto="#udp"/> </case>
    <!--   <case value="29"> <nextproto proto="#TP4"/> </case> -->
    <!--   <case value="45"> <nextproto proto="#IDRP"/> </case> -->
    <case value="50"> <nextproto proto="#ipsec_esp"/> </case>
    <case value="51"> <nextproto proto="#ipsec_ah"/> </case>
    <case value="58"> <nextproto proto="#icmp6"/> </case>
    <case value="89"> <nextproto proto="#ospf6"/> </case>
    <case value="103"> <nextproto proto="#pim6"/> </case>
  </switch>
</encapsulation>

<visualization>
  <showtemplate name="ipv6.nexthdr" showtype="dec">
    <showmap>
      <switch expr="buf2int(this)">
        <case value="0" how="Hop By Hop Option Header"/>
        <case value="43" show="Fragment Header"/>
        <case value="44" show="Authentication Header"/>
        <case value="51" show="Destination Option Header"/>
        <case value="60" show="Routing Header"/>
        <case value="50" show="Encapsulating Security Payload"/>
        <case value="58" show="Internet Control Message Protocol (ICMPv6)"/>
        <case value="59" show="No next Header"/>
        <default show="Upper Layer Header"/>
      </switch>
    </showmap>
  </showtemplate>

  <showtemplate name="ipv6.opttype" showtype="hex">
    <showmap>
      <switch expr="buf2int(this)">
        <case value="0" show="Pad1 Option"/>
        <case value="1" show="PadN Option"/>
        <case value="5" show="Router Alert Option"/>
        <default show="Error in IPv6 Option Type lookup"/>
      </switch>
    </showmap>
  </showtemplate>

  <showtemplate name="ipv6.optact" showtype="bin">
    <showmap>
      <switch expr="buf2int(this)">
        <case value="0" show="Skip over option"/>
        <case value="1" show="Discard packet silently"/>
        <case value="2" show="Discard packet-send ICMP"/>
        <case value="3" show="Discard packet-send ICMP if packet was unicast"/>
        <default show="Error in IPv6 Option Action lookup"/>
      </switch>
    </showmap>
```

```
    </showtemplate>

    <showtemplate name="ipv6.optchg" showtype="bin">
      <showmap>
        <switch expr="buf2int(this)">
          <case value="0" show="Option data does not change en-route"/>
          <case value="1" show="Option data may change en-route"/>
          <default show="Error in IPv6 Option Change lookup"/>
        </switch>
      </showmap>
    </showtemplate>

    <showtemplate name="ipv6.optroutalert" showtype="dec">
      <showmap>
        <switch expr="buf2int(this)">
          <case value="0" show="Datagram contains Multicast Listener Disc msg"/>
          <case value="1" show="Datagram contains RSVP message"/>
          <case value="2" show="Datagram contains an Active Networks msg"/>
          <default show="Error in IPv6 Router Alert Option lookup"/>
        </switch>
      </showmap>
    </showtemplate>

    <!-- Length of the hop by hop option header -->
    <showtemplate name="ipv6.hbhlen" showtype="dec">
      <showdtl>
        <text expr="(buf2int(this) * 8) + 8"/>
        <text value=" (field value = "/>
          <protofield showdata="showvalue"/>
        <text value=")"/>
      </showdtl>
    </showtemplate>

    <showsumtemplate name="ipv6">
      <if expr="($prevproto == #ip)  or ($prevproto == #ipv6)  or
                ($prevproto == #ppp) or ($prevproto == #pppoe) or
                ($prevproto == #gre)">
        <if-true>
          <text value=" - "/>
        </if-true>
        <if-false>
          <section name="next"/>
        </if-false>
      </if>

      <text value="IPv6: "/>
      <protofield name="src" showdata="showvalue"/>
      <text value=" => "/>
      <protofield name="dst" showdata="showvalue"/>
      <text value=" (Len " expr="buf2int(plen) + 40"/>
      <text value=")"/>
    </showsumtemplate>
  </visualization>
</protocol> <!-- End IPv6 definition -->

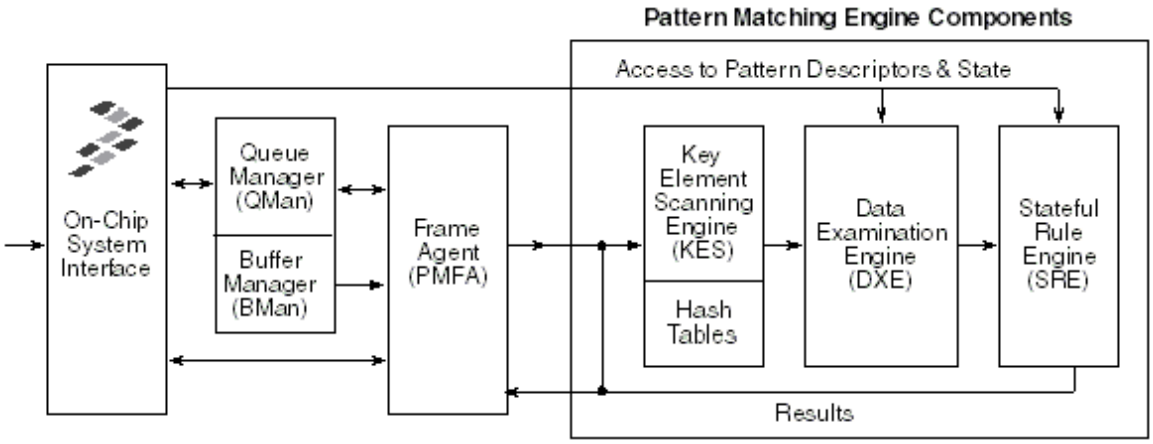
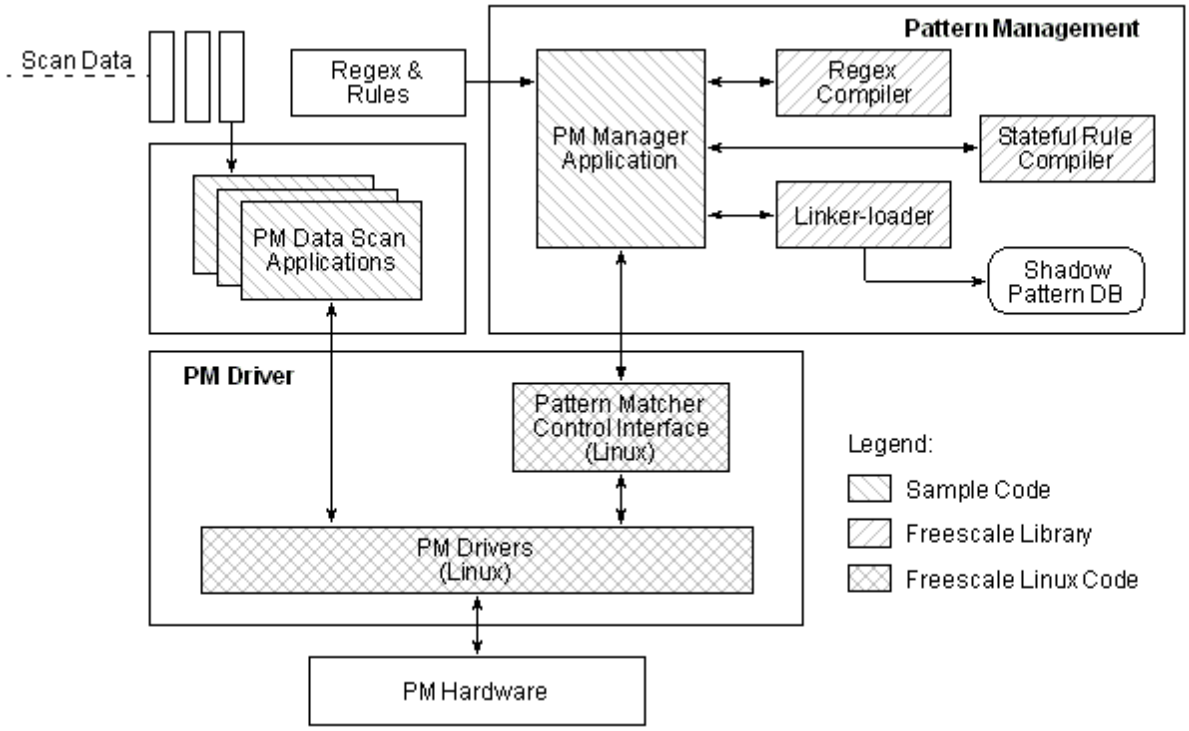
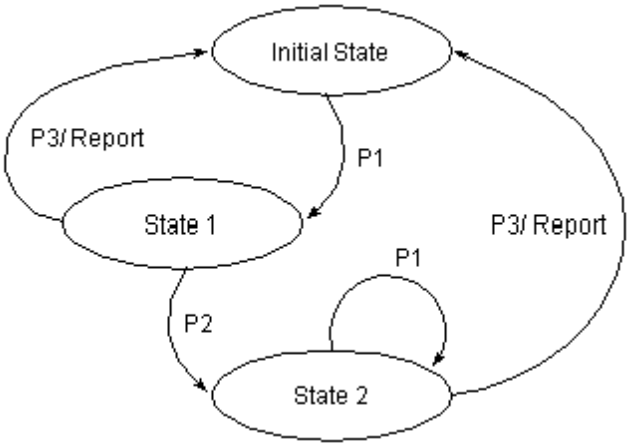
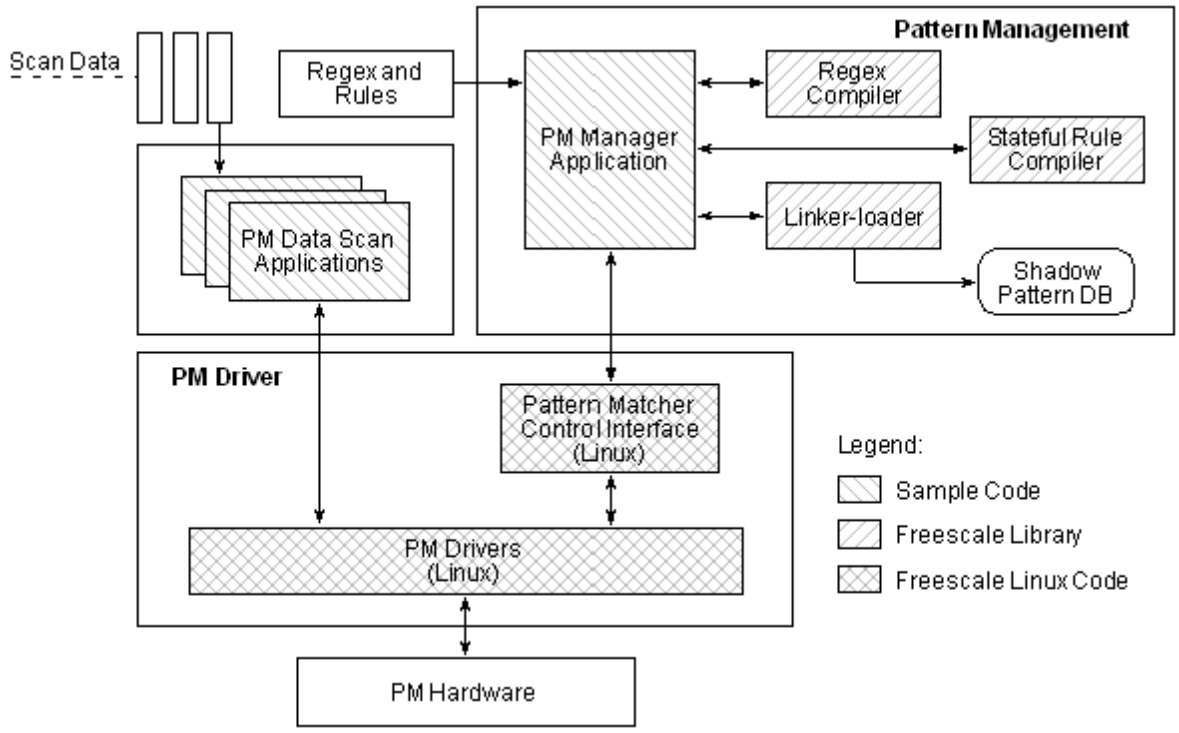
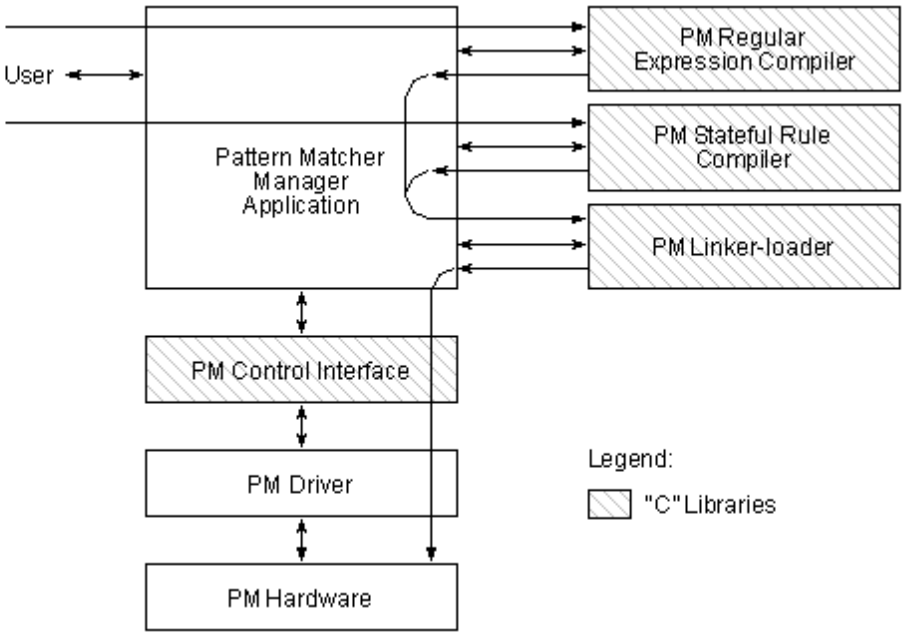
<!- snip - code removed ... -->

</netpdl>
```

```
<!-- End of Standard Protocol file -->
```

7.4.4.5.1.16 Custom protocol file - examples

7.4.4.5.1.16.1 Custom Protocol File - GTP Protocol

The following "GTP_example.xml" file describes the custom GTP protocol.

```
<?xml version="1.0" encoding="utf-8"?>
<netpdl name="GTP" description="GTP-U Example">
  <!-- Gtpu program is an extension to the udp hard shell -->
  <protocol name="gtpu" longname="GTP-U" prevproto="udp">
    <!-- fields in GTP header used for validation and calculating length -->
      <format>
        <fields>
          <field type="bit"    name="flags"    mask="0xE0" size="1" />
          <field type="bit"    name="pt"       mask="0x80" size="1" />
          <field type="bit"    name="version"  mask="0x07" size="1" />
          <field type="fixed"  name="mtype"    size="1" longname="message type"/>
          <field type="fixed"  name="length"   size="2" />
          <field type="fixed"  name="teid"     size="4" />
          <field type="fixed"  name="snum "    size="2" longname="sequence number"/>
          <field type="fixed"  name="npdunum"  size="1" longname="N-PDU number"/>
          <field type="fixed"  name="next"     size="1" longname="Next ext header type"/>
      </fields>
    </format>

    <execute-code>
      <!-- Check that UDP port is 2152 -->
      <before confirm="yes">
        <if expr="udp.dport == 2152">
          <if-true>
          </if-true>
          <if-false>
          <!-- Confirms UDP layer and exits-->
            <action type="exit" confirm="yes" advance="no" nextproto="return"/>
          </if-false>
        </if>
      </before>

      <!-- Done after UDP layer is confirmed-->
      <!--Check version and calculate length-->
      <after confirm="no">
        <if expr="version == 1">
          <if-true>
            <assign-variable name="$shimoffset_1" value="$NxtHdrOffset"/>
          </if-true>
          <if-false>
            <assign-variable name="$ShimR" value="0x23"/>
            <action type="exit" confirm="no" confirmcustom="no" nextproto="none"/>
          </if-false>
        </if>

        <if expr="flags != 0">
          <if-true>
            <assign-variable name="$NxtHdrOffset" value="$shimoffset_1+12"/>
          </if-true>
          <if-false>
```

```
            <assign-variable name="$NxtHdrOffset" value="$shimoffset_1+8"/>
          </if-false>
        </if>
        <action type="exit" confirm="no" confirmcustom="shim1" nextproto="none"/>
      </after>
    </execute-code>
  </protocol>
</netpdl>
```

7.4.5 Security Engine (SEC)

7.4.5.1 SEC Device Driver User Manual

Introduction and Terminology

The Linux kernel contains a Scatterlist CryptoAPI driver for the NXP SEC v4.x, v5.x, v6.x security hardware blocks.

It integrates seamlessly with in-kernel crypto users, such as IPSec, such that any IPSec suite that configures IPSec tunnels with the kernel will automatically use the hardware to do the crypto.

SEC v5.x is backward compatible with SEC v4.x hardware, so one can assume that subsequent SEC v4.x references include SEC v5.x hardware, unless explicity mentioned otherwise.

The name of the software driver module for SEC v4.x hardware is 'caam', after its internal block name: Cryptographic Accelerator and Assurance Module.

Table 183.

SEC hardware version	driver name
v4.x, v5.x, v6.x	caam

Module Loading

CAAM NXP Security Engine device drivers support either kernel built-in or module.

Kernel Configuration

The designated driver should be configured in the kernel by default for the target platform. If unsure, check CONFIG_CRYPTO_DEV_FSL_CAAM is set under "Cryptographic API" -> "Hardware crypto devices" in the kernel configuration:

Kernel Configure Tree View Options	Description
Cryptographic API ---> [*] Hardware crypto devices ---> <*> Freescale CAAM-Multicore driver backend (SEC) <*> Freescale CAAM Job Ring driver backend (SEC) (9) Job Ring size [ ] Job Ring interrupt coalescing <*> Register algorithm implementations with the Crypto API <*> Queue Interface as Crypto API backend <*> Public Key Cryptography Support in CAAM driver <*> Register hash algorithm implementations with the Crypto API	Enable CAAM device driver

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description
<*> Register caam device for hwrng API [ ] Enable debig output in CAAM driver	
Network support ---> Network option ---> <*> PF_KEY sockets <*> IP: AH transformation <*> IP: ESP transformation <*> IP: IPComp transformation <*> IP: IPsec transport mode <*> IP: IPsec tunnel mode	IPsec support, of course the TCP/IP networking option should be enabled

CAAM Driver specifics

The CAAM driver module implements and utilizes two interfaces:

- a job ring interface (JRI) for all crypto API service requests
- (on DPAA 1.x platforms) a queue interface (QI) for AEAD algorithms-based crypto API service requests

The Linux driver automatically sets the enable bit for the SEC hardware's Queue Interface (QI), depending on QI feature availability in the hardware. This enables the hardware to also operate as a DPAA component for use by e.g., USDPAA apps. This behaviour does not conflict with normal in-kernel job ring operation, other than the potential performance-observable effects of internal SEC hardware resource contention, and vice-versa.

Note the CAAM driver has sub-configuration settings, most notably hardware job ring size and interrupt coalescing. They can be used to fine-tune performance for a particular application.

The first item enables the basic Controller Driver, and the Job Ring backend. All suboptions are dependent on this.

The second item ("Job Ring Size") allows the user to select the size of the hardware job rings; if requests arrive at the driver enqueue entry point in a bursty nature, the bursts' maximum length can be approximated etc. One can set the greatest burst length to save performance and memory consumption.

The third item ("Job Ring interrupt coalescing") allows the user to select the use of the hardware's interrupt coalescing feature. Note that the driver software already performs IRQ coalescing, and zero-loss benchmarks have in fact produced better results with this option turned off.

If selected, two additional options become effective:

- Job Ring interrupt coalescing count threshold (CRYPTO_DEV_FSL_CAAM_INTC_THLD)

  Selects the value of the descriptor completion threshold, in the range 1-256. A selection of 1 effectively defeats the coalescing feature, and any selection equal or greater than the selected ring size will force timeouts for each interrupt.

- Job Ring interrupt coalescing timer threshold (CRYPTO_DEV_FSL_CAAM_INTC_TIME_THLD)

  Selects the value of the completion timeout threshold in multiples of 64 SEC interface clocks, to which, if no new descriptor completions occur within this window (and at least one completed job is pending), then an interrupt will occur. This is selectable in the range 1-65535.

The options to register to Crypto API, hwrng API respectively, allow the driver to register its algorithm capabilities with the kernel's crypto API. Deselect them only if you do not want crypto API requests to be performed on the SEC; they will be done in software (on the processor core).

Hash algorithms may be individually turned off, since the nature of the application may be such that it prefers software (core) crypto latency due to many small-sized requests.

Random Number Generation (RNG) may be manually turned off in case there is an alternate source of entropy available to the kernel.

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	fsl,sec-vX.Y (preferred) OR fsl,secX.Y

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/crypto/caam/	CAAM NXP Security Engine Driver

Corresponding Device Tree node

```
    crypto@30000 {
            compatible = "fsl,sec-v4.0";
            fsl,sec-era = <2>;
            #address-cells = <1>;
            #size-cells = <1>;
            reg = <0x300000 0x10000>;
            ranges = <0 0x300000 0x10000>;
            interrupt-parent = <&mpic>;
            interrupts = <92 2>;
            clocks = <&clks IMX6QDL_CLK_CAAM_MEM>,
                     <&clks IMX6QDL_CLK_CAAM_ACLK>,
                     <&clks IMX6QDL_CLK_CAAM_IPG>,
                     <&clks IMX6QDL_CLK_EIM_SLOW>;
            clock-names = "mem", "aclk", "ipg", "emi_slow";
    };
```

NOTE

See linux/Documentation/devicetree/bindings/crypto/fsl-sec{4,6}.txt for more info.

How to test the driver

To test the driver, in the kernel configuration menu, under "Cryptographic API" -> "Cryptographic algorithm manager", ensure that run-time self-tests are not disabled, i.e. the "Disable run-time self tests" (CONFIG_CRYPTO_MANAGER_DISABLE_TESTS) entry is not set. This will run standard test vectors against the driver after the driver registers its supported algorithms with the kernel crypto API, usually at boot-time. Then run test on the target system. Below is a snippet extracted from the boot log.

```
[...]
platform caam_qi.0: Linux CAAM Queue I/F driver initialised
caam 1700000.crypto: Entropy delay = 3200
caam 1700000.crypto: Instantiated RNG4 SH0
caam 1700000.crypto: Instantiated RNG4 SH1
caam 1700000.crypto: device ID = 0x0a12060000000000 (Era 8)
caam 1700000.crypto: job rings = 4, qi = 1
alg: No test for authenc(hmac(sha224),ecb(cipher_null)) (authenc-hmac-sha224-ecb-cipher_null-caam)
alg: No test for authenc(hmac(sha256),ecb(cipher_null)) (authenc-hmac-sha256-ecb-cipher_null-caam)
alg: No test for authenc(hmac(sha384),ecb(cipher_null)) (authenc-hmac-sha384-ecb-cipher_null-caam)
```

```
alg: No test for authenc(hmac(sha512),ecb(cipher_null)) (authenc-hmac-sha512-ecb-cipher_null-caam)
alg: No test for authenc(hmac(md5),cbc(aes)) (authenc-hmac-md5-cbc-aes-caam)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam)
alg: No test for authenc(hmac(sha384),cbc(aes)) (authenc-hmac-sha384-cbc-aes-caam)
alg: No test for authenc(hmac(md5),cbc(des3_ede)) (authenc-hmac-md5-cbc-des3_ede-caam)
alg: No test for authenc(hmac(md5),cbc(des)) (authenc-hmac-md5-cbc-des-caam)
alg: No test for authenc(hmac(md5),rfc3686(ctr(aes))) (authenc-hmac-md5-rfc3686-ctr-aes-caam)
alg: No test for authenc(hmac(sha1),rfc3686(ctr(aes))) (authenc-hmac-sha1-rfc3686-ctr-aes-caam)
alg: No test for authenc(hmac(sha224),rfc3686(ctr(aes))) (authenc-hmac-sha224-rfc3686-ctr-aes-
caam)
alg: No test for authenc(hmac(sha256),rfc3686(ctr(aes))) (authenc-hmac-sha256-rfc3686-ctr-aes-
caam)
alg: No test for authenc(hmac(sha384),rfc3686(ctr(aes))) (authenc-hmac-sha384-rfc3686-ctr-aes-
caam)
alg: No test for authenc(hmac(sha512),rfc3686(ctr(aes))) (authenc-hmac-sha512-rfc3686-ctr-aes-
caam)
caam algorithms registered in /proc/crypto
alg: No test for authenc(hmac(md5),cbc(aes)) (authenc-hmac-md5-cbc-aes-caam-qi)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam-qi)
alg: No test for authenc(hmac(sha384),cbc(aes)) (authenc-hmac-sha384-cbc-aes-caam-qi)
alg: No test for authenc(hmac(md5),cbc(des3_ede)) (authenc-hmac-md5-cbc-des3_ede-caam-qi)
alg: No test for authenc(hmac(md5),cbc(des)) (authenc-hmac-md5-cbc-des-caam-qi)
platform caam_qi.0: fsl,sec-v4.4 algorithms registered in /proc/crypto
caam_jr 1710000.jr: registering rng-caam
alg: No test for pkc(rsa) (pkc-rsa-caam)
alg: No test for pkc(dsa) (pkc-dsa-caam)
alg: No test for pkc(dh) (pkc-dh-caam)
caam 1700000.crypto: fsl,sec-v4.4 algorithms registered in /proc/crypto
[...]
```

Verifying driver operation and correctness

Other than noting the performance advantages due to the crypto offload, one can also ensure the hardware is doing the crypto by looking for driver messages in dmesg

The driver emits console messages at initialization time:

```
platform caam_qi.0: fsl,sec-v4.4 algorithms registered in /proc/crypto
caam 1700000.crypto: fsl,sec-v4.4 algorithms registered in /proc/crypto
```

If the messages are not present in the logs, either the driver is not configured in the kernel, or no SEC compatible device tree node is present in the device tree.

Incrementing IRQs in /proc/interrupts

Given a time period when crypto requests are being made, the SEC hardware will fire completion notification interrupts - either on the corresponding Job Ring or on the QMan (Queue Manager) portal IRQ (depending on what CAAM interface is the algorithm using):

```
cat /proc/interrupts | grep jr
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6
CPU7
[...]
 88:          0          0          0        941          0          0
0          0   OpenPIC    88 Level     ffe301000.jr
 89:          0          0          0          0          0          0
0          0   OpenPIC    89 Level     ffe302000.jr
 90:          0          0          0          0          0          0
0          0   OpenPIC    90 Level     ffe303000.jr
 91:          0          0          0          0          0          0
0          0   OpenPIC    91 Level     ffe304000.jr

cat /proc/interrupts | grep QMan
```

```
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6
CPU7
[...]
108:          0          0          0          0          0          0
0          0   OpenPIC   108 Level     QMan portal 7
110:          0          0          0          0          0          0
0          0   OpenPIC   110 Level     QMan portal 6
112:          0          0          0          0          0          0
0          0   OpenPIC   112 Level     QMan portal 5
114:          0          0          0          0          0          0
0          0   OpenPIC   114 Level     QMan portal 4
116:          0          0          0        285          0          0
0          0   OpenPIC   116 Level     QMan portal 3
118:          0          0          0          0          0          0
0          0   OpenPIC   118 Level     QMan portal 2
120:          0          0          0          0          0          0
0          0   OpenPIC   120 Level     QMan portal 1
122:          0          0          0          0          0          0
0          0   OpenPIC   122 Level     QMan portal 0
```

If the number of interrupts fired increment, then the hardware is being used to do the crypto.

If the numbers do not increment, then first check the algorithm being exercised is supported by the driver. If the algorithm is supported, there is a possibility that the driver is in polling mode (NAPI mechanism) and the hardware statistics in debugfs (inbound / outbound bytes encrypted / protected - see below) should be monitored.

Note: When QI is used, CAAM driver might be sharing the QMan portal with other drivers in the system; meaning that the interrupt counters shown in /proc/interrupts are for all drivers sharing the portal.

Verifying the 'selftest' fields say 'passed' in /proc/crypto

An entry such as this:

```
name         : cbc(aes)
driver       : cbc-aes-caam
module       : kernel
priority     : 3000
refcnt       : 1
selftest     : passed
type         : ablkcipher
async        : yes
blocksize    : 16
min keysize  : 16
max keysize  : 32
ivsize       : 16
geniv        : eseqiv
```

means the driver has successfully registered support for the algorithm with the kernel crypto API.

Note that although a test vector may not exist for a particular algorithm supported by the driver, the kernel will emit messages saying which algorithms weren't tested, and mark them as 'passed' anyway:

```
alg: No test for authenc(hmac(sha224),ecb(cipher_null)) (authenc-hmac-sha224-ecb-cipher_null-caam)
alg: No test for authenc(hmac(sha256),ecb(cipher_null)) (authenc-hmac-sha256-ecb-cipher_null-caam)
[...]
alg: No test for authenc(hmac(md5),cbc(aes)) (authenc-hmac-md5-cbc-aes-caam)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam)
[...]
alg: No test for authenc(hmac(md5),rfc3686(ctr(aes))) (authenc-hmac-md5-rfc3686-ctr-aes-caam)
[...]
```

Examining the hardware statistics registers in debugfs

The controller driver enables a user level view of performance monitor registers located within the controller's register partition. To enable this view, CONFIG_DEBUG_FS must be enabled in the kernel's configuration. If there is no mount of debugfs performed at bootup time, then a manual mount must be performed in order to view these registers. This normally can be done with a superuser shell command of:

```
mount -t debugfs none /sys/kernel/debug
```

Once done, the user can read controller registers in /sys/kernel/debug/caam/ctl. It should be noted that debugfs will provide a decimal integer view of most accessible registers provided, with the exception of the KEK/TDSK/TKEK registers; those registers are long binary arrays, and should be filtered through a binary dump utility such as hexdump.

Specifically, the CAAM hardware statistics registers available are:

fault_addr, or FAR (Fault Address Register): - holds the value of the physical address where a read or write error occurred.

fault_detail, or FADR (Fault Address Detail Register): - holds details regarding the bus transaction where the error occurred.

fault_status, or CSTA (CAAM Status Register): - holds status information relevant to the entire CAAM block.

ib_bytes_decrypted: - holds contents of PC_IB_DECRYPT (Peformance Counter Inbound Bytes Decrypted Register)

ib_bytes_validated: - holds contents of PC_IB_VALIDATED (Performance Counter Inbound Bytes Validated Register)

ib_rq_decrypted: - holds contents of PC_IB_DEC_REQ (Performance Counter Inbound Decrypt Requests Register)

kek: - holds contents of JDKEKR (Job Descriptor Key Encryption Key Register)

ob_bytes_encrypted: - holds contents of PC_OB_ENCRYPT (Performance Counter Outbound Bytes Encrypted Register)

ob_bytes_protected: - holds contents of PC_OB_PROTECT (Performance Counter Outbound Bytes Protected Register)

ob_rq_encrypted: - holds contents of PC_OB_ENC_REQ (Performance Counter Outbound Encrypt Requests Register)

rq_dequeued: - holds contents of PC_REQ_DEQ (Performance Counter Requests Dequeued Register)

tdsk: - holds contents of TDKEKR (Trusted Descriptor Key Encryption Key Register)

tkek: - holds contents of TDSKR (Trusted Descriptor Signing Key Register)

See the hardware documentation section "Performance Counter, Fault and Version ID Registers" for more information.

For extended testing process please refer to the Linux IPsec benchmark reproducibility guide.

Kernel configuration to support caam device driver

Algorithms Supported in the linux kernel scatterlist Crypto API

The linux kernel contains various users of the Scatterlist CryptoAPI, including its IPSec implementation, sometimes referred to as the NETKEY stack. The driver, after registering algorithm services with the CryptoAPI, is therefore used to process per-packet symmetric crypto requests and forward them to the SEC hardware.

Since all SEC version hardware processes requests asynchronous to the processor core, the driver registers asynchronous algorithm implementations with the crypto API: ahash, ablkcipher, and aead with CRYPTO_ALG_ASYNC set in .cra_flags.

Different combinations of hardware and driver software version support different sets of algorithms, so searching for the driver name in /proc/crypto on the desired target system will ensure the correct report of what algorithms it supports.

Authenticated Encryption with Associated Data (AEAD) Algorithms

These algorithms are used in applications where the data to be encrypted overlaps, or partially overlaps, the data to be authenticated, as is the case with the IPSec protocol.

These algorithms are implemented in the driver such that the hardware makes a single pass over the input data, and both encryption and authentication data are written out simultaneously.

The AEAD algorithms are mainly for use with IPSec ESP (however there is also support for TLS 1.0 record layer encryption).

At the time of writing, the CAAM driver currently supports offloading the following AEAD algorithms:

- "stitched" AEAD: all combinations of NULL, CBC-AES/DES/3DES-EDE, RFC3686-CTR-AES with MD-5, SHA-1,-224,-256,-384, and -512
- "true" AEAD: GCM-AES, GCM used in IPsec: RFC4543-GCM-AES and RFC4106-GCM-AES
- TLS 1.0 record layer with "stitched" CBC-AES-HMAC-SHA1

Note: Some of the algorithms register twice - one instance for running over the job ring interface and the other for running over the queue interface. The algorithms registered to run over queue interface will be preferred by the Crypto API, since they have a higher priority (CAAM_CRA_PRIORITY is 4000 for QI and 3000 for JRI).

An exhaustive list of algorithms follows:

authenc(hmac(md5),cbc(aes))

authenc(hmac(sha1),cbc(aes))

authenc(hmac(sha224),cbc(aes))

authenc(hmac(sha256),cbc(aes))

authenc(hmac(sha384),cbc(aes))

authenc(hmac(sha512),cbc(aes))

authenc(hmac(md5),cbc(des3_ede))

authenc(hmac(sha1),cbc(des3_ede))

authenc(hmac(sha224),cbc(des3_ede))

authenc(hmac(sha256),cbc(des3_ede))

authenc(hmac(sha384),cbc(des3_ede))

authenc(hmac(sha512),cbc(des3_ede))

authenc(hmac(md5),cbc(des))

authenc(hmac(sha1),cbc(des))

authenc(hmac(sha224),cbc(des))

authenc(hmac(sha256),cbc(des))

authenc(hmac(sha384),cbc(des))

authenc(hmac(sha512),cbc(des))

authenc(hmac(md5),rfc3686(ctr(aes)))

authenc(hmac(sha1),rfc3686(ctr(aes)))

authenc(hmac(sha224),rfc3686(ctr(aes)))

authenc(hmac(sha256),rfc3686(ctr(aes)))

authenc(hmac(sha384),rfc3686(ctr(aes)))

authenc(hmac(sha512),rfc3686(ctr(aes)))

authenc(hmac(md5),ecb(cipher_null))

authenc(hmac(sha1),ecb(cipher_null))

authenc(hmac(sha224),ecb(cipher_null))

authenc(hmac(sha256),ecb(cipher_null))

authenc(hmac(sha384),ecb(cipher_null))

authenc(hmac(sha512),ecb(cipher_null))

gcm(aes)

rfc4543(gcm(aes))

rfc4106(gcm(aes))

tls10(hmac(sha1),cbc(aes))

Cipher Encryption Algorithms

The CAAM driver currently supports offloading the following encryption algorithms:

cbc(aes)

cbc(des3_ede)

cbc(des)

ctr(aes)

rfc3686(ctr(aes))

xts(aes)

Authentication Algorithms

The CAAM driver's ahash support includes HMAC variants:

hmac(md5)

hmac(sha1)

hmac(sha224)

hmac(sha256)

hmac(sha384)

hmac(sha512)

md5

sha1

sha224

sha256

sha384

sha512

Asymmetric (public key) Algorithms

pkc(dh)

pkc(dsa)

pkc(rsa)

Random Number Generation

caam driver supports random number generation services via the kernel's built-in hwrng interface when implemented in hardware. To enable:

1. verify that the hardware random device file, e.g., /dev/hwrng or /dev/hwrandom exists. If it doesn't exist, make it with:

```
mknod /dev/hwrng c 10 183
```

2. verify /dev/hwrng doesn't block indefinitely and produces random data:

```
rngtest -C 1000 < /dev/hwrng
```

3. verify the kernel gets entropy:

```
rngtest -C 1000 < /dev/random
```

If it blocks, a kernel entropy supplier daemon, such as rngd, may need to be run. See linux/Documentation/hw_random.txt for more info.

Using the driver

Once enabled, the driver will forward kernel crypto API requests to the SEC hardware for processing.

Running IPSec

The IPSec stack built-in to the kernel (usually called NETKEY) will automatically use crypto drivers do offload the crypto to the SEC hardware. Documentation regarding how to set up an IPSec tunnel can be found in the respective open source IPSec suite packages, e.g. strongswan.org, openswan, setkey, etc.

Running OpenSSL

TODO: cross-reference to OpenSSL chapter/section and update/remove text below.

While not officially supported in the SDK, there are userspace interface implementations that enable offloading OpenSSL requests to the built-in kernel crypto API, and thus the SEC hardware via its respective driver. While the kernel officially supports the AF_ALG socket interface, various third-party cryptodev implementations are also available.

Here are some links to a couple of starting points:

```
http://carnivore.it/2011/04/23/openssl_-_af_alg
http://home.gna.org/cryptodev-linux/
http://ocf-linux.sourceforge.net/
```

Executing Custom Descriptors

caam has public descriptor submission interfaces, drivers/crypto/caam/jr.c:caam_jr_enqueue() and drivers/crypto/caam/qi.c:caam_qi_enqueue().

caam_jr_enqueue()

Name

caam_jr_enqueue — Enqueue a job descriptor head. Returns 0 if OK, -EBUSY if the ring is full, -EIO if it cannot map the caller's descriptor.

Synopsis

```
int caam_jr_enqueue (struct device *dev, u32 *desc,
 void (*cbk) (struct device *dev, u32 *desc, u32 status, void *areq),
 void *areq);
```

Arguments

dev: contains the job ring device that is to process this request.

desc: descriptor that initiated the request, same as “desc” being argued to caam_jr_enqueue.

cbk: pointer to a callback function to be invoked upon completion of this request. This has the form: callback(struct device *dev, u32 *desc, u32 stat, void *arg)

areq: optional pointer to a user argument for use at callback time.

caam_qi_enqueue()

Name

caam_qi_enqueue — Enqueue a frame descriptor (FD) into a QMan frame queue. Returns 0 if OK, -EIO if it cannot map the caller's S/G array, -EBUSY if QMan driver fails to enqueue the FD for some reason.

Synopsis

```
int caam_qi_enqueue(struct device *qidev, struct caam_drv_req *req);
```

Arguments

qidev: contains the queue interface device that is to process this request.

req: pointer to the request structure the driver application should fill while submitting a job to driver, containing a callback function and its parameter, Queue Manager S/Gs for input and output, a per-context structure containing the CAAM shared descriptor, the etc.

Please refer to the source code for example usage.

Supporting Documentation

For more information see the *Linux IPSec Benchmark Reproducibility Guide* located in the following SDK directory: sdk_documentation/pdf/Linux_IPSec_benchmark_reproducibility_guide.pdf

7.4.6 Pattern Matching Engine (PME)

7.4.6.1 PME Driver Release Notes

Description

This document describes PME software for the PME hardware block that is part of the QorIQ data path. The PME software runs on version 2.0, 2.1 and 2.2 of the PME hardware. The PME software includes the following components:

- Linux and USDPAA drivers
- Regular Expression compiler for Linux
- Stateful Rule compiler for Linux
- Pattern Matcher Manager (Linux)
- Pattern Matcher Configuration API (Linux)
- Sample Application – Scan Demo (Linux)
- Regular Expression Analyzer Tool (Linux Host only)

Linux and USDPAA Drivers

The PME driver software includes a Linux kernel driver and a PME driver library for USDPAA. The Linux driver provides an ioctl-based Linux user-space interface. The drivers' provide interfaces in Linux kernel and user-space for PME configuration, database setup, and data scanning. For USDPAA, the drivers provide a PME data scanning interface.

The drivers target the Linux and USDPAA environments. The majority of the code is shared between all environments.

The driver includes the following functionality.

PME Configuration interface

The PME configuration interface is an encapsulation of the PME CCSR register space and the global/error interrupt source. This is expected to be managed only by (and visible to) a control-plane operating system,

PME User-space Interface

The user-space interface provides pme_scan and pme_db devices for sending data for scanning and PME database setup respectively to the PME. There is also a `sysfs` interface to control PME global configuration parameters and to access PME statistics.

PME Low-level Driver Interface

The low-level API is a hardware abstraction layer (HAL) where the users have complete control over the data structures and interfaces used to communicate with the PME.

PME High-level Driver Interface

The PME high-level APIs provide a call-back based interface to the PME. The driver provides APIs to manage flow context and issue PMTCC and scan commands. The high-level driver internally co-ordinates commands to the PME and corressponding results from the PME.

PME Pattern Management Software

The Pattern Management software is used to configure the Pattern Matcher by enabling the addition, deletion and querying of patterns and stateful rules. The Pattern Management software components are summarized below.

Regex Compiler

Regular expressions (regexes) provide a powerful way of representing complex patterns. The regex compiler accepts a regex string with options or a file containing regexes, and converts these regexes into patterns stored in Pattern Matcher hardware-specific format, referred to as “test lines”. The regex compiler is provided as source code and as an executable program with command line interfaces. It is also available as a “C” library. The regex compiler is provided for both the x86 and PowerPC targets.

Stateful Rule Compiler

The stateful rule compiler accepts a file containing one or more stateful rules and converts the rules into Pattern Matcher hardware-specific format, referred to as “stateful rule reactions.” If stateful rule reactions are loaded for a pattern, these reactions are executed when that pattern matches. The regex compiler is provided as source code and as an executable program with command line interfaces. It is also available as a “C” library. The regex compiler is provided for both the x86 and PowerPC targets.

Linker-Loader

The linker-loader accepts patterns encoded in the hardware-specific format (test lines) and stateful rules converted into the hardware-specific format (stateful rule reactions) and generates a hardware-optimal pattern and stateful rule database (referred as the Pattern Matcher database). The linker-loader is available as source code and as a C library.

Pattern Matcher Manager

The Pattern Matcher manager (PMM) is an application that manages the creation and distribution of patterns. It uses NXP-provided libraries, namely the regex compiler, the stateful rule compiler and the linker-loader in order to compile, link and load patterns into the Pattern Matcher. PMM maintains a shadow database that allows it to dynamically add or delete discrete patterns from the PME database while the PME remains in-service. The PMM application is implemented as a Linux user-space process that runs on the QorIQ processor containing the Pattern Matcher.

Pattern Matcher Configuration API

The PMC provides a programming interface to Pattern Matcher Management applications for adding, removing, querying and committing regular expressions and stateful rules into Pattern Matcher hardware.

Regular Expression Analyzer Tool (Linux Host only)

The regex_analyzer tool is designed to assist users in getting the best performance out of the Pattern Matcher Engine (PME). It does this by analyzing regular expressions meant for PME hardware use and flagging the ones that may cause performance issues. This tool is available only on the Linux Host.

Software configuration, build information

The PME software is built when supporting platform is selected. The build is via the standard Yocto build process. The user-space tools and applications are available in the `bin_powerpc` directory after a build. The Linux driver is statically linked into the kernel or can be a compiled as a kernel module. A portion of the driver is always built into the kernel - this part allocates a contiguous chunk of memory at boot time that is needed by the PME.

PME Configuration Options

PME Kernel Configure Options	Description
CONFIG_FSL_PME2	Required to build the PME driver
CONFIG_FSL_PME2_CTRL	Compiles device support for the NXP PME2 pattern matching part contained in datapath-enabled SoCs (i.e. accessed via Qman and Bman portal functionality). At least one guest operating system must have this driver support, together with the appropriate device-tree entry, for PME2 functionality to be available. It is responsible for allocating system memory to the device and configuring it for operation. For this reason, it must be built into the kernel and will initialise during early kernel boot.
CONFIG_FSL_PME2_PDSRSIZE	Select the default size of the Pattern Description and Stateful Rule table as a number of 128 byte entries. This only takes effect if the device tree node doesn't have the 'fsl,pme-pdsr' property.
CONFIG_FSL_PME2_SRESIZE	Select the default size of the SRE Context Table as the number of 32 byte entries. This only takes effect if the device tree node doesn't have the 'fsl,pme-sre' property.
CONFIG_FSL_SRE_AIM	Select the alternate inconclusive match mode treatment.
CONFIG_PME2_SRE_ESR	Select if an End of SUI will produce a Simple End of SUI report.
CONFIG_FSL_PME2_SRE_CTX_SIZE_PER_SESSION	Select the default SRE Context Size per Session
CONFIG_FSL_PME2_SRE_CNR	Configured the number of stateful rules as a multiple of 256
CONFIG_FSL_SRE_MAX_INSTRUCTION_LIMIT	Select the maximum number of SRE instructions to be executed per reaction.
CONFIG_FSL_PME2_SRE_MAX_BLOCK_NUMBER	Select the maximum number of reaction head blocks to be traversed per pattern match event (e.g. a matched pattern or an End of SUI event).
CONFIG_FSL_PME2_PORTAL	This compiles I/O support for the NXP PME2 pattern matching part contained in datapath-enabled SoCs (i.e. accessed via Qman and Bman portal functionality).
CONFIG_FSL_PME2_HIGH	This compiles the high-level driver for PME2.
CONFIG_FSL_PME2_TEST_HIGH	This uses the high-level Qman driver (and the cpu-affine portals it manages) to perform high-level PME2 API testing with it.
CONFIG_FSL_PME2_TEST_SCAN	This uses the high-level Qman driver (and the cpu-affine portals it manages) to perform PME2 scan API testing with it.
CONFIG_FSL_PME2_TEST_SCAN_WITH_BPID	This performs the PME scan API test using buffers from the specified buffer pool.
CONFIG_FSL_PME_TEST_SCAN_WITH_BPID_SIZE	This performs the PME scan API test using a buffer pool with the specified size buffers.

Table continues on the next page...

Table continued from the previous page...

PME Kernel Configure Options	Description
CONFIG_FSL_PME2_DB	This compiles the database driver software for PME2. This provides APIs to update the PME2 database.
CONFIG_FSL_PME2_DB_QOSOUT_PRIORITY	The PME DB has a scheduled output frame queue. The QoS priority level for the FQ is configurable via this option.
CONFIG_FSL_PME2_SCAN	This compiles the scan driver software for PME2. This provides APIs for sending scan data to the PME and receiving scan results.
CONFIG_FSL_PME2_SCAN_DEBUG	Trace the PME2 scan driver software with more verbosity using this option.
PME2_STAT_ACCUMULATOR_UPDATE_INTERVAL	The PME statistics accumulator periodically reads current device statistics and adds them to running counters. The frequency of these updates can be controlled by this option.
CONFIG_FSL_PME_BUG_4K_SCAN_REV_2_1_4	Workaround for errata in PME version 2.1.4. Prevents scans of SUIs greater than 4095 - 127 bytes when this revision of HW is detected.

Source Files and Tools Binaries

PME software exists in Linux user and kernel space. All PME software, including tools (like the regex compiler) and Linux driver, are provided in source form. The list below shows the location of all PME files in the release.

Linux User-space

Source Files	Description
pme_tools/include/*.h	The PME Tools header files. example.h is a sample file to demonstrate the build flow for PME tools. See pme_tools/example for details.
pme_tools/applications/*.c	Source code for PME tools like pmm, regex and stateful rule compilers.
pme_tools/pmConfiguration/*	Pattern Matcher Configuration (PMC) api source code
pme_tools/controlInterface/*	The control interface source and compiled files.
pme_tools/loaderAgent/*	The loader agent source and compiled files.
pme_tools/bin_powerpc/*	Compiled PME tools, Scan demo and Test executables.
pme_tools/lib_powerpc/*	Compiled PME tools libraries.
pme_tools/ltib_supp/*	Files that are installed in the ramdisk by Yocto
pme_tools/common/*	Common software utilities used by PME Tools.
pme_tools/example/*	Sample use of PME Tools build and include file hierarchy.

Linux Kernel-space

Source Files	Description
drivers/staging/fsl_pme2/pme2_*.*	The PME2 drivers including user-space PME2 database and scan drivers, and kernel-space high and low-level drivers (except pme2_sample*, pme2_test* which are PME kernel test files).
include/linux/fsl_pme.h	The PME driver APIs

USDPAA

Source Files	Description
include/usdpaa/fsl_pme.h	The PME driver APIs
drivers/pme/*	The PME driver
lib_powerpc	USDPAA static libraries

Test Procedure

The PME software includes tests that are run from the Linux User-space as well as tests that run in the Linux kernel. The kernel tests can be configured via Kconfig as noted in the configuration options described above. These tests complement the PME unit tests.

PME Kernel Tests

The output of the PME tests is shown in the following excerpts. The tests are shows below as being statically linked into the Linux kernel. However, these tests could also have been built and run as loadable kernel modules.

PME High Level Test output:

```
PME2: high-level test starting
PME2: pme_ctx_init done
PME2: pme_ctx_enable done
PME2: pme_ctx_ctrl_update_flow done
pme2_test_high: ctrl_cb() invoked, fd;!
f10791d0: 0100 0000 2f98 93a0 0000 0020 0004 0000
PME2: pme_ctx_ctrl_read_flow done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079210: 0100 0000 2f98 93a0 0000 0020 0005 0000
Default Flow Context Read OK
PME2: pme_ctx_ctrl_nop done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079250: 0100 0000 ea07 fdfc 0000 0000 0007 0000
PME2: pme_ctx_ctrl_update_flow done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079290: 0100 0000 2f98 93a0 0000 0020 0004 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f10792d0: 0100 0000 2f98 93a0 0000 0020 0004 0000
PME2: pme_ctx_ctrl_read_flow done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079310: 0100 0000 2f98 93a0 0000 0020 0005 0000
PME2: pme_ctx_ctrl_nop done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079350: 0100 0000 ea07 fdfc 0000 0000 0007 0000
pme2_test_high: ctrl_cb() invoked, fd;!
```

```
f1079390: 0100 0000
```

PME Scan Test Output:

```
st: About to allocate bpool
st: Allocate buffer pool id 39
st: Allocate buffer of size 256
st: virt address ef980980
st: physical address 0x2f980980
st: Allocate buffer of size 0x100
st: Released to bman
st: Config bpid 39 with size 3
pme2_test_high: ctrl_cb() invoked, fd;!
f1079290: 0100 0000 2f98 93a0 0000 0020 0004 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f10792d0: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079350: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f10793d0: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079010: 0100 0000 2f98 93a0 0000 0020 0004 0000
st: Scan Test Passed
pme2_test_high: ctrl_cb() invoked, fd;!
f1079110: 0100 0000 2f98 93a0 0000 0020 0004 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079150: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f10791d0: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079250: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079290: 0100 0000 2f98 93a0 0000 0020 0004 0000
st: Scan Test Passed
```

Linux user-space Tests

```
Run the pmm application to load the sample regexs:
Setup the pme database:
 $ cd /sample_rules/
 $ pmm
 [ Enter text at the "pmm> " prompt]
 pmm> add regex file source sample_regexs
 REC: WARNING: sample_regexs: Line 22 Compiled pattern has only 1
 byte fingerprint. Performance may be impacted.
 The "sample_regexs" file was compiled with warnings.
 Successfully added 27 regexes to the PM DB with handle 0.
 Command execution time: 00:00:00 [hour:min:sec].
 pmm> commit
 Successfully committed changes made to the data base of expressions.
 Command execution time: 00:00:00 [hour:min:sec].
 pmm> quit
 Terminating the PMM application.
Run the pm_scan_demo application:
 $ pm_scan_demo
```

```
[ Enter text at the "> " prompt]
> example1 /FTF Americas 2006/
#00: Scanning 31 bytes.
match(0x01): len=0x11 offset=0x000000000000:0000001e tag=0x00000000
>quit
Number of successful scan: 1
Number of full match: 1
Number of inconclusive match: 0
Number of rule report: 0
```

Known Bugs, Limitations, or Technical Issues

1. BMan buffer pools for PME output not supported for Linux user space use.

2. In case of any PME error detection, the software PME Context transitions to DEAD state. This prevents use of a PME Context after an operational error.

Supporting Documentation

- P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual
- Pattern Matcher 2.0 Software User's Guide
- Pattern Matcher 2.0, 2.1, 2.2 Software API Reference Manual
- USDPAA “pme_loopback” User Guide

7.4.6.2 Pattern Matcher 2.0 Software User's Guide

7.4.6.2.1 Preface

7.4.6.2.1.1 About This Book

This user's guide is written for programmers developing software for a PowerQUICC processor with Pattern Matcher 2.0 capabilities. It describes the supplied pattern matcher software components, how to use various application programming interfaces, and how to integrate the resulting customized software. The supplied sample Pattern Matcher applications are also described in this document.

7.4.6.2.1.2 Audience

This manual supports system software and application programmers who want to use the Pattern Matcher 2.0 in their product.

7.4.6.2.1.3 Organization

This book contains the following topics.

- Overview - The Pattern Matcher hardware and software architecture.
- Regular Expressions -The regex operation and the compiler.
- Stateful Rules -The stateful rule concepts and the stateful rule language.
- Pattern Management Software -The functionality of the main software modules for pattern management, including the linker-loader, the pattern matcher manager, the PM statistics manager, and the distributed pattern management software .
- Pattern Matcher Driver -The Pattern Matcher driver software for Linux is the layer of software that provides applications an interface into the Pattern Matcher. This chapter discusses the application interfaces, initializing and configuring the drivers, memory structure, scanning data, third-party interfaces, and error handling.

- Data Scan Sample Application -Three main example applications, including a scan demo and a layer-7 packet classifier for Linux.
- Software Components - Software modules and a listing of sample code modules.
- Pattern Matcher FAQ - Frequently asked Pattern Matcher questions.

7.4.6.2.1.4 Conventions

This document uses the following notational conventions:

- `Courier`

  monospaced type indicates commands, command parameters, code examples, and file and directory names.
- *Italic* type indicates replaceable command or function parameters, that is, variables.
- **Bold** type indicates function names.

7.4.6.2.1.5 Suggested Reading

The following documents contain information that supplements this guide:

- Applicable PowerQUICC processor (contains Pattern Matcher 2.0) reference manual
- *Pattern Matcher 2.0 Software API Reference Manual* (PMAPIRM)

7.4.6.2.2 Overview

The Pattern Matcher 2.0 provides the following capabilities:

- High-performance, hardware pattern-matching of compressed and uncompressed data.
- On-chip hash tables for low system memory utilization.
- Patterns expressed in regex with capabilities beyond that provided by the regex language.
- Pattern-matching across data scan units (for example, can match patterns split across packets).
- Improvements over other Pattern Matcher technologies are as follows:
  - No pattern "explosion" to support "wildcarding"
  - Fast compilation of pattern database
  - Fast incremental additions to pattern database
  - Patterns stored in main DDR DRAM, not SRAM or FCRAM

7.4.6.2.2.1 Pattern Matcher Hardware

The Pattern Matcher 2.0 hardware details can be found in the applicable PowerQUICC processor reference manual. A general overview is provided below:

The Pattern Matcher 2.0 (henceforth referred to as Pattern Matcher) contains high-performance hardware capable of performing unanchored searches for up to 32,000 patterns. It offers support for built-in case insensitivity, repetition, and nested rules, and both predefined and user-defined character classes. Additionally, the Pattern Matcher provides the ability to create stateful relationships between data examination events, such as matched expressions detected by the Pattern Matcher so more complex regular expressions, as well as certain other complex multi-pattern rules, can be implemented directly in hardware.

The Pattern Matcher is implemented as a series of functional units pipelined together to achieve desired functionality and scan performance as shown in Figure 137. Pattern Matcher High Level Block Diagram on page 872:

- Pattern Matcher Frame Agent (PMFA)
- Key element scanner (KES)

- Data examination engine (DXE)
- Stateful rule engine (SRE)



Figure 137. Pattern Matcher High Level Block Diagram

Up to eight work-units can be in flight in the pipeline at any given time. A work-unit represents an atomic work request operation to the Pattern Matcher. Work request operations are conveyed to the Pattern Matcher through the Pattern Matcher Frame Agent (PMFA)via Queue Manager (QMan) Direct Connect Portal. The Pattern Matcher implements a single pipeline, which means that processing of work-units are completed in the same order as they were initially selected.

The first stage of the pipeline is the PMFA. It provides an interface to the Frame Queue datapath environment (Queue Manager and Buffer Manager) to receive work requests (scan data and control messages) and to send notifications/reports (if required) of completed work requests.

The core pattern-matching functionality is implemented as a three stage pipeline. The first stage is the key element scanner (KES), which serves as a preliminary filter by detecting the possibility of matches of up to 32,000 patterns simultaneously through a single pass through the data. The KES implements a multi-stage, hash-based, proprietary algorithm using on-chip memory to determine the likelihood of the data being a match with a pattern. Matches found at this stage are delivered to the data examination engine (DXE), which performs a more stringent comparison to determine if the match actually exists or not. The DXE implements a non-deterministic finite automaton (NFA) capable of implementing a significant subset of the regular expression (regex) pattern definition language, as well as many constructs that cannot be expressed in regex. Successfully matched patterns are passed to the next stage called the stateful rule engine (SRE). The SRE's main function is to execute stateful rules against pattern match events. Stateful rules provide the means to track state and context information between matches. The SRE also formats the reports of matches including pattern match events that don't trigger the execution of stateful rules. The reports are passed to the PMFA, which in turn produces an entry in the output frame queue that contains a pointer to an output data descriptor containing the pattern reports associated for a given work-unit.

7.4.6.2.2.2 Pattern Matcher Software

The Pattern Matcher software provides the ability to use the Pattern Matcher hardware assists to recognize various complex patterns in a stream of either data or packets. The application may be informed of every single pattern match or a summary of several pattern matches. Based on the pattern match results, the application can take appropriate actions, such as logging the pattern match events, dropping some packets, or forwarding the data onwards. The Pattern Matcher allows for data scanning with minimum data copy (for example, any data received from the network and stored in the system memory buffers may be sent to the Pattern Matcher without any need for data copy). Also, the Pattern Matcher is capable of detecting patterns spread across successive presentation of bytes in multiple scan units.

The figure below shows the supplied Pattern Matcher software components and their interactions.

- Pattern Management software
  - Regex Compiler

- Stateful rule Compiler
- Linker-loader
- Pattern Matcher (PM) Manager application (Linux implementation)

- Pattern Matcher Driver
  - Pattern Matcher (PM) Driver (Linux implementation)
  - Pattern Matcher (PM) Control Interface (Linux implementation)
- Pattern Matcher (PM) Data Scan Applications

The section, Software Components, contains a list of supplied Pattern Matcher software modules.



Figure 138. Pattern Matcher Software Overview

7.4.6.2.2.2.1 Pattern Management Software for Pattern Matcher

The Pattern Management software is used to configure the Pattern Matcher by enabling adding, deleting and querying patterns (arrangement of symbols to be matched) and stateful rules (means to track state and context information between pattern matches). The Pattern Management software components are explained in the following sections.

NOTE

Both the regex and stateful rule compilers are available as an executable program with command line interface and also as a "C" library with API.

7.4.6.2.2.2.1.1 Regex Compiler

Regular expressions (regexes) provide a powerful way of representing complex patterns. The regex compiler accepts a regex string with options or a file containing regexes, and converts these regexes into patterns stored in Pattern Matcher hardware-specific format, referred to as "test lines".

7.4.6.2.2.2.1.2 Stateful Rule Compiler

The stateful rule compiler accepts a file containing one or more stateful rules and converts the rules into Pattern Matcher hardware-specific format, referred to as "stateful rule reactions." If stateful rule reactions are loaded for a pattern, these reactions are executed when that pattern matches.

7.4.6.2.2.2.1.3 Linker-Loader

The linker-loader accepts patterns encoded in the hardware-specific format (test lines) and stateful rules converted into the hardware-specific format (stateful rule reactions) and generates a hardware-optimal pattern and stateful rule database (referred as the Pattern Matcher database). A copy of this database (referred as the shadow pattern database) is maintained by the linker-loader for reload and update purposes.

There are architectural advantages to having the separate step to link patterns using linker-loader from compiling patterns; these advantages are as follows:

- Different pattern compilers can coexist with single linker-loader. For example, the linker-loader can be used with the regex compiler from NXP and a proprietary pattern compiler for the customer's own signature definitions.
- Compilers can run on a number of different remote host systems.

The linker-loader is available as a "C" library.

7.4.6.2.2.2.1.4 Pattern Matcher Manager Application (PMM)

The Pattern Matcher manager application (PMM) is an example of an application that manages the creation and distribution of patterns. It uses NXP-provided libraries, namely the regex compiler, the stateful rule compiler and the linker-loader in order to compile, link and load patterns into the Pattern Matcher.

The PMM application is implemented as a Linux user-space process that runs on the PowerQUICC processor containing the Pattern Matcher.

Take note that the Pattern Management software can be implemented as a distributed application with only a small part running on directly on the PowerQUICC processor containing the Pattern Matcher.

7.4.6.2.2.2.2 Pattern Matcher Driver

The Pattern Matcher driver provides a means of sending commands (for example, pattern search request or pattern configuration messages) into the Pattern Matcher and delivering the responses from the Pattern Matcher to the software. The pattern search is performed under full application control via the Pattern Matcher Driver. After a unit of data (for example, data work-unit) has been searched, the application is presented with the pattern search report if configured to do so.

7.4.6.2.2.2.2.1 Pattern Matcher Control Interface

The PM Control Interface (PMCI) module is provided as a Linux user-space library. It contains C functional Interface to send and receive Pattern Matcher control commands to the Pattern Matcher via Pattern Matcher driver software. The PM control commands initialize PM driver and configure the PM hardware tables. The PMCI module converts complex PM control commands into PM driver primitives. The PMCI API functions and the PM Control command structures are defined in the Pattern Matcher 2.0 Software API Reference Manual.

7.4.6.2.2.2.3 Pattern Matcher Data Scan Applications

The PM Data Scan Applications are customer applications that use the Pattern Matcher hardware for accelerating pattern searches. These applications may receive data from network or other sources. When it comes time to search for patterns, applications use the PM Driver APIs to send a copy of the data to the Pattern Matcher hardware. The NXP Linux implementation of the PM Driver APIs provides mechanisms for zero-copy DMA operations. The pattern match events are reported back to the application as results of the scan operations. Linux-based Pattern Matcher Drivers provide a variety of scan APIs to the PM data scan applications, which are mainly:

- User-space blocking and non-blocking APIs
- Kernel-space blocking and non-blocking APIs

The expressions and stateful rules are added via the PMM before PM data scan applications can search data for these expressions and rules. Also, the expression and rules may be incrementally added or deleted dynamically without completing stopping the scan applications.

The source code for few data scan applications is provided as an example use of the PM Driver APIs.

7.4.6.2.3 Regular Expressions

The NXP regex compiler accepts search patterns using syntax similar to that in software-based regex engines, such as Perl or the open source Perl-compatible regular expression engine (PCRE). Despite the similar syntax, different software-based regex engines work differently and sometimes produce different match results. Although the NXP Pattern Matcher hardware combined with the NXP regex compiler behaves like other engines in many respects, it has important differences and unique features. The NXP regex engine described in this chapter is the combination of the Pattern Matcher hardware and the regex compiler.

NOTE

For details on the regex compiler, refer to *Pattern Matcher Compiler Software User's Guide.*

7.4.6.2.4 Application interface

The regex compiler is provided as a C library and also as a binary executable program under Linux. The PM Manager system software can compile single regex or multiple regexes stored in a file using either the library function or by invoking the regex compiler program.

7.4.6.2.4.1 Compiler API

The regex compiler API functions are located in following header file:

```
        <freescale pm source code path>/pm/user/include/pmrec.h
```

For details on the regex compiler API, refer to *Pattern Matcher 2.0 Software API Reference Manual*.

7.4.6.2.4.2 Command Line

The pmrec is the executable program to compile regexes from stdin or an input file, as shown in the following command help:

```
pmrec --help
Description:
      The regex compiler takes input from a file or stdin and
      converts the regex to patterns in an internal format.
      The output must be passed to the Linker-loader software in order
      to install the patterns on the hardware.
   Options:
      -h, --help           This help.
      -i, --input          The name of the file containing the user's regular expressions.
                           Defaults to STDIN.
      -o, --output         The name of the file where the output will be placed.
                           Defaults to 'regex.compiled'.
      -W, --Werror         Warnings are errors.
      -w, --w              Suppress warning messages. Note: Ignored if --Werror used.
      -n, --nostrings      Do not include expression strings in output binary file.
      -b, --8572rev1.0     Compile for 8572 rev 1.0 silicon.
Example:
      pmrec --input my_expressions --output my_expressions.out
```

The following example shows the regex input file.

Regex Input File

```
#
# Regular expression file
#
# Look for login attempts
login /login/set=1 subset=0xffff tag=0x01
# Look for logouts
logout /^logout/m tag=0x02
```

7.4.6.2.4.3 Scanning API

When the patterns are compiled, they must be added in the Pattern Matcher hardware using the linker-loader. After patterns are configured into the PM hardware, the PM driver pattern scanning interfaces are used to search for patterns.

7.4.6.2.4.4 Scan Result

The data scan operation generates s scan result notification from the Pattern Matcher. The scan result contains zero or more match reports. The match report contains information, such as the tag of the regex that matched and where in the data the match was found.

NOTE

If the noreport option is specified in the regex, no match report is generated for that regex.

The type of report, simple or verbose, is configurable on a per-scan stream basis. For a reference, the simple match report is described in Table 184. Simple Match Report on page 876. Refer to applicable PowerQUICC processor reference manual for the description of the verbose reports.

Table 184. Simple Match Report

Byte Offset	Bits	Description
0	0	Set to 0
0	1-3	Indication of match type as either full match or inconclusive match. The values varies based on the configuration of the inconclusive mode selection. For the default mode the values are: 000 Complete match within a search window 001 Inconclusive match on the right side of the search window 010 inconclusive match on the left side of the search window 011 inconclusive match on both left and right side of the window
0	4-7	Set to 0x1
1	0-7	Number of bytes matched by the pattern (max value of 128)
2	0-47	The number of bytes scanned initially prior to the current work-unit.
8	0-31	The position of rightmost byte of the match relative to the work-unit scanned
12	0-31	A 32-bit tag assigned to the regex that matched

At the end of the match reports, there is an optional end-of-SUI report. The PM driver configuration parameter is available to enable or disable the end-of-SUI reports per system. It is enabled by default. The end of SUI report is 5 bytes in length as described in Table 185. End of SUI Report on page 877.

Table 185. End of SUI Report

Byte Offset	Bits	Description
0	0-7	Set to 0x80
1	0-31	Total new bytes scanned for this report. This is equal to bytes in the work-unit.

The scan result notification presented by the PM driver also indicates whether or not the result is truncated due to insufficient buffer space. This can occur for two reasons, as follows:

- The software supplied output scatter/gather buffer is not large enough to contain all of the produced output.
- The DMA Engine channel's free buffer list is exhausted while the DMA Engine is buffering output data.

NOTE

The scan application behavior should be considered carefully in handling the scan result that may be truncated. The truncated results imply that not all of the matches are reported in the result. Additionally, there may be a partial match report or a partial end of SUI report in the result due to truncation.

The scan result notification also indicates whether or not there are any exceptions encountered by the Pattern Matcher. The Pattern Matcher exception codes are described in the application PowerQUICC Processor Reference Manual. The "DXE instruction limit error" exception is worthy of a note. This error indicates that the Pattern Matcher has exceeded the maximum allowed test line executions per pattern. Almost no real regexes can exceed such limit, but it is theoretically possible. If such limit is reached, either rewrite or delete the regex causing this error. Alternatively, using PM driver software, it is possible to increase or disable the maximum allowed test line executions limit.

7.4.6.2.5 Stateful rules

The stateful rule engine (SRE), a component of the Pattern Matcher, detects and reports the occurrence of specified events within the data content of individual sessions. A stateful rule defines a state machine that executes actions when specific events occur. The SRE does not directly examine the content but instead reacts to events from the data examination engine (DXE). Typical DXE events are notifications of successfully matched patterns. The DXE supplies additional information on each event, such as match position and the captured data from the scanned data content. The stateful rule can use this information in its reactions to events.

As shown in the example state machine diagram in the Figure 139. Stateful Rule State Machine on page 878, a stateful rule can be used to detect and report occurrences of patterns in a particular sequence. For example, the detection of pattern P3 is reported, but only after an occurrence of pattern P1 or an occurrence of P1 followed by P2. The complex scenarios, such as the example in Figure 139. Stateful Rule State Machine on page 878, can be written using a combination of the regex and stateful rules.



Figure 139. Stateful Rule State Machine

From a hardware perspective, a stateful rule is a set of reactions that is applied to a single meta-state record. Each reaction executes as a result of a specific data examination event, such as the confirmed detection of a pattern. The stateful rule compiler accepts the stateful rule and outputs binary data that represents SRE reactions. The binary output is used in an API call to the linker-loader to link the rule with appropriate patterns and to load the rule to the Pattern Matcher hardware. The expressions used in a stateful rule must be added through the linker-loader before the stateful rule is linked.

NOTE

For details on the stateful rule compiler, refer to *Pattern Matcher Compiler Software User's Guide*.

7.4.6.2.5.1 Stateful Rule Application Interface

The stateful rule compiler is provided as a C library and a binary executable program under Linux. The PM manager software can compile single or multiple rules stored in a file using either the library function or by invoking the compiler program.

7.4.6.2.5.1.1 Stateful Rule Compiler API

The stateful rule compiler header file is:

```
pm/user/include/pmsrc.h
```

For details on the compiler API, refer to the *Pattern Matcher 2.0 Software API Reference Manual*.

7.4.6.2.5.1.2 Command Line Options

The pmsrc is the executable program to compile rules from stdin or input file. See command help, as follows:

```
stateful_rule_compiler
   Description:
      The stateful rule compiler takes input from a file or STDIN and
      converts the user code to low level stateful rule instructions.
      The output must be passed to the linker/loader software in order
      to install the rules on the hardware.
   Options:
   -h, --help    This help.
   -i, --input <file>               The name of the file containing the users
                                    stateful rules.
                                    Defaults to STDIN.
   -o, --output <file>              The name of the file where the output will
                                    be placed.
                                    Defaults to 'stateful_rule.compiled'.
```

```
   -r, --report_pad <size>           Pad reports to this byte boundary.
                                     Default is none.
                                     Allowed values: 0 or 4
   -p, --string_pad <size>           Pad strings to this byte size.
                                     Default is 80.
                                     Allowed values: 0 < value <= 16384 (multiple of 2)
   -c, --report_constant_size <size> Constants within reports will be
                                     aligned to this byte boundary.
                                     Default is 4.
                                     Allowed values: 2, 4, 6, or 8
   -a, --allow_inconclusive          Allow inconclusive matching.
                                     Default is to disallow inconclusive matches.
   -W, --Werror                      Warnings are errors.
   -w, --w                           Suppress warning messages.Ignored if --Werror used.
Example:
      stateful_rule_compiler --input my_rules -output my_rules.out
```

The following example shows the stateful rule input file.

Stateful Rule Input File

STATEFUL_RULE: HTTP_Recognizer

RESET_STATE:

EVENT "http_request"

next_state AWAIT_response

STATE AWAIT_response:

EVENT "http_response"

report HTTP traffic observed

report {0x00000001}

next_state RESET_STATE

7.4.6.2.5.1.3 Scan Report

The scanning interface for stateful rules is no different than the interface for scanning regex. When data is scanned to search for regex, the stateful rules, if configured, are also executed for any matched regex. The stateful rule match report content is customized by the stateful rule reactions. It can be any number of bytes and values, depending on the reaction. Typically, rule writers select a certain stateful rule report header so that the scan applications can interpret the report consistently.

The report action discussed in "Actions" (Chapter 3, "Stateful Rules") found in *Pattern Matcher Compiler Software User's Guide* describes the optional header appended by the stateful rule compiler. Alternatively, stateful rule writers can use a report format that is identical to the Pattern Matcher simple report format.

7.4.6.2.6 Pattern management software

The pattern management software is built using the Pattern Matcher software libraries provided by NXP. The sample Pattern Matcher manager application (PMM), described in Pattern matcher manager (PMM) on page 882, is supplied by NXP as a Linux user-space application. The user can write their own PMM software or add PMM functionality to their existing management application. The high-level PMM application functions are as follows:

- Providing the user interface for adding, deleting, and querying source regexes and stateful rules
- Linking compiled regexes and compiled rules to create a PM hardware database
- Committing the PM hardware database to the PM hardware
- Monitoring PM hardware for any exceptions

The figure below shows the PMM application with NXP-supplied software components.

.



Figure 140. Locally-Managed PM Hardware

7.4.6.2.6.1 Compilers

7.4.6.2.6.2 Linker-loader for pattern management

The linker-loader, a sub-component of the pattern management software system, allows a user of the Pattern Matcher module to configure the hardware. You can use the linker-loader to configure new search patterns in the monitored data or to query the state of the Pattern Matcher hardware. The linker-loader manages the PM hardware tables, which are the trigger, confidence, and confirmation tables. These tables are described in the PowerQUICC processor reference manual for your product. The linker-loader is implemented as a library of functions performing different actions on the Pattern Matcher hardware. These functions are accessible through a well-defined API. In the Linux implementation, the linker-loader library is used by a user space application. Figure 141. Pattern matcher control data path on page 881 illustrates the PM control data path at a high level.



Figure 141. Pattern matcher control data path

The linker-loader library maintains a shadow database of the hardware tables that contains the Pattern Matcher hardware database plus additional information necessary for the software, such as all currently configured expressions and stateful rules. The hardware tables are highly compressed to save PM hardware internal memory usage. The linker-loader compresses the database as part of the commit phase.

The user interacting with the shadow database refers to expressions and rules by their names. The names do not have to be stored in the hardware database, and expressions are added or removed from the shadow database. Whenever the changes made to the shadow database are loaded to the PM hardware, it is reconfigured to reflect the state of the shadow database. The expressions and stateful rules can be incrementally deleted from or added to the Pattern Matcher hardware through the linker-loader library. When incremental changes are committed, the linker-loader does not load the entire shadow database to the hardware. Only incremental changes are loaded, thus limiting any outage while making incremental changes.

NOTE

Because the shadow database is embedded in the linker-loader library, take care to ensure data integrity through the Pattern Matcher manager (PMM) application. The memory used for the shadow database is not persistent. It is allocated in the context of the PMM application.

The API to the linker-loader library are described in the Pattern Matcher 2.0 Software API Reference Manual.

7.4.6.2.6.2.1 Linker-Loader API

The linker-loader is provided as a C library. The pattern management software can include the linker-loader library to link and load patterns and rules into the Pattern Matcher hardware. The linker-loader API is a functional API, and each API function runs to completion. In other words, the requested operation is completed and the results of the operation are available when the invoked function returns. The API functions are divided into the following categories, all of which are described in the *Pattern Matcher 2.0 Software API Reference Manual*:

- Initialization primitives
- Expression related primitives
- Stateful rule related primitives
- Debug primitives

Note that the linker-loader library does not interact directly with the Pattern Matcher control interface. Instead, the library sends Pattern Matcher control messages through functions provided by the PMM application. The PMM registers functions

(Refer to the PM loader agent functions in the Pattern Matcher 2.0 Software API Reference Manual). That is, it sends PM control messages to the Pattern Matcher control interface indirectly using callback functions registered by the PMM. The callback functions provide you the flexibility to implement the communication pipe between the linker-loader and the PM control interface that is most appropriate for the pattern management system. The PM software architecture allows the PMM application to be split into three parts, one that compiles regex and rules, another that interacts with the linker-loader, and yet another that interacts with the Pattern Matcher control interface.

7.4.6.2.6.3 Pattern matcher manager (PMM)

The sample Pattern matcher manager (PMM) is a NXP application implemented as a Linux user-space process. The PMM application provides a simple command-based interface to compile, link, and load expressions and stateful rules. It runs under Linux 2.6 within the same processor system (host) on which the PM hardware resides. This section describes the key commands available through the PMM command-line interface. For more information, refer to the PMM help text.

The PMCC/PMCD application is an alternative application that provide the same features of PMM. PMCD runs as a daemon(memory resident program), and PMCC runs as a client with a command-line interface identical to PMM. To use PMCC/PMCD, run “pmcd” first, then run “pmcc”.

The difference between them is PMM is a standalone application without any persistent datapath that spans multiple invocations of PMM. It starts with an empty expression and rule database, so a new session of PMM cannot add or delete items already committed to PME by a previous PMM session. With PMCC/PMCD, PMCD will keep the database in memory. As long as PMCD is running, the PME shadow database remains persistent. With PMCD running, any new session of PMCC can add or delete items added by previous PMCC sessions.

It is not recommended to run PMM and PMCD, or multiple session of them at the same time. Since each session of PMM or PMCD assume the sole controller role of the PME, the PME database corruption can occur with multiple controllers.

7.4.6.2.6.3.1 Adding Regexes and Rules

The add command is used to add regex or rules to the linker-loader database, as shown in the Adding Regexes and Rules example that follows. The regex(es) or rule(s) to be added can be in the source format (yet to be compiled), or they can be in the binary format (already compiled). If the regex is presented in the source format, it is first compiled. The compilation must finish successfully before the regex is added to the linker-loader database. If the regex or rule is presented in the binary format, it is added to the linker-loader database without preprocessing. Clearly, adding regexes or rules in the binary format is faster.

When source formatted regexes or rules are added from a file, the compiled results can be stored in a binary formatted file. Specify the optional binary keyword followed by a file name.

Adding Regexes and Rules

```
add regex file source myexpressions.src

add regex name e1 exp /matchme/tag=0x01

add regex file binary regexes.bin

add rule file source /tmp/rules.src binary /tmp/rules.bin
```

7.4.6.2.6.3.2 Committing Added Regexes and Rules

The commit command is used to configure the Pattern Matcher hardware with the regexes and rules that are added to the system. The first, or initial, invocation of the commit command results in an optimal distribution of the pattern records in the PM hardware. Such optimization boosts the performance of the PM hardware. The second and subsequent, or incremental, invocations of the commit command do not perform such optimization. Depending on a particular set of regexes in the database, the regexes that are added during incremental commits, and number of incremental commits, this may not pose

any performance issue. However, in general the pattern management system design should use the first commit command with a regex and rule database that is as close to its final state as possible. With many incremental commits, over time the database may be not be as optimum as it can be with an initial commit.

NOTE

As part of the initial commit, the PM hardware database is reset. That is, all previously committed patterns and rules are invalidated.

7.4.6.2.6.3.3 Deleting Regexes and Rules

The delete command is used to delete previously added expressions or rules. One or more expressions or rules can be deleted with one command, as shown in the following example:

```
delete regex all

delete regex name p1
```

The number of expression or rule names accepted by the command is limited by the number of arguments allowed in a CLI command. When a request is made to delete all items of a given kind and the operation completes with an error, only the items that can be deleted are deleted. For example, when all the expressions are deleted, the expressions that are part of rules are not deleted.

7.4.6.2.6.3.4 Showing Regexes and Rules

The show command can be used to display the fields of the records added to the system-for example, the expression or rule records. The command can also be used to display version information for some software components, such as the linker-loader. An example of showing regexes and rules is as follows:

```
show regex all
```

7.4.6.2.6.3.5 Other Commands

There are other useful commands within PMM to set, read, and reset various Pattern Matcher attributes.

7.4.6.2.6.3.6 PM Statistics

The Pattern Matcher hardware offers a variety of statistics counters for debugging and analyzing pattern scanning functions. The statistics counters are read reset; that is, they are reset to zero when software reads them. Also, if software does not read the counters frequently, they risk rollover. The statistics offers are from all three PME engies, KES, DXE, and SRE.

The PM statistics collection is done via PME kernel drivers. The counters are queried from the driver interface. The PMM sample applicaion provides means to read statistics from driver.

The following shows the output of the statistics query command from the sample PMM application.

Output of Statistics Query Command

```
pmm> The PM H/W Statistics:                Current       Previous          Delta

          PM Input Bytes           (KES) :          50000              0          50000

          PM Output Report Bytes   (SRE) :         283155              0         283155

          PM Trigger 1B Hits       (KES) :              0              0              0

          PM Trigger 2B Hits       (KES) :              0              0              0

          PM Trigger Variable Hits (KES) :              0              0              0
```

```
PM Trigger Special Hits  (KES) :          10786              0          10786

PM Confidence Stage Hits (KES) :          10786              0          10786

PM Matches               (DXE) :           5345              0           5345

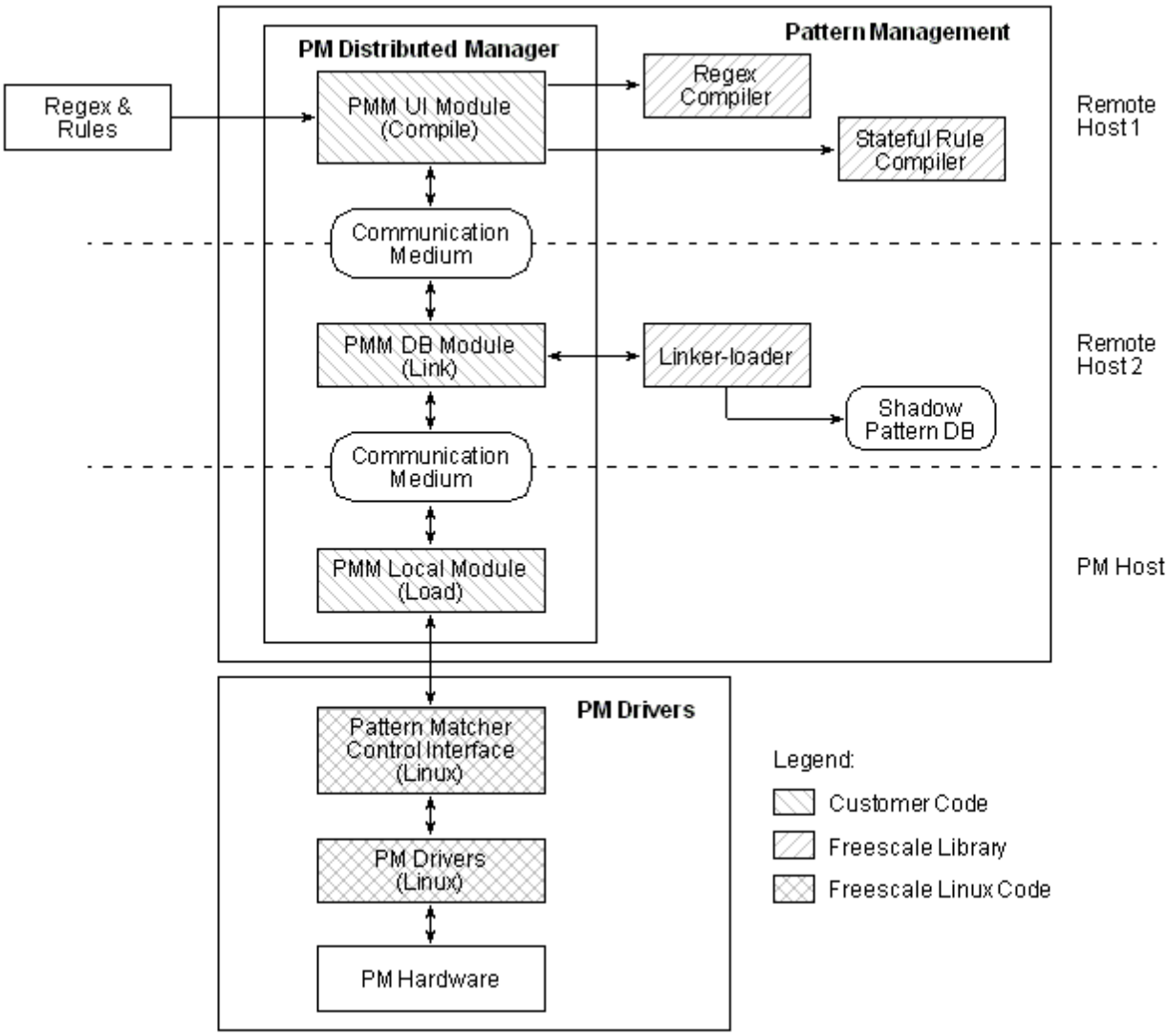
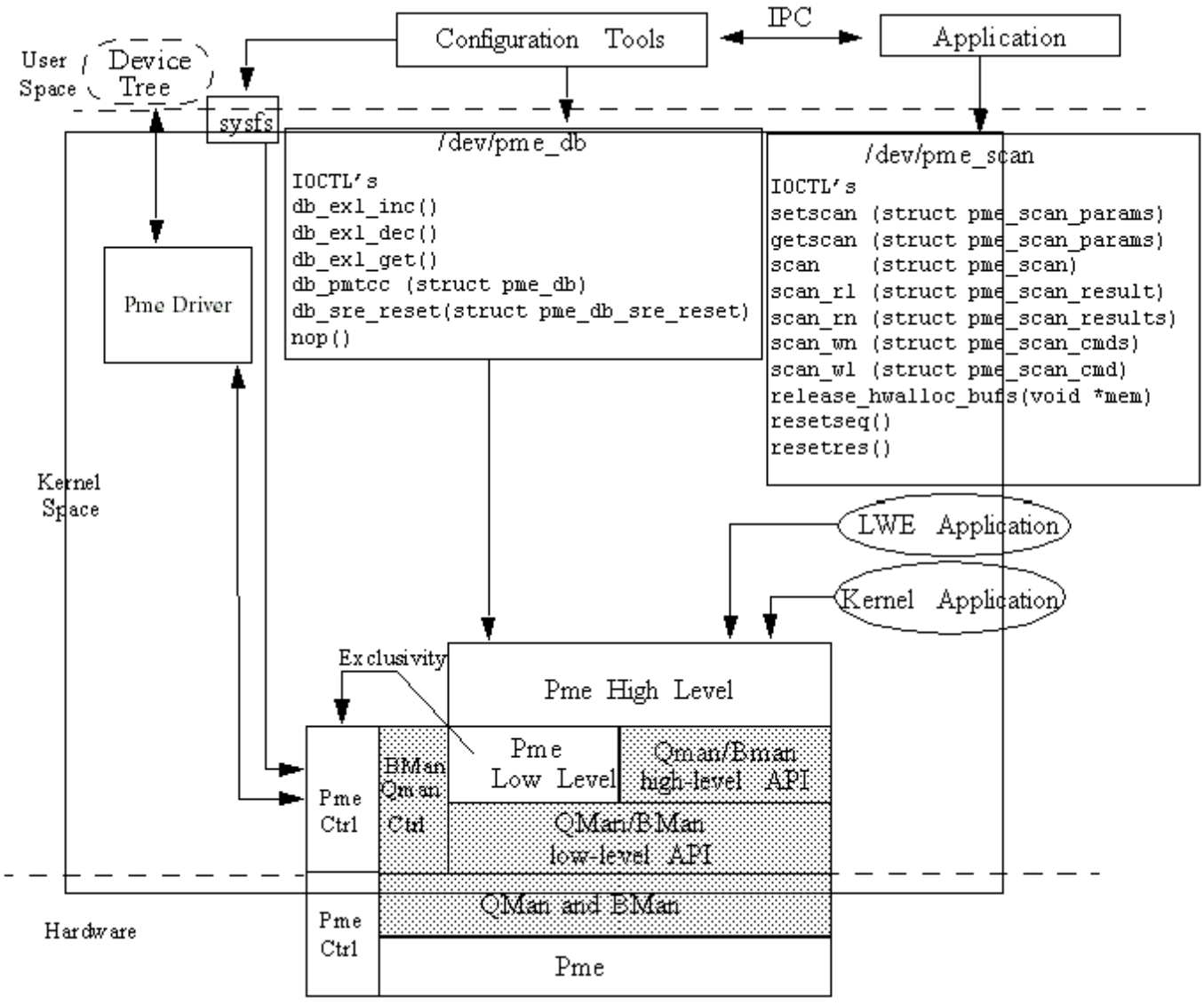
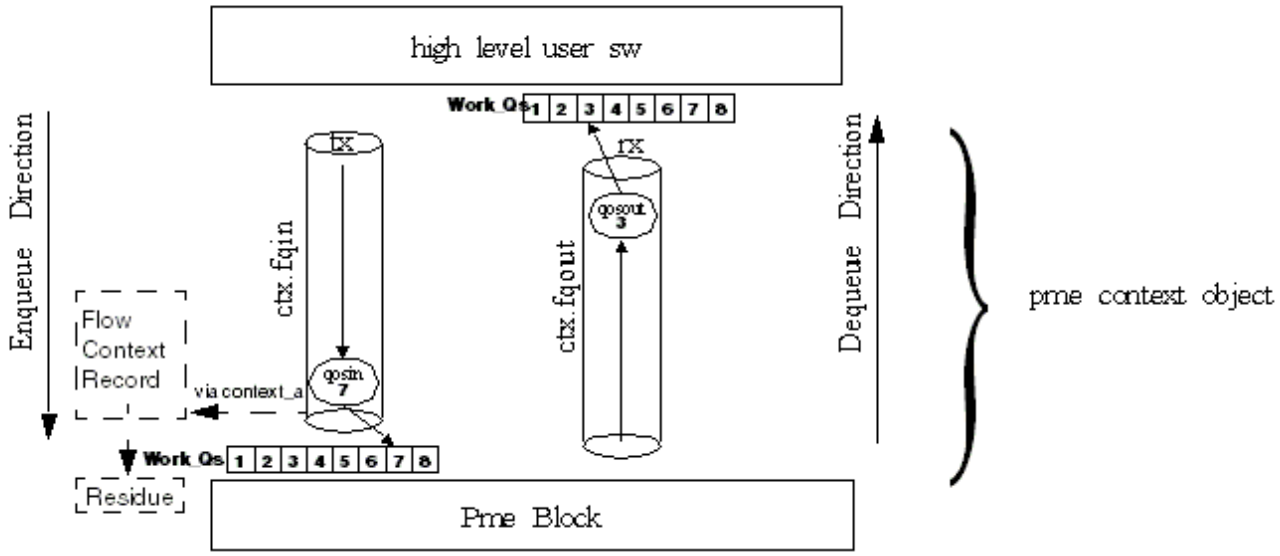
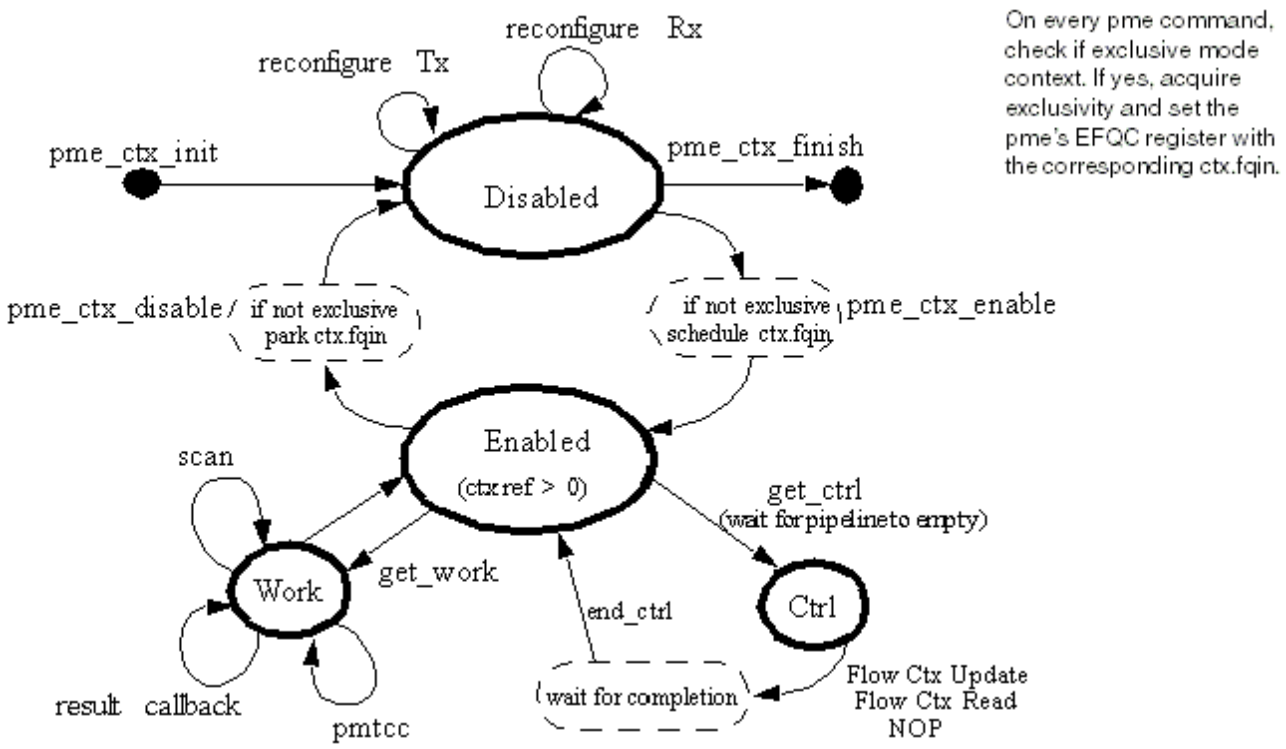
PM SR Execution by DXE   (SRE) :           5345              0           5345

PM SR Execution by SUI   (SRE) :          10659              0          10659

PM SUI With Matches      (DXE) :           5285              0           5285

PM SUI With Reports      (SRE) :          10664              0          10664

PM Input SUIs            (KES) :          10664              0          10664

PM Matches with DRCC     (DXE) :          10786              0          10786
```

7.4.6.2.6.4 Distributed Pattern Management Software

The PMM application can run directly on the host processor with the PM hardware (as a local PMM) or it can be a implemented as a distributed application with only a small part running directly on the host processor. Pattern management software on page 879 depicts the PMM application implemented as a local PMM with a single user-space process and all user interface functionality linked together with the compile, linker-loader, and PMCI libraries. In this local case, the PMM application and PM hardware are collocated on the same system. However, a more distributed PMM application can be designed; that is, expressions and rules can be compiled and linked on a different system from where the PM hardware resides. Running the PMM in a distributed environment can be attractive in low-cost embedded systems without sufficient resources to run the compilers and the linker-loader. Also, it may be desirable to centralize the compilation and linking functions to one location or system and then distribute the compiled and linked configuration to a possibly larger number of pattern scanning systems. There are several ways the PMM functions can be distributed between two or more hosts. Figure 142. Distributed PMM on page 885 shows a design where expressions and stateful rules are compiled on one host, linked on another host, and finally loaded to PM hardware from the processor on the target PM hardware.



Figure 142. Distributed PMM

In a distributed PMM application, the different parts of the PMM application must communicate with each other using communication mechanisms specific to each PMM application design. The networking and communication details are abstracted away from the Pattern Matcher software libraries through appropriate API functions. This abstraction gives the PMM application the flexibility to implement the communication mechanisms appropriate for it, with the following constraints:

- The PM control messages from linker-loader must be reliably delivered to the PMCI, and the sequence of the sent messages must be preserved.
- Communication failures must be reported through appropriate error return codes from the send and receive functions.

7.4.6.2.7 Pattern Matcher driver software

The Pattern Matcher driver for Linux provides applications an interface for configuring and using the Pattern Matcher.

The PME driver software includes a Linux kernel driver and a PME driver library for USDPAA. The driver also provides an ioctl-based Linux user-space interface. The drivers' provide interfaces in Linux kernel and user-space for PME configuration, database setup, and data scanning. For USDPAA, the drivers provide a PME data scanning interface only.

Typical data scanning applications will only use the PME scanning and statistics interfaces. The configuration and PME database setup interfaces would typically be used during device initialization by Pattern Management software like the user-space Pattern Matcher Manager (PMM).

The following section describes the high level functionality of the PME driver software. Details of the driver APIs are available in the PME 2.0 Software API Reference Manual.

The PME2.0 Driver software components are shown in the following diagram.



Figure 143. PME driver components

7.4.6.2.7.1 PME configuration interface

The PME configuration interface is an encapsulation of the PME CCSR register space and the global/error interrupt source. This is available to applications running on a control-plane operating system, the hypervisor and/or a boot manager. Non control-plane applications do not have access to the configuration interface.

7.4.6.2.7.1.1 Externally Configured Settings

The U-Boot software is responsible for configuring the PID to LIODN mapping registers as well as the PME's LIODNR register which is used when the PME accesses its private memory.

7.4.6.2.7.1.2 PME device-tree node

The PME device tree node represents the PME device and its CCSR configuration space. When a Linux kernel has PME support built in, it will react to this device tree node by configuring and managing the PME device.

The device-tree node sits within the CCSR node ("soc") and is of the following form:

```
soc@fe000000 {
[...]
pme: pme@316000 {
compatible = "fsl,p4080-pme", "fsl,pme";
reg = <0x316000 0x10000>;
fsl,pme-pdsr = <0x0 0x20000000 0x0 0x01000000>
fsl,pme-sre = <0x0 0x30000000 0x0 0x01000000>
};
[...]
};
```

7.4.6.2.7.1.2.1 fsl,pme-pdsr

This property specifies the start location and size of the Pattern Description and Stateful Rule Table in system memory. The table base address must be aligned to a natural 128-byte address boundary. The maximum size of the table is 128 Mbytes. The current driver implementation allows this memory resource to be specified via the 'fsl,pme-pdsr' device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. The 'fsl,pme-pdsr' property specifies a 2-tuple of address and size (address is optional), specifying the physical address range. These elements are expressed as 64-bit values, so take two cells each;

```
fsl,pme-pdsr = <0x0 0x20000000 0x0 0x01000000>;
```

Optionally, only the size can be specified and the kernel will try to allocate the contiguous memory during boot time.

```
fsl,pme-pdsr = <0x0 0x100000>
```

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the Linux OS, it will attempt to reserve the range against use by the OS.

7.4.6.2.7.1.2.2 fsl,pme-sre

This property specifies the start location and size of the SRE Context Table in system memory. The table's base address must be aligned to a natural 32-byte address boundary. The maximum size of the table is 4 Gbytes. The current driver implementation allows this memory resource to be specified via the 'fsl,pme-sre' device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. The 'fsl,pme-sre' property specifies a 2-tuple of address and size (address is optional), specifying the physical address range. These elements are expressed as 64-bit values, so take two cells each;

```
        fsl,pme-sre = <0x0 0x20000000 0x0 0x01000000>;
```

Optionally, only the size can be specified and the kernel will try to allocate the contiguous memory during boot time.

```
fsl,pme-sre = <0x0 0x300000>;
```

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the Linux OS, it will attempt to reserve the range against use by the OS.

7.4.6.2.7.2 PME user space interface

The PME user space interfaces comprises the following:

- sysfs
- /dev/pme_db
- /dev/pme_scan

7.4.6.2.7.2.1 sysfs

The *sysfs* interface provides access to PME CCSR space. It is used to control global configuration of PME device parameters and provides an interface for accessing PME statistics. The path to the driver attributes is:

/sys/bus/cf_platform/drivers/cf-fsl-pme

The */dev/pme_db* device is used to send Pme pattern matcher database configuration requests via the Pme's Exclusive Frame Queue Control (EFQC) mechanism. This device requires root permissions. The EFQC exclusivity is referenced counted, so by default it is asserted on-demand and released when the processing is done for the context. However, exclusivity can be maintained by using the db_exl_inc, and db_exl_dec ioctls, which provide supplementary increments and decrements of the reference count. These operations are performed from user space using the following ioctl system calls.

- increment reference count
- decrement reference count
- get the current reference count
- send database request and receive response (synchronous)
- send a nop Pme command

The */dev/pme_scan* device is used to interface with the PME device via the QMan interface. This interface can only be used for scanning operations. The device can be used to perform synchronous (the calling thread is blocked until an operation's completion) scans or asynchronous (once an operation has begun, the calling thread is able to perform other processing but needs to query the completion of the operation later) scans.

7.4.6.2.7.2.2 Pme Scan

The /dev/pme_scan device is used to interface with the Pme device via the QMan interface. This interface can only be used for scanning operations. The device can be used to perform synchronous (the calling thread is blocked until an operation's completion) scans via the scan() api or asynchronous (once an operation has begun, the calling thread is able to perform other processing but needs to query the completion of the operation later) scans via the scan_w[1n] and scan_r[1n] APIs. Only flow mode scanning requests are supported by this device. These operations are performed from user space using the ioctl system calls specified below.

- set parameters for scanning operations
- get (retrieve) currently set scanning operation parameters
- reset sequence number
- reset residue
- do single synchronous scan
- send single asynchronous scan command
- send multiple asynchronous scan commands
- get single synchronous scan response
- get multiple synchronous scan responses
- release bman acquired buffers to bman: release_hwalloc_bufs

The settable and retrievable scan parameters are grouped as follows:

- Residue attribute
  - residue enable/disable
  - current number of byte in the residue (read only)
- SRE attributes
  - session_id
  - report verbosity
  - end of sui report enabled/disbabled
- DXE attributes
  - compare limit
  - match limit
- Pattern attributes
  - pattern set
  - pattern subset

7.4.6.2.7.3 PME Linux kernel, USDPAA driver interface

The PME kernel/USDPAA driver provides a high and low level interface for PME users.

7.4.6.2.7.3.1 PME High-level Driver API

The PME high-level APIs provide a call-back based interface to the PME. The driver provides APIs to manage flow context and issue PMTCC and scan commands. The high-level driver internally co-ordinates the commands to PME and corresponding results. The user-specified call-back functions are called with results from PME. The driver also returns the user-specified context information provided by the application when invoking the call-back function. This saves the applications from maintaining a mapping of active scan requests and their related contexts.

The high-level interface provides the following:

- pme context management. Management operations include: initialize, destroy, disable, enable, queury_state, reconfigure. The pme context is comprised of:
  - input/output frame queues
  - flow context record (when in flow mode)
  - frame queues scheduling levels
  - exclusivity state
  - stashing control of frame queue context
  - residue (optional when in flow mode)
- manage mux/demux of scans and pmtcc commands.
- pme input/output qman frame queue setup, modification and teardown.
- registration of user provided scan and pmtcc result callbacks
- statistics management



Figure 144. Pme Context Object

7.4.6.2.7.3.1.1 PME Context

The Pme high level layer is represented by a pme_context object. This object allows a user to communicate with the Pme device using the encapsulated QMan interface via two driver allocated frame queues.

The PME context is a modal object which can be initialized in one of the following modes:

- scan_flow: This mode permits the user to send *scan, nop, flow context write and flow context* read PME requests. Upon enabling the context the input frame queue is placed in a scheduled state
- scan_direct: This mode permits the user to send *scan and nop* PME requests. Upon enabling the context the input frame queue is placed in a scheduled state

The following PME Context modes exist for use by the user-space Pattern Matcher Management software. They can also be used by applications making dynamic changes to the PME database or for exclusive scan operations..

- pmtcc: In this mode the context is placed in *exclusive* mode. The input (tx) frame queue is left in an unscheduled (parked) state. Only *pmtcc and nop* pme requests are permitted. This mode is only available if the api is built with control functionality and if the operating system has access to the PME's CCSR map.
- scan_exclusive_direct: This mode permits the user to send *scan and nop* PME requests. Upon enabling the context the input frame queue is left in an unscheduled (parked) state. This mode is only available if the api is built with control functionality and if the operating system has access to the PME's CCSR map.
- scan_exclusive_flow: This mode permits the user to send *scan, nop, flow context write and flow context read* PME requests. Upon enabling the context the input frame queue is left in an unscheduled (parked) state. This mode is only available if the api is built with control functionality and if the operating system has access to the PME's CCSR map

Once initialized, a pme_context object cannot change modes. If a different mode is required, a new context must be allocated/ initialized. The following diagram depicts the pme_context state diagram. Depending on the mode, not all possible states are attainable.



Figure 145. PME Context State Diagram

The PME high-level APIs use the QMan high-level APIs. As such, the driver maintains control over certain fields, like the "contextB" field in the frame descriptor, and these are not available for direct application use. Applications can specify their own per-command context information. This is done by embedding the per-command token field in a larger structure which is returned back in the command result.

7.4.6.2.7.3.1.2 Typical Scan Operation Flow

Typical steps required to setup and perform scanning operations:

- Initialize a pme context via pme_ctx_init().
- Enable the ctx via pme_ctx_enable()
- If operating in flow pme, setup the flow context using pme_sw_flow_new(), setting the appropriate fields and then invoking pme_ctx_ctrl_update_flow(). The flow is released on teardown via pme_sw_flow_free().
- The context is ready for scanning operations. Build a frame description object and invoke the scan operation via pme_ctx_scan().
- Upon receiving the result from the PME, the callback function specified in the pme context object will be invoked in interrupt context in the Linux kernel. For USDPAA, the callback is invoked in the application context.
- Process the resulting output frame description object passed into the callback function.
- Once scanning is done, disable the context via pme_ctx_disable()
- Release all resources in the context via pme_ctx_finish().

7.4.6.2.7.3.1.3 Statistics Management

The high-level driver also provides an interface to read PME statistics counters. The values returned are software accumulated versions of the counter. The driver maintains the statistic counters by periodic (user-programmable) accesses to the read-

reset hardware counters. The access have to be frequent enough to prevent the counters from rolling over. The accumulated statistics are available to the user space application via the sysfs interface.

7.4.6.2.7.3.2 PME Low-level Driver API

The low level API is a hardware abstraction layer (HAL) where users' have complete control over structures used to communicate with the PME. This includes initialization of the input and output frame queues and the ownership for setting up the frame descriptor including command codes and the "contextB" field. The low level API is used by the high level driver for communicating with the PME.

The low-level interface provides the following;

- allocation/deallocation of residue data
- allocation/deallocation of hw flow context data
- allocation/deallocation of sw flow context data
- pme frame queue command/status abstraction
- frame descriptor setup for following pme commands: *nop, flow context write, flow context read, pmtcc* and *scan*.

The low level PME driver APIs do not encapsulate QMan portal interactions. Users are assumed to be using the low level PME APIs to have control on their interaction with the devices - including the use of QMan and BMan low level APIs.

7.4.6.2.7.4 Driver Build Configuration

The PME driver is built using the standard build process. The build configuration options and default values are specified in: <linux directory>/drivers/staging/fsl_pme2/Kconfig.

7.4.6.2.8 Data Scan Sample Application

The data scan sample application covered in this chapter are as follows:

- *Pattern matcher scan demo application*. A command line Linux user-space application that scans data strings given on a command line parameter or data from a file.
- *Snort*. An open-source network intrusion prevention and detection system using a rule-driven language, which combines the benefits of signature, protocol, and anomaly-based inspection methods.

7.4.6.2.8.1 Pattern Matcher Scan Demo Application (pm_scan_demo)

```
pm_scan_demo
```

is a sample command line Linux user-space application that scans data strings given on a command line parameter or data from a file. The data is scanned for patterns that are preloaded to the Pattern Matcher hardware database by the Pattern Matcher manager application (PMM). The pattern match results are displayed to standard output. This application demonstrates the use of Pattern Matcher hardware assist to scan for complex regexes and stateful rules.

7.4.6.2.8.1.1 Usage

The **pm_scan_demo** command line parameters are described in the help text shown here. The subsections provide simple examples to demonstrate the use of these parameters and how they apply to the Pattern Matcher features.

```
# pm_scan_demo --help
Usage:
        pm_scan_demo [<options>]
Description:
        A sample application that scans user entered string or file, and
        prints the reported matches.
Options:
  -s, --str <string>
        Scan a string, passthru mode only.
```

```
        May use \xNN to represent a byte in hexidecimal.
  -f, --file <filename>
        Scan the content of a file.
        Multiple strings and files may be scanned sequentially as
        separate scan units by repeating the -f and -s options.
  -r, --residue
        Enable residue feature.
 --set <num>
        Expression set to scan against.  Default = 0.
  --subset <num>
        Expression subset-mask to scan against.  Default = 0xffff.
  --session <num>
        Stateful rule session id number.  Default = 1.
  --verbose
        Enable verbose report.  HW will report more data per match.
  --silent
        Disable printing of match report.
  --detail
        Shows scan data covered by matches. Requires ANSI/VT100 terminal.
Example:
        pm_scan_demo -s abcd -s defg --residue --detail
```

7.4.6.2.8.1.1.1 Simple Literal String Scan

The 'pm_scan_demo: Simple Scan of Literal Pattern' example that follows uses the PMM application to add a simple regex to scan for the string

```
abcd
```

and uses the

```
pm_scan_demo
```

application to demonstrate the match result. Notice that the match result, highlighted in bold, shows an offset of 0x0:8, which is

```
workunit_offset_within_stream:match_offset_within_workunit
```

Also, notice that the

```
match_offset
```

is at the end of the match (not the start of the match), which is an offset of 8 in this example.

pm_scan_demo: Simple Scan of Literal Pattern

```
# cd <path of the pattern matcher applications>
# <path>pmm
Successfully created the PMM DB.
pmm> add regex name p1 exp /abcd/i tag=0x100
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> show exp all
name=p1           expression="/abcd/"   options="i = 1, m = 0, s = 0, report = 1, tag
= 0x00000100"
total number of expressions = 1
total number of rules       = 0
pmm> commit
```

```
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
# <path>pm_scan_demo -s "xyz abcd xyz"
#00: Scanning 12 bytes.
match(0x01): len=0x04 offset=0x000000000000:00000008 tag=0x00000100
Number of successful scan:    1
Number of full match:         1
Number of inconclusive match: 0
Number of rule report:        0
```

7.4.6.2.8.1.1.2 Use of Residue

The 'pm_scan_demo: Scan of Literal Pattern that Span Multiple Work-Units' example that follows uses the PMM application to add a simple regex to scan for the string abcd and uses the pm_scan_demo application to demonstrate detection of patterns that cross work units. Notice that the second match result, highlighted in bold letters, shows two matches. The first match crosses between the two work units.

pm_scan_demo: Scan of Literal Pattern that Span Multiple Work-Units

```
# pmm
Successfully created the PMM DB.
pmm> add regex name p1 exp /abcd/i set=0 tag=0x100
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm> commit
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
pmm> quit
Terminating the PMM application.
# <path>pm_scan_demo --set=0 -s "abxyz abcd abxyz abcd xyz"
#00: Scanning 25 bytes.
match(0x01): len=0x04 offset=0x000000000000:0000000a tag=0x00000100
match(0x01): len=0x04 offset=0x000000000000:00000015 tag=0x00000100
Number of successful scan:    1
Number of full match:         2
Number of inconclusive match: 0
Number of rule report:        0
#
# pm_scan_demo --set=0 --residue -s "abxyz ab" -s "cd abxyz abcd xyz"
#00: Scanning 8 bytes.
( No match or rule reported. )
#01: Scanning 17 bytes.
match(0x01): len=0x04 offset=0x000000000008:00000002 tag=0x00000100
match(0x01): len=0x04 offset=0x000000000008:0000000d tag=0x00000100
Number of successful scan:    2
Number of full match:         2
Number of inconclusive match: 0
Number of rule report:        0
#
```

7.4.6.2.8.1.1.3 Use of Pattern Set and Subsets

The 'pm_scan_demo: Scan of Patterns in Different Sets' example that follows uses the PMM application to add regexes in different sets and subsets, and the 'pm_scan_demo: Scan of Patterns in Subsets' example that follows uses the pm_scan_demo application to demonstrate scanning for patterns in sets and subsets. Notice that the match results, highlighted in bold letters, show matches appropriate to the pattern set or subsets.

pm_scan_demo: Scan of Patterns in Different Sets

```
# pmm
Successfully created the PMM DB.
pmm>
pmm> add regex name p1 exp /abcd/set=1 tag=0x100
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> add regex name p2 exp /pqrst/set=2 tag=0x200
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> commit
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> quit
Terminating the PMM application.
# pm_scan_demo --set 1 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x04 offset=0x000000000000:00000008 tag=0x00000100
Number of successful scan:    1
Number of full match:         1
Number of inconclusive match: 0
Number of rule report:        0
# pm_scan_demo --set 0 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
( No match or rule reported. )
Number of successful scan:    1
Number of full match:         0
Number of inconclusive match: 0
Number of rule report:        0
# pm_scan_demo --set 2 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x05 offset=0x000000000000:00000012 tag=0x00000200
Number of successful scan:    1
Number of full match:         1
Number of inconclusive match: 0
Number of rule report:        0
#
```

pm_scan_demo: Scan of Patterns in Subsets (continued)

```
# pmm
Successfully created the PMM DB.
pmm>
pmm> add regex name p1 exp /abcd/set=1 subsets=0x1 tag=0x100
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> add regex name p2 exp /pqrst/set=1 subsets=0x2 tag=0x200
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
```

```
pmm> commit
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> quit
Terminating the PMM application.
# pm_scan_demo --set 1 --subset 0x1  -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x04 offset=0x000000000000:00000008 tag=0x00000100
Number of successful scan:    1
Number of full match:         1
Number of inconclusive match: 0
Number of rule report:        0
# pm_scan_demo --set 1 --subset 0x2  -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x05 offset=0x000000000000:00000012 tag=0x00000200
Number of successful scan:    1
Number of full match:         1
Number of inconclusive match: 0
Number of rule report:        0
# pm_scan_demo --set 1 --subset 0x3  -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x04 offset=0x000000000000:00000008 tag=0x00000100
match(0x01): len=0x05 offset=0x000000000000:00000012 tag=0x00000200
Number of successful scan:    1
Number of full match:         2
Number of inconclusive match: 0
Number of rule report:        0
```

7.4.6.2.8.1.2 Source Code Description

The demo application source code is in the pm_scan_demo.c file. The basic operations of the application are as follows:

1. Read the input string from the command line or the content of the specified file into memory.

   The entire file is read into memory, so the file size is limited by the available memory.

2. Open the Pattern Matcher scanner device using the PM device driver APIs:

   ```
   scan_fd = open(PME_DEV_SCAN_PATH, O_RDONLY)
   ```

3. Initialize the scanner device parameters such as set, subset, session ID in a local scanner_params structure and then configure the scanner device with these parameters.

   ```
   ioctl(scan_fd, PMEIO_SETSCAN, &scanner_params)
   ```

4. Set up a structure to pass scan data, size of scan data, and other required parameters for the scan operation. The structure type is struct pme_scan. Then call the driver to execute the scan operation.

   ```
   ioctl(scan_fd, PMEIO_SCAN, &pm_oper)
   ```

5. After the scan operation, the result of the scan such as match report buffer length and error flags are also returned in members of the struct pme_scan structure.

6. This demo uses BMan buffers to receive match reports. These free buffers should be freed with this call:

   ```
   ioctl(scan_fd, PMEIO_RELEASE_BUFS, &pme_oper.result.output.data)
   ```

7.4.6.2.9 Software components

The software modules that accompany the Pattern Matcher 2.0 hardware are both production quality and sample software modules. The production and sample quality code is defined as follows.

Production-quality code:

- Completeness; includes necessary functionality to make it usable in a production environment
- Robustness; includes all necessary recovery functionality
- Fully validated/verified
- Supported

Sample code:

- Software example (or template) to help customers/partners to implement their own versions
- May or may not be of production quality but not often viewed as production quality
- Sometimes viewed as bare-bones (does the minimum functions)

7.4.6.2.9.1 Software Modules

The PM Software Modules table lists software version 2.0 software modules with packaging information and the supported platforms for each module.

PM Software Modules

Module	Packaging	OS	Platform Supported
Regex Compiler	source code	Linux	QorIQ™, x86
Stateful Rule Compiler	source code	Linux	QorIQ™ x86
Linker-Loader	source code	Linux	QorIQ™
PM Control Interface	source code	Linux	QorIQ™
PM Driver	Kernel patches (source code) USDPAA Library (source code)	Linux	QorIQ™

7.4.6.2.9.2 Sample Code

Below, the Pattern Matcher Sample Code table lists software version 2.0 sample code modules.

Pattern Matcher Sample Code

Module	Packaging	OS	Platform Supported
PM Manager (PMM)	Executable and source code	Linux	QorIQ™
Pattern scanner application (PM Scan Demo)	Executable and source code	Linux	QorIQ™
pmcd	Executable and source code	Linux	QorIQ™
pmcc	Executable and source code	Linux	QorIQ™

7.4.6.2.10 Pattern Matcher FAQ

7.4.6.2.10.1 What is it and why is it needed?

The Pattern Matcher Engine (PME, or sometimes just PM) offloads processing from the PowerQUICC core(s) by performing, in hardware, regular expression pattern matching of data against patterns stored in the pattern database. This type of unanchored searching of patterns is processing-intensive and is difficult to perform in software at multi-gigabit rates. Even in sub-gigabit rate deployments, the offload provided by the PME can free the core(s) to perform more complex, content-oriented tasks.

7.4.6.2.10.2 What is a regular expression?

A regular expression (or regex) is a description of a set of strings, written using a powerful and flexible notation/syntax. The expression may contain literal characters and meta-characters, supporting wildcards, grouping, quantification and alternation. Matching a regular expression refers to comparing the arrangement of symbols described by the regular expression against an input string, usually incoming network data.

7.4.6.2.10.3 What types of regular expressions are supported by the supplied compiler?

The first supplied compiler supports a major subset of the PERL regular expression syntax, as well as capabilities beyond that provided by PERL.

7.4.6.2.10.4 What is a pattern?

A pattern is an arrangement of symbols and instructions used by the PME to evaluate against an input string or SUI. It is the hardware representation of a regular expression.

7.4.6.2.10.5 What is an SUI?

A string under inspection (SUI) refers to the string of bytes to be searched by the PME. The PME evaluates this input string against patterns in the pattern database. Often, but not necessarily, the SUI refers to incoming network data.

7.4.6.2.10.6 Why is regular expression matching difficult to perform at multi-gigabit rates in software?

In order to match expressions against the incoming data, the incoming data must be scanned byte by byte and evaluated against the thousands of expressions in the database. The expressions not only contain literal characters, but may contain wildcards, repeats, alternations, and other regular expression constructs, where each expression may represent an almost infinite number of strings. Regular expression matching, therefore, requires not only matching of literal characters, but also what is known as regular expression evaluation.

7.4.6.2.10.7 What applications require pattern matching?

Pattern matching of application layer protocol signatures forms the basis for high-performance application-aware networking. It is used in various content processing and security applications such as intrusion detection/prevention, anti-virus, anti-spam, and other applications where the content of the packet or stream needs to be examined. Regular expressions are gaining widespread adoption over literal string matching for these types of applications because of their expressive power and flexibility to describe complex patterns.

7.4.6.2.10.8 Can the PME scan packet headers as well as packet content?

Not easily (nor is this usually required to be performed in hardware). The PME was designed for the more difficult task of scanning character-based content for unanchored matches. Scanning packet headers requires bit-level granularity, making it impractical for the byte-based PME to perform. However, since packet headers are typically anchored, it is often reasonable to delegate this task to a software pre-processing stage.

7.4.6.2.10.9 Can the PME detect patterns across packet boundaries?

Yes. The PME uses the residue method to detect cross-packet patterns. The residue method is stateless and therefore can scale as the number of patterns grow. There is no hard limit on how many packets the pattern may span (other than the limit imposed by the maximum length of the pattern), or whether the content in the first packet matches the start of one or the start of many patterns.

7.4.6.2.10.10 What is the maximum length of a pattern?

128 bytes long. Patterns in excess of 128 bytes can be supported through stateful rules functionality.

7.4.6.2.10.11 How many patterns can be searched for simultaneously?

32000.

7.4.6.2.10.12 What is an NFA?

Non-deterministic finite automaton (NFA) is one of two generally accepted methods for performing pattern matching. The NFA evaluates each alternative at every byte position of the incoming data. Unlike DFA, an NFA may have multiple transitions out of a state with the same input. As the NFA evaluates each transition, it must remember and "backtrack" to execute alternate transitions if the evaluation fails.

7.4.6.2.10.13 What is a DFA?

Deterministic finite Aautomaton (DFA) is one of two generally accepted methods for performing pattern matching. A DFA tracks all possible matches as each byte of data is consumed. Each possible input character of the alphabet set is associated with at most one transition in any given state.

7.4.6.2.10.14 Which method is better: NFA or DFA?

There is no single correct answer. An NFA typically supports more regex features but requires many passes over the data as each alternative is evaluated, whereas a DFA only passes over the data once. Part of the reason a DFA only needs to examine the data once is that a lot of work has been performed up front during the compilation of the expressions, such that the compiled DFA state graph contains all the interdependencies between the different expressions. Because of these interdependencies, an incremental change to the expression database may require a recompilation of the entire DFA state graph. For these reasons, a software-based DFA has less flexibility, requires more memory, and has a longer expression compile time, but tends to perform faster than a software-based NFA.

7.4.6.2.10.15 Is the PME an NFA or a DFA?

Since the drawbacks of a DFA, such as the longer compile time, are inherent and cannot be easily overcome, the PME was modeled after an NFA, augmented with specific techniques to improve performance. Part of the reason a traditional NFA is slower is because the NFA needs to evaluate each expression at every byte position of the incoming data. When there are a large number of expressions or alternatives, performance drops. To overcome this, the PME incorporates a hardware pre-processing stage called the Key Element Scanner, which works by eliminating almost all candidate expressions prior to the regex evaluation such that the NFA only needs to perform the regex evaluation for one or a very small number of expressions at any given byte position. In fact, under normal operation when there is no potential match, the regex evaluation does not even need to be performed.

7.4.6.2.10.16 Where is the pattern database stored?

The pattern database is stored in a combination of on-chip memory and the main SDRAM.

7.4.6.2.10.17 Why is SRAM not required?

One key design objective of the PME is to minimize any reliance on expensive low-latency memories. The solution the design team undertook to achieve this objective makes use of a combination of on-chip and external memory. On-chip memory contains just enough pattern information to filter out all but the most likely matches. Only when a likely match has occurred does the PME require an external SDRAM memory access. Pipeline stalls due to memory accesses are minimized with the presence of FIFOs in between the different internal components of the PME. This relative infrequency of external memory access, in addition to the interconnecting FIFOs, leads to performance with a lower dependency on memory latency.

7.4.6.2.10.18 Does the PME support case-insensitivity?

Yes. Support for case-insensitivity is inherent in the design, and does not lead to pattern explosion.

7.4.6.2.10.19 Can contextual searches be performed?

The situation where interest for a specified pattern is predicated upon the detection of another pattern can be performed using Stateful rules.

7.4.6.2.10.20 What is a stateful rule?

A Stateful rule is a set of user-defined actions that are executed by the PME when specified pattern match events occur. The actions include changing state, assignments, bitwise operations, addition, subtraction, and relational operations.

7.4.6.2.10.21 What are the benefits of stateful rules?

Stateful rules enable the PME to go beyond basic pattern matching. Rather than just informing software whenever a pattern is matched, stateful rules allow the PM to act on pattern match events such that a match is declared to software only when user-specified conditions are met. For example, a stateful rule can be written to allow the PME to discriminate between matches found in the header portion versus the body portion of the application message. A stateful rule can allow the PM to declare a match only when all constituent parts of a signature match (such as in an anti-virus application). A Stateful rule can allow the PM to track a normal protocol exchange and glean information between a client and a server (such as in a SIP-aware application retrieving information on the media channel). In all these examples, Stateful rules postpone the need for application software to be invoked until concrete actions need to be taken.

7.4.6.2.10.22 How many stateful rules are supported?

32,768.

7.4.6.2.10.23 What information does the pattern match and stateful rule reports contain?

The pattern match report contains information such as the byte offset of the detected pattern (right edge of the match) and the pattern identifier. The Stateful rule report is customizable, and can include more detailed information on the match, any captured fields, and any context previously saved.

7.4.6.2.10.24 What's the performance of the PME?

The raw hardware performance of the PME is 2.4 Gbps in the MPC8572 PowerQUICC processor. However, the actual system performance may vary due to software and memory access overhead.

7.4.6.2.10.25 What software is included and supported for the PME?

All the software needed to compile, link, and load expressions into the hardware are provided. This includes the regular expression compiler, the stateful rule compiler, the linker-loader, and the low-level drivers for interacting with the hardware.

7.4.6.2.10.26 What operating systems are supported?

In general, the supplied software has been designed to be agnostic of the operating system where possible. The regular expression compiler and the stateful rule compiler are supported on Linux; the linker-loader is supported for Linux; the low-level drivers for interacting with the PM are supported for Linux. Porting of the supplied software to other operating systems is not expected to be difficult.

7.4.6.2.10.27 How can existing pattern management application software use the PME?

The PME software is provided as a library of functions with clearly-defined APIs and capabilities to compile, link, and load the patterns onto the hardware. Both single expression and multi-expression inputs are supported. Some application-specific pre-processing may be required to convert existing expressions into the PERL-like format understood by the supplied compiler.

7.4.6.2.10.28 How does software invoke the PME?

The PME provides a DMA-based "look-aside" interface with four independent DMA channels, which the supplied driver software uses to command pattern matching/decompression work, and to receive notifications of completed pattern matching/decompression work. For each DMA channel, a ring structure (array of entry locations) of programmable depth is used to dispatch commands to the PME. Similarly, a ring structure (array of entry locations) of programmable depth is used by the PME to inform the software of events such as the availability of a pattern scan result. The PME supports "gather buffer lists" for specifying input data and "scatter buffer lists" for specifying the location of output data.

7.4.6.2.10.29 Can software perform other tasks while the PME is scanning the data?

Yes, the supplied driver supports both blocking and non-blocking APIs.

7.4.6.2.10.30 How does software obtain the match results?

Completion of a command can be signaled through an interrupt or through polling. Application software can then read the results from the notification FIFO, including any results in a scatter list.

7.4.6.2.11 Revision History

Document Revision History

Rev. Number	Date	Substantive Change(s)
2	8/2009	Added support for PME2.0 (supporting P4080).
1	4/2008	Removed sections 2.1, 2,2, 2.3, 3.1, and 3.2.
0	11/2007	Initial public release.

7.4.6.3 Pattern Matcher 2.0, 2.1, 2.2 Software API Reference Manual

7.4.6.3.1 Introduction

This reference manual provides detailed information about Pattern Matcher software application programming interfaces (APIs).

This manual is intended for designers developing software for the NXP Pattern Matcher. Readers are expected to be familiar with Pattern Matcher hardware functionality and the information provided in the Pattern Matcher 2.0 Software User's Guide prior to using this document.

All programming interfaces available for use with the Pattern Matcher software are listed in this document. The interfaces are grouped according to software components within the document. Each interface includes a functional description as well as information about syntax and return values. Each interface description also provides information about libraries and kernel modules as well as header files required to use that programming interface.

The Pattern Matcher software is provided as linkable C libraries and kernel drivers, as well as executable binaries where appropriate. This document describes the programming interface for the software. Software executables are described in the Pattern Matcher Software User Guide.

7.4.6.3.1.1 Conventions

The following notational conventions are used in this guide:

Courier monospaced type indicates code examples and file or directory names

Italic type used within interface descriptions indicates interface parameters

7.4.6.3.2 Software Component Overview

The NXP Pattern Matcher provides high-performance hardware pattern matching of data. Input data is searched for user specified patterns by the hardware. The Pattern Matcher software enables users to efficiently configure, use, and manage the Pattern Matcher hardware. The software comprises the following components:

- Regex Compiler-The regex compiler converts user defined patterns, specified as regular expressions, into patterns in a NXP hardware-specific format.
- Stateful Rule Compiler-Stateful rules define stateful relationships between pattern matching results and also specify the actions to be taken in the defined states. The stateful rule compiler converts user specified stateful rules into a NXP hardware-specific format.
- Pattern Matcher Configuration - Pattern Matcher Configuration software provides users with an interface to configure Pattern Matcher hardware. The simplest programatic interface to configure the pattern matcher hardware is with the PMC API. The following two APIs are available for backward compatibility with existing pattern management applications.
- Linker-Loader-The linker-loader accepts patterns and stateful rules encoded in the NXP hardware-specific format and generates a hardware-optimal pattern and reaction database. The linker-loader also provides interfaces to configure and monitor the Pattern Matcher database used by hardware.
- Control Interface-The PME control interface module (PMCI) provides C functional interfaces to send and receive Pattern Matcher control commands to the Pattern Matcher through Pattern Matcher driver software. The PME control commands initialize PME driver and configure the PME hardware tables. The PMCI module converts complex PME control commands into PME driver primitives.
- Driver-The Pattern Matcher driver software provides users with an interface to the Pattern Matcher. The driver software is targeted to the linux kernel and linux user space (USDPAA) targets. For Linux, the driver provides user-level and kernel-level interfaces to efficiently transfer, control, and scan data to the hardware, and provide pattern matching results to applications. All PME database updates are expected to be done on a control core running Linux. For USDPAA, the driver provides an interface to scan data and get match results. In all cases, the PME driver encapsulates direct interaction with the underlying QMan and BMan functionality. There is also a platform specific considerations to bear in mind when working with the interfaces described here, please see

NOTE

The APIs in this document are grouped according to above software components. Additional information about each software component is provided in the appropriate section.

7.4.6.3.3 Regular expression compiler

This chapter describes the application programming interface (API) for the Pattern Matcher regular expression compiler (PMREC). The PMREC allows the user to compile regular expression patterns into a format that is used internally within the NXP Pattern Matcher hardware. All the PMREC functions run to completion, meaning a PMREC function returns the operation performed by that function has been completed.

PMREC is delivered as a linkable C library called libregex.a.

PMREC Required Files

File	Description
libregex.a	Contains all the PMREC objects
pmrec.h	Defines the interface to the PMREC module
generic_types.h	Contains generic type definitions and is included by pmrec.h
pm_defs.h	Defines the interface that is common to PMLL, PMREC, PMSRC, and so on, and is included by pmrec.h
inttypes.h	Defines the basic integer types and is included by generic_types.h
stdbool.h	Defines the bool type and is included by generic_types.h

7.4.6.3.3.1 pmrec_compile

pmrec_compile-Compile a file of regular expressions.

Synopsis

```
#include <pmrec.h>

pmrec_error_codes_t pmrec_compile (

char                    *input_file_name_p,

char                    *output_file_name_p,

pmrec_module_options_t *options_p,

char                    **compile_msg_p);
```

Description

Compile regular expression(s) into NXP hardware format. The user must supply valid input and output file names. The user must also supply two optional parameters (either set to a valid value or set to NULL). The first is an object that contains compile options. The second is a pointer to a character pointer that allows the compiler to pass back a detailed compiler message. Both of the last two parameters may be set to NULL if default options and no compiler message are desired. If a pointer is supplied and no compiler message is generated, the string will be set to NULL. Note that if a pointer is supplied for the compiler message, the user is responsible for freeing the string that is returned by the compiler library.

pmrec_module_options_t is defined in pmrec.h. It is listed below for easy reference.

typedef struct pmrec_module_options

```
{
    pmrec_debug_levels_t           debug_level;
    pmrec_group_definition_t       group_definition;
    char                           group_def_filename_p;
    pmrec_equivalence_definition_t equivalence_definition;
    char                           *equiv_def_filename_p;
    bool                           warnings_are_errors;
    bool                           suppress_warnings;
bool                           hide_strings;
   bool                            one_byte_triggers;
   bool                            silicon_8572_rev_1_0;
} pmrec_module_options_t;
```

group_def_filename_p and equiv_def_filename_p are for future use and should be set to NULL.

A typical setting for the options would be:

```
pmrec_module_options_t options = { pmrec_debug_none_e,
                                   pmrec_groups_default0_e,
                                   NULL,
                                   pmrec_equivalence_default0_e,
                                   NULL,
                                   false,
                                   false};
```

Return Value

This function returns a code of type *pmrec_error_codes_t*. This type is defined in pmrec.h. The possible return codes are as follows:

pmrec_ok_e

Success. Generally expect to see this code.

pmrec_warning_e

Compile warning found.

pmrec_internal_error_e

Should not see this error code. Indicates an unexpected condition.

pmrec_no_trigger_found_e

No triggerable symbols found in expression

pmrec_missing_end_block_e

Should not see this error code. Indicates an internal issue with the data structures.

pmrec_missing_end_class_e

Should not see this error code. Indicates an internal issue with the data structures.

pmrec_no_output_file_e

No output file was specified.

pmrec_nested_repeat_found_e

A nested repeat in the expression was found. Hardware does not support this.

pmrec_expression_too_large_e

Expression is too large to fit within the allowable instructions.

pmrec_nested_capture_found_e

A nested capture in the expression was found. Hardware does not support this.

pmrec_compile_failed_e

Generic compile failure. Specific details will be in the compile message.

pmrec_input_file_open_error_e

Error opening supplied input file.

pmrec_invalid_element_size_e

Should not see this error. Indicates an internal issue with data structures.

pmrec_output_file_open_error_e

Error opening supplied output file.

pmrec_output_file_write_error_e

Error writing to output file.

pmrec_max_test_lines_exceeded_e

Expression generates too many test lines to fit in hardware.

pmrec_no_start_position_found_e

Should not see this error. Indicates an issue finding start position for test lines.

pmrec_code_string_mismatch_e

Should not see this error. Indicates erroneous number of error code strings.

pmrec_must_expand_grouped_duplication_e

Indicates a condition the compiler cannot deal with currently. Specifically expressions like (ab(c|d|e))+. This should be expanded to (abc|abd|abe)+

pmrec_malloc_failure_e

Failure while calling malloc. System memory likely exhausted.

pmrec_invalid_option_e

Invalid option was passed through compiler options.

pmrec_invalid_group_def_e

Invalid group definition was passed through compiler options.

pmrec_invalid_equiv_def_e

Invalid equivalence definition was passed through compiler options.

pmrec_syntax_error_e

Regex format or syntax was invalid

Related information

pmrec_get_error_string on page 906

7.4.6.3.3.2 pmrec_get_error_string

pmrec_get_error_string-Get the string associated with a compiler error.

Synopsis

```
#include <pmrec.h>

const char *pmrec_get_error_string (pmrec_error_codes_t, code);
```

Description

The function returns a brief string describing the supplied error code.

Return Value

Pointer to the error string.

Related information

pmrec_compile on page 903

7.4.6.3.4 Stateful rule compiler

This chapter provides a description of the API for the Pattern Matcher stateful rule compiler (PMSRC). The PMSRC compiles stateful rules written in a NXP stateful rule source code language into a format that is used internally within the NXP pattern matcher hardware. All the PMSRC functions run to completion, meaning a PMSRC function returns the operation performed by the function has been completed.

The PMSRC is delivered as a linkable C library called libstatefulRules.a.

PMSRC Required Files

libstatefulRules.a	Contains all the PMSRC objects.
pmsrc.h	Defines the interface to the PMSRC module
generic_types.h	Contains the generic type definitions and is included by pmsrc.h
pm_defs.h	Defines the interface that is common to PMLL, PMREC, PMSRC, etc., and is included by pmsrc.h
inttypes.h	Defines the basic integer types and is included by generic_types.h

Table continues on the next page...

Table continued from the previous page...

stdbool.h	Defines the bool type and is included by generic_types.h

7.4.6.3.4.1 pmsrc_compile

pmsrc_compile-Compile a file of stateful rules.

Synopsis

```
#include <pmsrc.h>

pmsrc_ErrorCodes_t pmsrc_compile (

 char    *input_file_name_p,

 char    *output_file_name_p,

 pmsrc_module_options_t *options_p,

 char **compile_msg_p);
```

Description

Compile stateful rule(s) into NXP hardware instructions. The user must supply valid input and output file names. The user must also supply two optional parameters (set to a valid value or set to NULL). The first is an object that contains compile options. The second is a pointer to a character pointer that allows the compiler to pass back a detailed compiler message. Both of the last two parameters may be set to NULL if default options and no compiler messages are desired. If a pointer is supplied and no compiler message is generated, the string will be set to NULL. Note that if a pointer is supplied for the compiler message, the user is responsible for freeing the string that is returned by the compiler library.

pmsrc_module_options_t is defined in pmsrc.h, listed below for easy reference.

typedef struct pmsrc_module_options

```
{
   pmsrc_debug_levels_t   debug_level;

   pmsrc_report_pad_t     report_pad;

   uint32_t               string_pad;

   pmsrc_report_cnst_sz_t report_cnst_sz;

   pmsrc_match_type_t     allow_inconclusive;

   bool                   warnings_are_errors;

   bool                   suppress_warnings;

} pmsrc_module_options_t;
```

A typical setting for the options is as follows:

```
        pmsrc_module_options_t options = {
                              pmsrc_debug_off_e,
                              pmsrc_report_pad4_e,
                              PMSRC_DEFAULT_STRING_PAD,
                              pmsrc_report_cnst_sz4_e,
                              pmsrc_conclusive_only_matches_e,
                              false,
                              false};
```

Return Value

This function returns a code of type *pmsrc_error_codes_t*. This is defined in pmsrc.h. The possible return codes are as follows:

pmsrc_ok_e

Success. Generally expect to see this code.

pmsrc_warnings_e

Compile warning found. See compiler message for details.

pmsrc_input_file_open_e

Trouble opening supplied input file.

pmsrc_output_file_open_e

Trouble opening supplied output file.

pmsrc_no_output_file_e

No output file supplied.

pmsrc_compile_failed_e

Compile failed. See compiler message for details.

pmsrc_null_list_pointer_e

Should not see this error. Indicates internal issues with data structures.

pmsrc_no_output_written_e

No output was written to output file.

pmsrc_code_string_mismatch_e

Should not see this error. Indicates incorrect number of error strings.

pmsrc_undefined_state_name_e

A state name was referred to in the stateful rule that does not exist.

pmsrc_internal_error_e

Should not see this error. Indicates an unexpected internal condition.

pmsrc_malloc_failure_e

Failure while calling malloc. System memory likely exhausted.

pmsrc_empty_stack_e

Should not see this error. Indicates issue with internal data structures.

pmsrc_name_exists_e

Rule name used more than once in file.

pmsrc_empty_input_e

Empty input file.

pmsrc_unrecognized_debug_option_e

Unrecognized debug option. Allowed options are as follows:

pmsrc_debug_off_e

pmsrc_debug_summary_e

pmsrc_debug_test_e

pmsrc_unrecognized_report_pad_option_e

Unrecognized report pad option. Allowed options are:

pmsrc_report_pad_none_e

pmsrc_report_pad4_e

pmsrc_unrecognized_string_pad_option_e

Unrecognized string pad option. Allowed values are from 0 to PMSRC_MAX_STRING_PAD_SIZE.

pmsrc_invalid_option_e

Unrecognized option.

pmsrc_string_pad_option_too_large_e

String pad option too large. Maximum defined as PMSRC_MAX_STRING_PAD_SIZE.

pmsrc_unrecognized_report_constant_size_option_e

Unrecognized report constant size. Allowed options are as follows:

pmsrc_report_cnst_sz2_e

pmsrc_report_cnst_sz4_e

pmsrc_report_cnst_sz6_e

pmsrc_report_cnst_sz8_e

pmsrc_unrecognized_allow_conclusive_option_e

Unrecognized value for allow conclusive option. Allowed values are as follows:

pmsrc_conclusive_only_matches_e

pmsrc_conclusive_and_inconclusive_matches_e

Related information

pmsrc_get_error_string on page 909

7.4.6.3.4.2 pmsrc_get_error_string

pmsrc_get_error_string-Get the string associated with a compiler error.

Synopsis

```
#include <pmsrc.h>

const char *pmsrc_get_error_string(pmsrc_error_codes_t code);
```

Description

The function returns a brief string describing the supplied error code.

Return Value

Pointer to the error string.

Related information

pmsrc_compile on page 907

7.4.6.3.5 Pattern Matcher configuration (PMC) API

This chapter provides a description of the Pattern Matcher configuration (PMC) API.

The PMC provides a programming interface to Pattern Matcher management applications for adding, removing, querying and committing regular expressions and stateful rules into Pattern Matcher hardware.

Pattern Matcher Configuration API supports:

- multiple, independant applications adding/deleting patterns and rules
- a persistent Pattern database
- Handle statistics

PMC is implemented as a user-space header file and a library. The steps involved in configuring patterns in the Pattern Matcher hardware are as follows:

instantiate PMC Daemon (PMCD) process. The PMCD is a provided as a Linux executable binary for the target SoC which includes Pattern Matcher functionality (e.g. P4080 or MPC8572).

add regular expressions from a file via a call to pmc_add_expr_file

optionally add stateful rules from a file via call to pmc_add_rule_file

commit changes to hardware via call to pmc_commit

All the PMC functions run to completion, meaning that a PMC function returns the operation performed by the function has been completed.

PMC required files

libpmc.a	Contains all the PMC objects
pmc.h	Defines the interface to the PMC module
pmcd	PMC Deamon

7.4.6.3.5.1 Status codes

```
typedef struct pmc_status {

pmc_ok_e                    = 0,
```

```
pmc_init_fail_e            = 1,

   pmc_connect_fail_e         = 2,

   pmc_send_fail_e            = 3,

   pmc_recv_fail_e            = 4,

   pmc_commit_fail_e          = 5,

   pmc_open_file_fail_e       = 6,

   pmc_record_read_fail_e     = 7,

   pmc_unsupported_pme_ver_e  = 8,

   pmc_unsupported_si_ver_e   = 9,

   pmc_incompatible_si_ver_e  = 10,

   pmc_no_memory_e            = 11,

   pmc_record_add_fail_e      = 12,

   pmc_delete_fail_e          = 13,

   pmc_query_fail_e           = 14,

   pmc_set_var_trig_fail_e    = 15,

pmc_analysis_fail_e        = 16,

} pmc_status_t;
```

7.4.6.3.5.2 Data structures

```
/* Expressions */
typedef struct pmc_expr {
          char                    *name_p;
          char                    *expr_str_p;
          char                    *option_str_p;
          uint32_t                  db_index;
          struct pmc_expr              *next_p;
} pmc_expr_t;
/* Rules */
typedef struct pmc_rule {
          char                     *name_p;
          uint32_t                  num_reactions;
          char                    **expr_names;
          uint32_t                  db_index;
} pmc_rule_t;
/* Statistics */
typedef struct pmc_stats {
/* ====================================================================
 *                Pattern Matcher hardware stats
```

```
 * ========================================================================*/
/* The number of input bytes reported by KES */
uint64_t  pm_input_bytes;
/* The number of output report bytes reported by SRE */
uint64_t  pm_output_bytes;
/* The number of one byte trigger hits reported by KES */
uint64_t  pm_trigger_one_byte_hits;
/* The number of two byte trigger hits reported by KES */
uint64_t  pm_trigger_two_byte_hits;
/* The number of variable byte trigger hits reported by KES */
uint64_t  pm_trigger_variable_hits;
/* The number of special trigger hits reported by KES */
uint64_t  pm_trigger_special_hits;
/* The number of confidence stage hits reported by KES */
uint64_t  pm_confidence_hits;
/* The number of matches reported by DXE */
uint64_t  pm_matches;
/* The number of SR executions triggered by DXE reported by SRE */
uint64_t  pm_dxe_executions;
/* The number of SR executions triggered by End of SUI reported by SRE */
uint64_t  pm_end_of_sui_executions;
/* The number SUIs with matches reported by DXE */
uint64_t  pm_sui_matching_patterns;
/* The number of SUIs that generate reports reported by SRE */
uint64_t  pm_sui_generating_reports;
/* The number of input SUIs reported by KES */
uint64_t  pm_input_suis;
/* The number of matches with DRCC reported by DXE */
uint64_t  pm_selected_matches;
/* ========================================================================
 *                Deflate hardware stats
 * ========================================================================*/
/* The number of deflate input bytes reported by DFL */
uint64_t  df_input_bytes;
/* The number of deflate output bytes reported by DFL */
uint64_t  df_output_bytes;
/* The number of deflate work units reported by DFL */
uint64_t  df_decompressions;
/* ========================================================================
 *                Linker Loader software stats
 * ========================================================================*/
/* ------------------------------------------------------------
 *                DXE/SRE table statistics.
 * ------------------------------------------------------------*/
/* The total number of the DXE/SRE entries present in the DXE/SRE
 * table, i.e., this is the DXE/SRE table size.  This value is
 * specified in a call to the pmll_db_create() function. */
uint32_t  dxeSreEntryNum;

/* The number of the base entries in the DXE/SRE table.  For a given
 * release of PMLL this number is constant.  This number is also the
 * minimum size of the DXE/SRE table. */
uint32_t  dxeSreBaseEntryNum;

/* The number of the DXE/SRE extension entries, i.e., the number of
 * the "non-base" entries in the DXE/SRE table. */
uint32_t  dxeSreExtensionEntryNum;

/* The number of the currently allocated extension entries. */
uint32_t  dxeSreAllocatedExtensionEntryNum;
```

```
        /* The number of the currently available extension entries. */
        uint32_t  dxeSreAvailableExtensionEntryNum;
        /* ------------------------------------------------------
         *                 SRE session statistics.
         * ------------------------------------------------------*/
        /* The number of the SRE sessions.  This value is specified in a
         * call to the pmll_db_create() function. */
        uint32_t  sreSessionCtxNum;

        /* The size (in bytes) of each of the SRE sessions.  This value is
         * specified in a call to the pmll_db_create() function. */
        uint32_t  sreSessionCtxSize;

        /* The size (in bytes) of the session digest area.  This value
         * depends on the number of SRE rules as specified by the user in
         * a call to the pmll_db_create() function. */
        uint32_t  sreSessionDigestSize;

        /* The size (in bytes) of the session flags area.  This value is
         * constant for a given release of PMLL. */
        uint32_t  sreSessionFlagsSize;

        /* The size (in bytes) of the session context areas. */
        uint32_t  sreSessionCtxAreaSize;

        /* The size (in bytes) of the currently allocated session context areas. */
        uint32_t  sreAllocatedSessionCtxAreaSize;

        /* The size (in bytes) of the currently available session context areas. */
        uint32_t  sreAvailableSessionCtxAreaSize;
        /* ------------------------------------------------------------
         *             Expression and pattern statistics.
         * ------------------------------------------------------------*/
        /* The maximum number of patterns that can be configured.  This value is
         * constant for a given release of PMLL. */
        uint32_t patternMaxNum;

        /* The number of the currently configured expressions. */
        uint32_t  expNum;

        /* The number of the currently configured "special" patterns. */
        uint32_t  specialPatternNum;

        /* The number of the currently configured "one-byte" patterns. */
        uint32_t  oneBytePatternNum;

        /* The number of the currently configured "two-byte" patterns. */
        uint32_t  twoBytePatternNum;

        /* The number of the currently configured "variable" patterns. */
        uint32_t  variablePatternNum;

        /* The total number of the currently configured patterns. */
        uint32_t  totalPatternNum;

        /* The currently configured variable trigger size. */
        uint32_t  variableTriggerSize;

        /* ------------------------------------------------------------
```

```
 *               Rule and reaction statistics.
 * ------------------------------------------------------------*/
/* The maximum number of stateful rules.  This value is specified in
 * a call to the pmll_db_create() function. */
uint32_t  statefulRuleMaxNum;

/* The maximum number of stateless rules.  This value is constant
 * for a given release of PMLL. */
uint32_t  statelessRuleMaxNum;

/* The maximum number of all rules.  This value is constant for a
 * given release of PMLL. */
uint32_t  totalRuleMaxNum;

/* The number of the currently configured stateless rules. */
uint32_t  statelessRuleNum;

/* The number of the currently configured stateful rules. */
 uint32_t  statefulRuleNum;

/* The number of the currently configured rules. */
uint32_t  totalRuleNum;

/* The number of the currently configured end-of-SUI reactions. */
uint32_t  endOfSuiReactionNum;
}
```

7.4.6.3.5.2.1 Add regular expressions from compiled bin file

Adds all regular expressions in the precompiled binary file. Option to allow auto commit.

```
pmc_status_t pmc_add_expr_file(                        char *filename_p,
                                        bool auto_commit,
                                        char **info_msg_p)
```

7.4.6.3.5.2.2 Add stateful rules from compiled bin file

Adds all stateful rules in the precompiled binary file. Option to allow auto commit.

```
pmc_status_t pmc_add_rule_file(                        char *filename_p,
                                       bool auto_commit,
                                       char **info_msg_p)
```

7.4.6.3.5.2.3 Delete regular expression by name or set and subset

Deletes the regular expression referenced by name or set and subset. Option to allow auto commit.

```
pmc_status_t pmc_del_expr_name(                    char *name_p,
                                       bool auto_commit,
                                       char **info_msg_p)
pmc_status_t pmc_del_expr_set(                         int set,
                                       int subset,
                                       bool auto_commit,
                                       char **info_msg_p)
```

7.4.6.3.5.2.4 Delete regular expressions by compiled bin file

Deletes all regular expressions in the precompiled binary file. Option to allow auto commit.

```
pmc_status_t pmc_del_expr_file(                                 char *filename_p,
                                                        bool auto_commit,
                                                        char **info_msg_p)
```

7.4.6.3.5.2.5 Delete stateful rule by name

Deletes the stateful rule referenced by name. Option to allow auto commit.

```
pmc_status_t pmc_del_rule_name(                           char *name_p,
                                                        bool auto_commit,
                                                        char **info_msg_p)
```

7.4.6.3.5.2.6 Delete stateful rule by compiled bin file

Deletes all stateful rules in the precompiled binary file. Option to allow auto commit.

```
pmc_status_t pmc_del_rule_file(                                 char *filename_p,
                                                        bool auto_commit,
                                                        char **info_msg_p)
```

7.4.6.3.5.2.7 Commit changes to hardware

Commit all changes to hardware.

```
pmc_status_t pmc_commit(char **info_msg_p)
```

7.4.6.3.5.2.8 Delete all regular expressions and rules

Delete all expressions or all rules or both. Option to allow auto commit.

```
pmc_status_t pmc_del_all_expr(                              bool auto_commit,
                                                        char **info_msg_p)


pmc_status_t pmc_del_all_rules(                              bool auto_commit,
                                                        char **info_msg_p)
pmc_status_t pmc_del_all(                                 bool auto_commit,
                                                        char **info_msg_p)
```

7.4.6.3.5.2.9 Query regular expression by name or set and subset

Query information on a single named regular expression or a list of expressions based on set and subset. The information is returned in a single pmc_expr_t record or a linked list of pmc_expr_t records. See pmc_free_expr to cleanup.

```
pmc_status_t pmc_query_expr_name(                     char *name_p,
                                                        pmc_expr_t **expr_p,
                                                        char **info_msg_p)
pmc_status_t pmc_query_expr_set(                                int set,
                                                        int subset,
                                                        pmc_expr_t **expr_p,
                                                        char **info_msg_p)
```

7.4.6.3.5.2.10 Query stateful rule by name

Query information on a single named stateful rule. The information is returned in a pmc_rule_t record. See pmc_free_rule to cleanup.

```
pmc_status_t pmc_query_rule_name(                              char *name_p,
                                                    pmc_rule_t **rule_p,
                                                    char **info_msg_p)
```

7.4.6.3.5.2.11 Query first regular expression in the database

Query the first regular expression in the database. The index of the expression is stored in the expression object. See pmc_free_expr to cleanup.

```
pmc_status_t pmc_query_expr_first( pmc_expr_t **expr_p,
                                                    char **info_msg_p)
```

7.4.6.3.5.2.12 Query next regular expression in the database

Query the next regular expression in the database. This function will return the next expression based on the index of the current expression that is passed in. See pmc_free_expr to cleanup.

```
pmc_status_t pmc_query_expr_next (                               uint32_t       index,
                                                    pmc_expr_t **next_expr_p,
                                                    char **info_msg_p)
```

7.4.6.3.5.2.13 Query first stateful rule in the database

Query the first stateful rule in the database. The index of the rule is stored in the rule object. See pmc_free_rule to cleanup.

```
pmc_status_t
pmc_query_rule_first(                                                    pmc_rule_t **rule_p,
                                                                               char
**info_msg_p)
```

7.4.6.3.5.2.14 Query next stateful rule in the database

Query the next stateful rule in the database. This function will return the next rule based on the index of the current rule that is passed in. See pmc_free_rule to cleanup.

```
pmc_status_t pmc_query_rule_next(                              uint32_t index,
                                                    pmc_rule_t **next_rule_p,
                                                    char **info_msg_p)
```

7.4.6.3.5.2.15 Free expression and rule objects after queries

Free the expression and rule objects returned by the various queries.

```
void pmc_free_expr(pmc_expr_t *expr_p);
void pmc_free_rule(pmc_rule_t *rule_p);
```

7.4.6.3.5.2.16 Query statistics

Query PM statistics. The information is returned in a pmc_stats_t record.

```
pmc_status_t pmc_query_stats(                                pmc_stats_t *stats_p,
                                                  char **info_msg_p)
```

7.4.6.3.5.2.17 Reset statistics

Reset PM statistics.

```
pmc_status_t pmc_reset_stats()
```

7.4.6.3.5.2.18 Set variable length trigger

Set the size of the variable length trigger. This may only be used when the database is empty.

```
pmc_status_t pmc_set_var_trig_size(                        uint32_t size,
                                           char **info_msg_p)
```

7.4.6.3.5.2.19 Expression analysis

Run the expression analysis tool. Be advised this can take in the order of minutes to complete depending on the number of expressions in the database.

```
pmc_status_t pmc_analysis(                  char **log_str_p,
                                 char **info_msg_p);
```

7.4.6.3.6 Linker-loader

This chapter provides a description of the API of the Pattern Matcher linker-loader (PMLL). The linker-loader accepts expression and rule records encoded in the NXP hardware specific format, links these records into a hardware-optimal image-also called the shadow database (shadow DB)-and loads this image into the PM hardware.

PMLL is implemented as a user-space library called libpmll.a. The generic steps involved in configuring the Pattern Matcher hardware are as follows:

- Initialize the PMLL module with a call to the pmll_module_init() function
- Call the pmll_db_create() function to create a PMLL shadow DB and register the PMLA functions with the new shadow DB
- Use the pmll_connection_handle_set() function to associate a PMLA channel with the created PMLL shadow DB
- Optionally reconfigure the equivalence table and the variable trigger length attributes using the pmll_equivalence_table_set() and pmll_variable_trigger_size_set() functions
- Use the PMLL expression functions to add expressions to the shadow DB
- Use the PMLL rule functions to add rules to the shadow DB
- Call the pmll_commit() function to get PMLL to link the expression and rule records and to load the configuration on the PM hardware

All the PMLL functions run to completion, meaning that a PMLL function returns the operation performed by the function has been completed.

PMLL Files

File	Description
libpmll.a	Contains all the PMLL objects
pmll.h	Defines the interface to the PMLL moduleT
generic_types.h	Contains the generic type definitions and is included by pmll.h
pm_defs.h	Defines the interface that is common to the PMLL, PMREC, PMSRC, etc., and is included by pmll.h
pmp.h	Contains the definition of the Pattern Matcher Protocol and is included by pmll.h
libpthread.a	Defines the pthread objects; used by some of the PMLL objects
byteswap.h, endian.h	Provide support for little and big-endian byte swapping and are included by pmp.h
inttypes.h	Define the basic integer types and is included by generic_types.h
stdbool.h	Defines the bool type and is included by generic_types.h
netinit/in.h	Provides support for the hton and ntoh macros and is included by pmp.h

7.4.6.3.6.1 pmll return codes

Most of the PMLL functions return status codes. The different return codes are defined in the pmll_status_t enumerated type and are listed below.

```
pmll_ok_e
```

This is the generic success code.

```
pmll_error_e
```

This is the generic error code. It is often used as an initial value for the return status variable.

```
pmll_too_few_error_strings_e
```

More error codes than strings are defined.

```
pmll_too_many_error_strings_e
```

More error strings than codes are defined.

```
pmll_unsupported_record_version_e
```

The record version passed in is not supported.

```
pmll_null_pointer_parameter_e
```

Unexpected NULL pointer passed in as a parameter.

```
pmll_out_of_memory_e
```

Memory allocation failed - no more memory.

```
pmll_invalid_name_e
```

The passed in expression or rule name is invalid, for example, has zero length.

```
pmll_exp_name_is_in_use_e
```

An expression with the specified expression name exists.

```
pmll_exp_is_not_in_name_db_e
```

An expression with the specified expression name does not exist.

```
pmll_no_trigger_type_e
```

Could not determine the trigger type for a pattern.

```
pmll_unsupported_trigger_type_e
```

The trigger type found in the expression record is not supported.

```
pmll_too_few_trig_row_format_defs_e
```

The number of the row trigger formats is too small.

```
pmll_too_many_trig_row_format_defs_e
```

The number of the row trigger formats is too big.

```
pmll_failed_to_commit_pattern_e
```

Failed to commit a pattern.

```
pmll_entry_write_failed_e
```

Failed to write a table entry to the PM H/W.

```
pmll_bad_confirmation_rec_size_e
```

The size of the confirmation record structure is not as expected.

```
pmll_module_not_initialized_e
```

The PMLL module has not been initialized.

```
pmll_bad_trigger_rec_size_e
```

The size of the trigger record structure is not as expected.

```
pmll_invalid_parameter_value_e
```

The value of a parameter passed to the function is invalid.

```
pmll_too_many_one_byte_patterns_e
```

Adding the pattern would exceed the "1-byte" pattern limit.

```
pmll_too_many_two_byte_patterns_e
```

Adding the pattern would exceed the "2-byte" pattern limit.

```
pmll_too_many_variable_patterns_e
```

Adding the pattern would exceed the "variable" pattern limit.

```
pmll_too_many_special_patterns_e
```

Adding the pattern would exceed the "special" pattern limit.

```
pmll_too_many_patterns_e
```

Adding the pattern would exceed the global pattern limit.

```
pmll_invalid_db_handle_e
```

The passed in PMLL DB handle is invalid.

```
pmll_table_index_too_big_e
```

The index table passed is too big.

```
pmll_pattern_not_in_confid_table_e
```

A pattern could not be found in the confidence table.

```
pmll_bad_logic_e
```

Bad logic in function.

```
pmll_too_many_reactions_in_rule_e
```

Found too many reactions in a rule.

```
pmll_no_reactions_in_rule_e
```

Found no reactions in a rule.

```
pmll_reaction_too_big_e
```

Reaction size is too big.

```
pmll_null_reaction_pointer_in_rule_e
```

Found a NULL reaction pointer in a rule.

```
pmll_rule_name_is_in_use_e
```

A rule with the specified rule name exists.

```
pmll_too_many_rules_e
```

Adding the rule would exceed the rule limit.

```
pmll_rule_is_not_in_name_db_e
```

A rule with the specified rule name does not exist.

```
pmll_exp_is_used_in_rule_e
```

An expression is used in a rule.

```
pmll_failed_to_init_mutex_e
```

Initialization of a mutex failed.

```
pmll_failed_to_destroy_mutex_e
```

Destruction of a mutex failed.

```
pmll_no_more_records_e
```

There are no more records in PMLL DB.

```
pmll_failed_to_create_name_db_e
```

Failed to create a name DB.

```
pmll_failed_to_add_exp_to_name_db_e
```

Failed to add an expression record to the PMLL expression name DB.

```
pmll_failed_to_add_rule_to_name_db_e
```

Failed to add a rule record to the PMLL rule name DB.

```
pmll_index_is_not_in_use_e
```

The index is not in use.

```
pmll_failed_to_create_reset_table_e
```

Failed to create the PMLL DB reset table.

```
pmll_failed_to_create_block_table_e
```

Failed to create the PMLL DB block table.

```
pmll_failed_to_create_cli_menu_e
```

Failed to create a CLI menu.

```
pmll_failed_to_register_cli_command_e
```

Failed to register a CLI command.

```
pmll_too_few_confirmation_blocks_e
```

The number of the confirmation blocks is too small.

```
pmll_out_of_extension_blocks_e
```

Run out of the extension blocks.

```
pmll_failed_to_create_rule_id_table_e
```

Failed to create the PMLL DB rule ID table.

```
pmll_out_of_rule_ids_e
```

Run out of rule Ids.

```
pmll_reaction_too_small_e
```

Size of a reaction is too small.

```
pmll_misaligned_reaction_e
```

Instructions in the reaction are misaligned.

```
pmll_failed_to_create_rule_ctx_table_e
```

Failed to create the PMLL DB rule context table.

```
pmll_session_ctx_area_too_small_e
```

Session context area size is smaller than the minimum size required by PMLL.

```
pmll_failed_to_alloc_rule_ctx_area_e
```

Failed to allocate a rule context area.

```
pmll_vtrigger_size_out_of_range_e
```

Variable trigger size is out of range.

```
pmll_exps_are_present_in_db_e
```

Expressions are present in the PMLL DB.

```
pmll_bad_reaction_event_type_e
```

Unsupported reaction event type.

```
pmll_bad_session_ctx_area_size_e
```

Session context area size is not a multiple of a session context entry.

```
pmll_table_reset_failed_e
```

Failed to reset entries in PM H/W tables.

```
pmll_pmhi_init_failed_e
```

Failed to initialize the PMLL PM H/W I/O module.

```
pmll_failed_to_set_atomic_attr_e
```

Failed to set the PMP atomic attribute.

```
pmll_failed_to_set_vtrig_size_attr_e
```

Failed to set the PMP variable trigger size attribute.

```
pmll_failed_to_set_end_of_sui_attr_e
```

Failed to set the PMP end-of-SUI attribute.

```
pmll_flush_failed_e
```

The PMLA flush operation failed.

```
pmll_failedto_reset_rule_ctxs_ee
```

Failed to reset a rule context.

```
pmll_failed_to_set_batch_attr_e
```

Failed to set the batch attribute.

```
pmll_pmla_channel_is_down_e
```

PMLA channel is down.

```
pmll_pmla_channel_is_not_set_e
```

PMLA channel is not set.

```
pmll_handle_table_create_failure_e
```

Failed to create the PMLL handle table.

```
pmll_handle_table_destroy_failure_e
```

Failed to destroy the PMLL handle table.

```
pmll_failed_to_allocate_handle_e
```

Failed to allocate a PMLL handle.

```
pmll_too_many_stateful_rules_req_e
```

The requested maximum number of stateful rules is too big.

```
pmll_cannot_register_null_pmla_func_e
```

A NULL pointer cannot be registered as a PMLA function pointer.

7.4.6.3.6.2 pmll_api_version_get

pmll_api_version_get-Retrieve the version of the PMLL API.

Synopsis

```
#include <pmll.h>
uint32_t pmll_api_version_get(void);
```

Description

The pmll_api_version_get() function retrieves the version of the API of the PMLL library being used.

Return Value

The pmll_api_version_get() function returns the version of the API of the PMLL library being used.

7.4.6.3.6.3 pmll_commit

pmll_commit - Commit the PMLL shadow DB to the PM hardware.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_commit(
unsigned int pmllDbHandle,
char         *expName_p);
```

Description

The pmll_commit() function triggers all the PMLL commit operations. The current version of the pmll_commit() function implements both the linking and the loading functions of PMLL. Once the pmll_commit() function returns, the optimized image of the PMLL shadow DB with the pmllDbHandle handle has been written to the PM hardware and the PM hardware is ready to scan data for the configured expressions and rules.

When the pmll_commit() function fails to link the patterns present in the PMLL shadow DB and returns with the pmll_failed_to_commit_pattern_e error code, the name of the expression containing the pattern that caused the PMLL linking failure is returned through the expName_p parameter.

Return Value

The pmll_commit() function returns pmll_ok_e upon success or an error code upon a failure. The pmll_commit() function can return the following error codes:

```
pmll_bad_logic_e
pmll_entry_write_failed_e
pmll_entry_write_failed_e
pmll_failed_to_commit_pattern_e
pmll_failed_to_reset_rule_ctxs_e
pmll_failed_to_set_atomic_attr_e
pmll_failed_to_set_batch_attr_e
pmll_failed_to_set_end_of_sui_attr_e
pmll_failed_to_set_vtrig_size_attr_e
pmll_flush_failed_e
pmll_invalid_db_handle_e
pmll_module_not_initialized_e
pmll_null_pointer_parameter_e
pmll_out_of_memory_e
pmll_pmla_channel_is_down_e
pmll_pmla_channel_is_not_set_e
pmll_table_reset_failed_e
```

When the pmll_failed_to_commit_pattern_e error code is returned the name of the expression containing the pattern that caused the PMLL linking failure is returned through the expName_p parameter. At present this failure is critical. There is no

recovery mechanisms built into the pmll_commit() function and most of the time this failure results in corrupting the PMLL shadow DB. The user must rebuild the PMLL shadow DB from scratch.

7.4.6.3.6.4 pmll_connection_handle_set, pmll_connection_handle_get

pmll_connection_handle_set, pmll_connection_handle_get-Set, get the PMLA connection handle.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_connection_handle_set(

unsigned int pmllDbHandle,

handle_t pmlaHandle);
```

pmll_status_t pmll_connection_handle_get(

```

unsigned int pmllDbHandle,

handle_t *pmlaHandle_p);
```

Description

pmll_connection_handle_set() associates a PMLA connection with the PMLL shadow DB with the pmllDbHandle handle. The user specified pmlaHandle is used by PMLL to refer to the set of PMLA functions registered previously with PMLL when the PM shadow DB was created (using pmll_db_create). The pmlaHandle is needed as an input parameter when calling PMLA functions.

pmll_connection_handle_get() retrieves the PMLA connection handle that is currently associated with the PMLL shadow DB with the pmllDbHandle handle. The retrieved PMLA connection handle is returned through the pmlaHandle_p parameter.

Return Value

The pmll_connection_handle_set() and pmll_connection_handle_get() functions return pmll_ok_e upon success or an error code upon a failure.

The pmll_connection_handle_set() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
```

The pmll_connection_handle_get() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

7.4.6.3.6.5 pmll_db_create, pmll_db_destroy

pmll_db_create, pmll_db_destroy-Create, destroy a PMLL shadow DB.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_db_create(
     pmll_db_params_t
*pmllDbParams_p,
     unsigned int
```

```
*pmllDbHandle_p);
pmll_status_t pmll_db_destroy(unsigned int pmllDbHandle);
```

Description

pmll_db_create() creates a new PMLL shadow DB. Upon successful completion the function returns a non-negative integer handle that was assigned to the created PMLL shadow DB. PMLL API functions that operate on a specific shadow DB take this handle as an input parameter. The handle is returned through the pmllDbHandle_p parameter.

The pmll_db_params_t type of the pmllDbParams_p parameter is defined as follows:

```
typedef int pmll_pmla_bulk_begin_function_t(
     handle_t                                  pmlaHandle);
typedef int pmll_pmla_bulk_end_function_t(
     handle_t                                  pmlaHandle);
typedef int pmll_pmla_flush_function_t(
     handle_t                                  pmlaHandle);
typedef int pmll_pmla_read_function_t(
     handle_t                                  pmlaHandle,
     pmp_msg_t                                 *msg_p);
typedef int pmll_pmla_write_function_t(
     handle_t                                  pmlaHandle,
     pmp_msg_t                                 *msg_p);
typedef const char *pmll_pmla_error_string_get_t(
     int                                       errorCode);
typedef struct {
     pmll_pmla_bulk_begin_function_t
          *pmlaBulkBeginFunction_p;
     pmll_pmla_bulk_end_function_t
          *pmlaBulkEndFunction_p;
     pmll_pmla_flush_function_t
          *pmlaFlushFunction_p;
     pmll_pmla_read_function_t
          *pmlaReadFunction_p;
     pmll_pmla_write_function_t
          *pmlaWriteFunction_p;
     pmll_pmla_error_string_get_t
          *pmlaErrorStringGetFunction_p;
} pmll_pmla_functions_t;
typedef struct {
     pmll_pmla_functions_t
pmlaFunctions;
     pmp_extension_block_num_attr_t

dxeSreTableSize;
     pmp_context_area_size_attr_t
sreSessionCtxSize;
     pmp_context_max_num_attr_t
sreSessionCtxNum;
     pmp_max_stateful_rule_num_attr_t
                                                                              sreRuleNum;
} pmll_db_params_t;
```

The pmlaFunctions parameter contains pointers to the PMLA functions that must be registered with the PMLL shadow DB. All the functions must be specified. If any of the PMLA function pointers are NULL, then the pmll_db_create() function will fail with the pmll_cannot_register_null_pmla_func_e error code. Refer to the "PMLL PMLA Functions" section later in this chapter for more details on the PMLA functions.

The dxeSreTableSize, sreSessionCtxSize, sreSessionCtxNum and sreRuleNum parameters indicate PM hardware settings. These settings are specified when loading PM hardware drivers and can be queried. The dxeSreTableSize parameter

indicates the total number of confirmation entries, including both the pattern confirmation entries and the reaction extension entries available to the PM hardware. The minimum value of entries is 74240 and the maximum value is 1048576. The sreSessionCtxSize parameter indicates the size of the PM hardware context table entry associated with a single session. The value of sreSessionCtxSize must be between 64 and 131072. The sreSessionCtxNum parameter indicates the number of PM sessions supported by the PM hardware. The sreRuleNum parameter indicates the maximum number of stateful rules that can be configured in the PM hardware. The value of sreRuleNum must be between 0 and 32768. PMLL relies on the settings provided by the user to match the PM hardware settings. It does not query the hardware to sanitize the values.

Each PMLL shadow DB has its own mutex to protect accesses to the PMLL shadow DB. This means that the developers using the PMLL library do not have to create external mutual exclusion mechanisms to protect the integrity of the PMLL shadow DB.

pmll_db_destroy() destroys the PMLL shadow DB with the pmllDbHandle handle. The function removes all the rules and expressions from the DB and frees all the memory that was used by the PMLL shadow DB.

Return Value

Both the pmll_db_create() and pmll_db_destroy functions return pmll_ok_e upon success or an error code upon a failure.

The pmll_db_create() function can return the following error codes:

```
     pmll_bad_session_ctx_area_size_e
     pmll_cannot_register_null_pmla_func_e
     pmll_failed_to_allocate_handle_e
     pmll_failed_to_create_block_table_e
     pmll_failed_to_create_name_db_e
     pmll_failed_to_create_reset_table_e
     pmll_failed_to_create_rule_ctx_table_e
     pmll_failed_to_create_rule_id_table_e
     pmll_failed_to_init_mutex_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
     pmll_out_of_memory_e
     pmll_session_ctx_area_too_small_e
     pmll_too_few_confirmation_blocks_e
     pmll_too_many_stateful_rules_req_e
```

The pmll_db_destroy() function can return the following error codes:

```
     pmll_failed_to_destroy_mutex_e
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
```

7.4.6.3.6.6 pmll_equivalence_table_set, pmll_equivalence_table_get

pmll_equivalence_table_set, EquivalenceTableGet-Set, get the equivalence table.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_equivalence_table_set(
     unsigned int
pmllDbHandle,
     const uint8_t
*equivalenceTable_p);
```

pmll_status_t pmll_equivalence_table_get(

```
     unsigned int
pmllDbHandle,
```

```
     uint8_t
*equivalenceTable_p);
```

Description

pmll_equivalence_table_set() can be used to configure new values in the equivalence table associated with the PMLL shadow DB with the pmllDbHandle handle. The new values of the equivalence table are passed to the function through the equivalenceTable_p parameter. A default equivalence table is defined in pm_defs.h. Note that the pmll_equivalence_table_set() function will fail if it is called when there are expressions present in the PMLL shadow DB.

pmll_equivalence_table_get() retrieves the values in the equivalence table associated with the PMLL shadow DB with handle pmllDbHandle.

The size of the buffer pointed to by equivalenceTable_p must be at least PMP_EQUIVALENCE_ENTRY_SIZE bytes.

Return Value

The pmll_equivalence_table_set() and pmll_equivalence_table_get() functions return pmll_ok_e upon success or an error code upon a failure.

The pmll_equivalence_table_set() function can return the following error codes:

```
     pmll_exps_are_present_in_db_e
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

The pmll_equivalence_table_get() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

7.4.6.3.6.7 pmll_error_string_get

pmll_error_string_get-Get the error string associated with a PMLL error code.

Synopsis

```
#include <pmll.h>
const char *pmll_error_string_get(pmll_status_t errorCode);
```

Description

The pmll_error_string_get() function returns a concise string describing the error condition associated with errorCode.

Return Value

The pmll_error_string_get() function returns a concise string describing the error associated with errorCode.

7.4.6.3.6.8 pmll expression functions

pmll_exp_add-Add an expression to a PMLL shadow DB.

pmll_exp_delete-Delete an expression from a PMLL shadow DB.

pmll_exp_all_delete-Delete all expressions from a PMLL shadow DB.

pmll_exp_index_next_get-Get the index of the next PMLL expression.

pmll_exp_name_in_use-Check if an expression name is in use.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_exp_add(
     unsigned int
pmllDbHandle,
     uint32_t
recordVersion,
     const
pm_exp_record_t
*expRecord_p,
     uint32_t
*index_p);
```

pmll_status_t pmll_exp_delete(

```
     unsigned int
pmllDbHandle,
     uint32_t
idx);
```

pmll_status_t pmll_exp_all_delete(

```
     unsigned int
pmllDbHandle);
```

pmll_status_t pmll_exp_index_next_get(

```
     unsigned int
pmllDbHandle,
     uint32_t                                                                    idx,
     uint32_t
*nextIndex_p);
```

pmll_status_t pmll_exp_name_in_use(

```
     unsigned int
pmllDbHandle,
     const char
*name_p,
     bool
*nameIsInUse_p,
     uint32_t
*idx_p);
```

Description

pmll_exp_add() adds a single PMLL expression to a PMLL shadow DB with the pmllDbHandle handle. The function uses the index_p parameter to return an index that is assigned to the expression by PMLL. Other PMLL expression functions take this index as an input parameter that identifies the expression. The recordVersion parameter indicates the version of the expression record specified by expRecord_p. The supported version for Pattern Matcher 1.1 is PMLL_EXP_RECORD_V_1_0_0, which is defined in pmll.h.

The expRecord_p parameter is a pointer to the pm_exp_record_t structure, which is defined in pm_defs.h. The pm_exp_record_v_1_0_0_t type below defines the version 1.0.0 of the PMLL expression record. The user needs to create this record and cast it to the generic PMLL expression record type (pm_exp_record_t) and use this generic type in the PMLL functions.

typedef struct pm_pattern_record_t {

```
     uint8_t
        triggerEntry[PM_TRIGGER_ENTRY_SIZE];
     uint8_t
        keyElementEntry[PM_KEY_ELEMENT_ENTRY_SIZE];
     uint8_t
        confirmationEntry[PM_CONFIRMATION_ENTRY_SIZE];
     struct pm_pattern_record_t
        *nextPatternRecord_p;
} pm_pattern_record_t;
```

typedef struct {

```
     char
name_s[PM_NAME_MAX_LENGTH];
     pm_pattern_record_t
patterns;
} pm_exp_record_v_1_0_0_t;
```

typedef uint8_t *pm_exp_record_t;

pmll_exp_delete() deletes the expression with idx index from the PMLL shadow DB specified by pmllDbHandle.

pmll_exp_all_delete() deletes all the expressions from the PMLL shadow DB specified by pmllDbHandle.

pmll_exp_index_next_get() retrieves the next valid expression index after the idx index in the PMLL shadow DB specified by pmllDbHandle. The retrieved index is returned through the nextIndex_p parameter. To get the first valid index the idx parameter must be set to the PMLL_NULL_INDEX value. The pmll_exp_index_next_get() function returns pmll_no_more_records_e when there are no valid expression records with index values greater than the passed in idx index value. The pmll_exp_index_next_get() function allows the user to "walk" through all the expressions in the PMLL shadow DB in an efficient way.

pmll_exp_name_in_use() checks if the name_p expression name is already in use in the PMLL shadow DB specified by pmllDbHandle. This is used to verify that a name is not already in use before attempting to add it to the PMLL shadow DB. If the name_p expression name is in use, the nameIsInUse_p parameter is set to true and the idx_p parameter is set to the index of the expression that uses name_p. If the name_p is not in use, then nameIsInUse_p is set to false.

Return Value

All the PMLL expression functions returns pmll_ok_e upon success or an error code upon a failure.

The pmll_exp_add() function can return the following error codes:

```
     pmll_exp_name_is_in_use_e
     pmll_failed_to_add_exp_to_name_db_e
     pmll_invalid_db_handle_e
     pmll_invalid_name_e
     pmll_module_not_initialized_e
     pmll_no_trigger_type_e
     pmll_null_pointer_parameter_e
     pmll_out_of_extension_blocks_e
     pmll_out_of_memory_e
     pmll_too_many_one_byte_patterns_e
     pmll_too_many_patterns_e
     pmll_too_many_patterns_e
     pmll_too_many_special_patterns_e
     pmll_too_many_two_byte_patterns_e
     pmll_too_many_variable_patterns_e
     pmll_unsupported_record_version_e
     pmll_unsupported_trigger_type_e
```

The pmll_exp_delete() function can return the following error codes:

```
     pmll_exp_is_not_in_name_db_e
     pmll_exp_is_used_in_rule_e
     pmll_index_is_not_in_use_e
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
```

The pmll_exp_all_delete() function can return the following error codes:

```
     pmll_exp_is_not_in_name_db_e
     pmll_exp_is_used_in_rule_e
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
```

The pmll_exp_index_next_get() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_no_more_records_e
     pmll_null_pointer_parameter_e
```

The pmll_exp_name_in_use() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

See Also

pmll_supported_exp_record_versions_get(), pmll_supported_exp_record_versions_num_get()

7.4.6.3.6.9 pmll_module_init, pmll_module_shutdown

pmll_module_init-Initialize the PMLL module.

pmll_module_shutdown-Shuts down the PMLL module.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_module_init(
     unsigned int
initialDbHandleNum,
     bool
expandDbHandleNumFlag);
pmll_status_t pmll_module_shutdown(void);
```

Description

pmll_module_init() initializes the PMLL module. No other PMLL API function will work properly until the pmll_module_init() function has been called. The initialDbHandleNum parameter indicates the initial number of the PMLL shadow DB handles to be created. The expandDbHandleNumFlag flag indicates if the pool of PMLL shadow DB handles can be expanded or not when there are no more handles available in the pool. In order to allow the dynamic expansion of the PMLL shadow DB handle pool the expandDbHandleNumFlag flag must be set to true. If it is set to false, dynamic expansion of the PMLL shadow DB handle pool will not be allowed.

pmll_module_shutdown() destroys the global resources used by the PMLL module and shuts down the PMLL module.

Return Value

The pmll_module_init() and pmll_module_shutdown() functions return pmll_ok_e upon success or an error code upon failure.

The pmll_module_init() function can return the following error codes:

```
     pmll_bad_confirmation_rec_size_e
     pmll_bad_trigger_rec_size_e
     pmll_handle_table_create_failure_e
     pmll_pmhi_init_failed_e
     pmll_too_few_error_strings_e
     pmll_too_few_trig_row_format_defs_e
     pmll_too_many_error_strings_e
     pmll_too_many_trig_row_format_defs_e
```

The pmll_module_shutdown() function can return the following error codes:

```
     pmll_handle_table_destroy_failure_e
```

7.4.6.3.6.10 pmll pmla functions

pmll_pmla_functions_set-Set the PMLA functions.

pmll_pmla_functions_get-Retrieve the PMLA functions.

pmll_pmla_bulk_begin-Indicate the beginning of a bulk operation.

pmll_pmla_bulk_end-Indicate the end of a bulk operation.

pmll_pmla_error_string_get-Get error string describing an error code.

pmll_pmla_flush-Flush messages in the PMLA connection.

pmll_pmla_read-Read a PMP message.

pmll_pmla_write-Write a PMP message.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_pmla_functions_get(
     unsigned int
pmllDbHandle,

pmll_pmla_functions_t
*functions_p);
```

pmll_status_t pmll_pmla_functions_set(

```
     unsigned int
pmllDbHandle,

pmll_pmla_functions_t
*functions_p);
```

Description

PMLA is responsible for implementing a communication channel to and from the PMCI module and for conveying PMP control messages across that communication channel. Users can develop and use their own versions of PMLA functions. This allows the communication channel to the PMCI module to be defined by users according to their requirements. Sample PMLA functions are delivered with the PM software. They are implemented in pmla_slm_api.c.

The PMLL module uses the following PMLA functions:

pmll_pmla_bulk_begin(), pmll_pmla_bulk_end(), pmll_pmla_error_string_get(), pmll_pmla_flush(), pmll_pmla_read() and pmll_pmla_write() functions.

The PMLA functions called by PMLL are registered with PMLL through a call to the pmll_db_create() function. The PMLA functions associated with a PMLL shadow DB can subsequently be changed using the pmll_pmla_functions_set() and pmll_pmla_functions_get() functions.

In pmll_pmla_functions_set() and pmll_pmla_functions_get() the pointers to the PMLA functions are passed through the functions_p parameter and the functions are registered against the PMLL shadow DB with the pmllDbHandle handle. The registered PMLA functions are used by PMLL to implement the commit and a number of other operations. The return value of zero in all the registered PMLA functions, except for the pmll_pmla_error_string_get() function, indicates that the operation was successful; a non-zero return value indicates an error. PMLL does not interpret the PMLA error codes. The signatures of the PMLA functions that must be registered and the pmll_pmla_functions_t structure are given below.

typedef int pmll_pmla_bulk_begin_function_t (handle_t pmlaHandle);

typedef int pmll_pmla_bulk_end_function_t (handle_t pmlaHandle);

typedef const char *pmll_pmla_error_string_getFunction_t (int errorCode);

typedef int pmll_pmla_flush_function_t (handle_t pmlaHandle);

typedef int pmll_pmla_read_function_t (handle_t pmlaHandle, pmp_msg_t *msg_p);

typedef int pmll_pmla_write_function_t (handle_t pmlaHandle, pmp_msg_t *msg_p);

```
typedef struct {
     pmll_pmla_bulk_begin_function_t

*pmlaBulkBeginFunction_p;
     pmll_pmla_bulk_end_function_t

*pmlaBulkEndFunction_p;
     pmll_pmla_flush_function_t

*pmlaFlushFunction_p;
     pmll_pmla_read_function_t

*pmlaReadFunction_p;
     pmll_pmla_write_function_t

*pmlaWriteFunction_p;
     pmll_pmla_error_string_get_t

*pmlaErrorStringGetFunction_p;
} pmll_pmla_functions_t;
```

When registering the PMLA functions with pmll_pmla_functions_set() function all the functions in the functions_p structure must be defined. A NULL function pointer in the functions_p structure will cause pmll_pmla_functions_set() to fail with the pmll_cannot_register_null_pmla_func_e error code. None of the functions in the functions_p structure are registered in this case.

pmll_pmla_functions_get() retrieves the pointers to the PMLA functions currently registered with the PMLL shadow DB with the pmllDbHandle handle. The retrieved function pointers are returned through the functions_p parameter.

The pmll_pmla_bulk_begin() and pmll_pmla_bulk_end() functions are invoked by the pmll_commit() function at the beginning and the end of loading of the PMLL shadow DB image to the PM hardware. PMLL has no requirements and imposes no restrictions on how these functions are implemented. A failure of either of these functions is logged but otherwise ignored by PMLL. The functions are intended to indicate the start and end of multiple related messages sent by the PMLL to the PMCI.

The pmll_pmla_error_string_get() is expected to return a string describing the specified error code. This function is used by PMLL while generating logs.

The pmll_pmla_flush() is invoked from the pmll_commit() function and it is expected to call the pmci_flush() function. The pmci_flush() function blocks until all the outstanding requests on the PMCI channel are processed.

The pmll_pmla_read() function is expected to call the pmci_read() function, which implements reading of a PMP control message from the PM hardware.

The pmll_pmla_write() function is expected to call the pmci_write() function, which implements writing of a PMP control message to the PM hardware.

Return Value

pmll_pmla_functions_get() and pmll_pmla_functions_set() functions return pmll_ok_e upon success or an error code upon failure.

The pmll_pmla_functions_get() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

The pmll_pmla_functions_set() function can return the following error codes:

```
     pmll_cannot_register_null_pmla_func_e
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

7.4.6.3.6.11 pmll rule functions

pmll_rule_add-Add a rule to a PMLL shadow DB.

pmll_rule_delete-Delete a rule from a PMLL shadow DB.

pmll_rule_all_delete-Delete all rules from a PMLL shadow DB.

pmll_rule_index_next_get-Get the index of the next PMLL rule.

pmll_rule_name_in_use-Check if a rule name is in use.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_rule_add(
     unsigned int
pmllDbHandle,
     uint32_t
recordVersion,
     const
pm_rule_record_t
*ruleRecord_p,
     uint32_t
*index_p);
pmll_status_t pmll_rule_delete(
     unsigned int
pmllDbHandle,
     uint32_t
idx);
pmll_status_t pmll_rule_all_delete(
     unsigned int
pmllDbHandle);
pmll_status_t pmll_rule_index_next_get(
     unsigned int
```

```
pmllDbHandle,
     uint32_t                                                                    idx,
     uint32_t
*nextIndex_p);
pmll_status_t pmll_rule_name_in_use(
     unsigned int
pmllDbHandle,
     const char
*name_p,
     bool
*nameIsInUse_p,
     uint32_t
*idx_p);
```

Description

pmll_rule_add() adds a single PMLL rule to a PMLL shadow DB with the pmllDbHandle handle. The function uses the index_p parameter to return an index that is assigned to the rule by PMLL. Other PMLL rule functions take this index as an input parameter that identifies the rule. The recordVersion parameter indicates the version of the rule record to be added and passed in using the ruleRecord_p parameter. The supported version for Pattern Matcher 1.1 is PMLL_RULE_RECORD_V_1_0_0, which is defined in pmll.h.

The ruleRecord_p parameter is a pointer to the pm_rule_record_t structure, which is defined in pm_defs.h. The pm_rule_record_v_1_0_0_t type below defines the version 1.0.0 of the PMLL rule record. The user needs to create this record and cast it to the generic PMLL rule record type (pm_rule_record_t) and use this generic type in the PMLL functions.

typedef enum {

```
     pm_null_reaction_event_type_e
=  0,
     pm_pattern_reaction_event_type_e
=  1,
     pm_end_of_sui_reaction_event_type_e
=  2,
     pm_last_reaction_event_type_e
= pm_end_of_sui_reaction_event_type_e
} pm_reaction_event_type_t;
```

typedef struct pm_reaction_entry_t {

```
     uint32_t
reactionEventType;
     char
expName_s[PM_NAME_MAX_LENGTH];
     struct pm_reaction_entry_t

*nextReactionEntry_p;
     uint32_t
reactionSize;
     uint8_t
reactionData[PM_MAX_REACTION_SIZE];
} pm_reaction_entry_t;
```

typedef struct ruleRecord_t {

```
     char
name_s[PM_NAME_MAX_LENGTH];
     uint32_t
reactionNum;
     pm_reaction_entry_t
```

```
*reactionEntry_p;
} pm_rule_record_v_1_0_0_t;
```

typedef uint8_t *pm_rule_record_t;

pmll_rule_delete() deletes the rule with the idx index from the PMLL shadow DB with the pmllDbHandle handle.

pmll_rule_all_delete() deletes all the rules from the PMLL shadow DB with the pmllDbHandle handle.

pmll_rule_index_next_get() retrieves the next valid rule index after the idx index in the PMLL shadow DB with the pmllDbHandle handle. The retrieved index is returned through the nextIndex_p parameter. To get the first valid index the idx parameter must be set to the PMLL_NULL_INDEX value. The pmll_rule_index_next_get() function returns pmll_no_more_records_e when there are no valid rule records with index values greater than the passed in idx index value. The pmll_rule_index_next_get() function allows the user to "walk" through all the rules in the PMLL shadow DB in an efficient way.

pmll_rule_name_in_use() checks if the name_p rule name is already in use in the PMLL shadow DB with the pmllDbHandle handle. If the name_p rule name is in use, the function sets the nameIsInUse_p parameter to true and the idx_p parameter to the index of the rule that uses the name_p rule name. If the name_p rule name is not in use, the function sets the nameIsInUse_p parameter to false.

Return Value

All the PMLL rule functions returns pmll_ok_e upon success or an error code upon failure.

The pmll_rule_add() function can return the following error codes:

```
    pmll_bad_reaction_event_type_e
    pmll_exp_is_not_in_name_db_e
    pmll_failed_to_add_rule_to_name_db_e
    pmll_failed_to_alloc_rule_ctx_area_e
    pmll_failed_to_alloc_rule_ctx_area_e
    pmll_invalid_db_handle_e
    pmll_invalid_name_e
    pmll_misaligned_reaction_e
    pmll_module_not_initialized_e
    pmll_no_reactions_in_rule_e
    pmll_no_reactions_in_rule_e
    pmll_null_pointer_parameter_e
    pmll_Aout_of_memory_e
    pmll_out_of_rule_ids_e
    pmll_reaction_too_big_e
    pmll_reaction_too_small_e
    pmll_rule_name_is_in_use_e
    pmll_too_many_reactions_in_rule_e
    pmll_too_many_rules_e
    pmll_unsupported_record_version_e
```

The pmll_rule_delete() function can return the following error codes:

```
    pmll_index_is_not_in_use_e
    pmll_invalid_db_handle_e
    pmll_module_not_initialized_e
```

The pmll_rule_all_delete() function can return the following error codes:

```
    pmll_invalid_db_handle_e
    pmll_module_not_initialized_e
    pmll_rule_is_not_in_name_db_e
```

The pmll_rule_index_next_get() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_no_more_records_e
     pmll_null_pointer_parameter_e
```

The pmll_rule_name_in_use() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

See Also

pmll_supported_rule_record_versions_get(), pmll_supported_rule_record_versions_num_get()

7.4.6.3.6.12 pmll_stats_get

pmll_stats_get-Retrieve the PMLL statistics.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_stats_get(
     unsigned int
pmllDbHandle,
     pmll_stats_t
*llStats_p);
```

Description

The pmll_stats_get() function retrieves the PMLL statistics for the PMLL shadow DB with the pmllDbHandle handle. The retrieved statistics are returned to the caller through the llStats_p parameter. The pmll_stats_t structure is defined as follows:

typedef struct {

```
     struct {                                        // uncompressed trigger stats.
          uint32_t  variableChainNum;                   // # of variable trigger chains
          uint32_t  twoByteChainNum;                    // # of 2-byte trigger chains
          uint32_t  oneByteChainNum;                  // # of 1-byte trigger chains
          uint32_t  specialChainNum;                    // # of special trigger chains
     };
     struct {                                        // post-cluster-compres. trigger stats.
          // # of variable resident trigger chains
          uint32_t  variableResidentChainNum;
          // # of variable guest trigger chains
          uint32_t  variableGuestChainNum;
          // # of 2-byte resident trigger chains
          uint32_t  twoByteResidentChainNum;
          // # of 2-byte guest trigger chains
          uint32_t  twoByteGuestChainNum;
          // # of 1-byte resident trigger chains
          uint32_t  oneByteResidentChainNum;
          // # of 1-byte guest trigger chains
          uint32_t  oneByteGuestChainNum;
          // # of special resident trigger chains
          uint32_t  specialResidentChainNum;
          // # of special guest trigger chains
          uint32_t  specialGuestChainNum;
```

```
     };
     struct {                                        // post-foldback-compression trigger
stats.
          // number of primary entries
          uint32_t  primaryEntryNum;
          // number of secondary entries
          uint32_t  secondaryEntryNum;
     };
} pmll_stats_t;
```

Return Value

The pmll_stats_get() function returns pmll_ok_e upon success or an error code upon a failure. The pmll_stats_get() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

7.4.6.3.6.13 pmll version functions

pmll_supported_exp_record_versions_get-Get the supported expression record versions.

pmll_supported_exp_record_versions_num_get-Get the number of the supported expression versions.

pmll_supported_rule_record_versions_get-Get the supported rule record versions.

pmll_supported_rule_record_versions_num_get-Get the number of the supported rule versions.

pmll_version_str_get-Get a version description string.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_supported_exp_record_versions_get(
     uint32_t *versionTable_p,
     uint32_t tableSize);
uint32_t pmll_supported_exp_record_versions_num_get(void);
pmll_status_t pmll_supported_rule_record_versions_get(
     uint32_t *versionTable_p,
     uint32_t tableSize);
uint32_t pmll_supported_rule_record_versions_num_get(void);
char * pmll_version_str_get(
     uint32_t  version,
     char     *versionStr_p);
```

Description

pmll_supported_exp_record_versions_get() allows the user to retrieve the numerical values of the supported expression record versions. The number of the supported versions can be retrieved with pmll_supported_exp_record_versions_num_get(). The supported versions are returned to the caller through the versionTable_p table which should have tableSize entries in it.

pmll_supported_exp_record_versions_num_get() retrieves the number of the supported expression record versions.

pmll_supported_rule_record_versions_get() allows the user to retrieve the numerical values of the supported rule record versions. The number of the supported versions can be retrieved with pmll_supported_rule_record_versions_num_get(). The supported versions are returned to the caller through the versionTable_p table which should have tableSize entries in it.

pmll_version_str_get() returns a string describing the numerical version passed in the version parameter. The returned version string is stored in the versionStr_p buffer that should be at least PMLL_VERSION_STR_LENGTH bytes long.

Return Value

The pmll_supported_exp_record_versions_get() and pmll_supported_rule_record_versions_get() functions return pmll_ok_e upon success or an error code upon failure.

The pmll_supported_exp_record_versions_get() function can return the following error codes:

```
     pmll_null_pointer_parameter_e
```

The pmll_supported_rule_record_versions_get() function can return the following error codes:

```
     pmll_null_pointer_parameter_e
```

The pmll_supported_exp_record_versions_num_get() and pmll_supported_rule_record_versions_num_get() return the numbers of the supported expression or rule record versions, respectively.

The pmll_version_str_get() function returns the pointer passed in the versionStr_p parameter.

7.4.6.3.6.14 pmll_variable_trigger_size_get, pmll_variable_trigger_size_set

pmll_variable_trigger_size_get-Get the size of the variable trigger.

pmll_variable_trigger_size_set-Set the size of the variable trigger.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_variable_trigger_size_get(
     unsigned int pmllDbHandle,
     uint32_t *variableTriggerSize_p);
```

pmll_status_t pmll_variable_trigger_size_set(

```
     unsigned int pmllDbHandle,
     uint32_t variableTriggerSize);
```

Description

pmll_variable_trigger_size_get() retrieves the currently configured value of the variable trigger size in the PMLL shadow DB with the pmllDbHandle handle. The retrieved value is returned to the caller through the variableTriggerSize_p parameter.

pmll_variable_trigger_size_set() sets the value of the variable trigger size in the PMLL shadow DB with the pmllDbHandle handle to variableTriggerSize. The function should be called prior to adding any expressions to the PMLL shadow DB. The function will return a failure status code if it is invoked when there are expressions present in the PMLL shadow DB.

Return Value

The pmll_variable_trigger_size_get() and pmll_variable_trigger_size_set functions return pmll_ok_e upon success or an error code upon a failure.

The pmll_variable_trigger_size_get() function can return the following error codes:

```
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_null_pointer_parameter_e
```

The pmll_variable_trigger_size_set() function can return the following error codes:

```
     pmll_exps_are_present_in_db_e
     pmll_invalid_db_handle_e
     pmll_module_not_initialized_e
     pmll_vtrigger_size_out_of_range_e
```

7.4.6.3.7 Pattern Matcher driver

This chapter describes the interfaces provided by the Pattern Matcher driver. The Pattern Matcher driver can be built into the kernel statically or as a set of dynamically-loaded kernel modules. In both cases, a small piece of platform support is built into the kernel statically to handle platform-initialisation, early-boot contiguous memory allocation, etc. The driver provides both user space and kernel space APIs that are used to configure the hardware as well as perform pattern matching operations.

The user space API is described in the following sections:

PME Configuration on page 940 provides the details on the various PME configuration requirements and sysfs interface. The configuration attribute may reside in the device-tree, as Kconfig options and as sysfs entries. There are also statistics which are collected on a continuous basis and made available either via a kernel api and sysfs.

PME Scan on page 951 provides an interface to perform pattern matching operations. This interface operates on the /dev/pme_scan device using various ioctls.

PME Database on page 947 provides an interface for configuring and modifying the PME database in hardware. This operates via the /dev/pme_db device and would not typically be used directly but instead user-space interfaces like PMCI and PMM would be used that incorporate the proprietary tools necessary to compile, link, and load patterns and rules.

The various user space ioctl() calls require an appropriate device descriptor parameter. The devices are created via a call to open(). The table, Devices for IOCTL System Calls, summarizes the devices required for each type of ioctl system call.

Devices for IOCTL System Calls

Device	IOCTL Type
/dev/pme_scan	PME Scan
/dev/pme_db	PME Database

PME Kernel API on page 960 allows kernel programmers to create pattern matcher contexts that can be used for scanning operations.

The PME Driver is a patch-set to the kernel providing a driver (static or module) and a header file that defines the driver interfaces. If built as a module, the driver names are pme.ko, pme_scan.ko and pme_db.ko. The header file is fsl_pme.h, located in the linux include directory-included via "#include <linux/fsl_pme.h>".

7.4.6.3.7.1 PME Configuration

Various items must be configured in order to utilize the PME device. These include device-tree entries, Kconfig parameters, module parameters (when using loadable modules) and also sysfs entries.

7.4.6.3.7.1.1 Device Tree Node Entry

Synopsis

pme: pme@316000 {

```
          compatible = "fsl,pme"
          reg = <0x316000 0x10000>;
 /* fsl,pme-pdsr = <0x0 0x23000000 0x0 0x01000000>; */
 /* fsl,pme-sre = <0x0 0x24000000 0x0 0x00a00000>; */
 interrupts = <16 2 1 5>;
};
```

Description

The Pme device tree node represents the Pme device and its CCSR configuration space. When a linux kernel has Pme support built in, it will react to this device tree node by configuring and managing the Pme device. It uses the Pme configuration interface to implement this. The device-tree node sits within the CCSR node ("soc").

"*fsl,pme-pdsr*" property specifies the start location and size of the Pattern Description and Stateful Rule Table in system memory. The table base address must be aligned to a natural 128-byte address boundary. The maximum size of the table is 128 Mbytes for PME v.2.0, 64 Mbytes for PME v2.1 and 32 Mbytes for PME v2.2. The current driver implementation allows this memory resource to be specified via the "*fsl,pme-pdsr*" device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. This property specifies a 2-tuple of address and size (address is optional), specifying the physical address range. These elements are expressed as 64-bit values, so take two cells each;

fsl,pme-pdsr = <0x0 0x20000000 0x0 0x01000000>;

Optionally, only the size can be specified and the kernel will try to allocate the contiguous memory during boot time.

fsl,pme-pdsr = <0x0 0x100000>

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the Linux OS, it will attempt to reserve the range against use by the OS.

"*fsl,pme-sre*" property specifies the start location and size of the SRE Context Table in system memory. The table base address must be aligned to a natural 32-byte address boundary. The maximum size of the table is 4 Gbytes. The current driver implementation allows this memory resource to be specified via this device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. This node property specifies a 2-tuple of address and size (address is optional), specifying the physical address range. These elements are expressed as 64-bit values, so take two cells each;

fsl,pme-sre = <0x0 0x20000000 0x0 0x01000000>;

Optionally, only the size can be specified and the kernel will try to allocate the contiguous memory during boot time.

fsl,pme-sre = <0x0 0x300000>;

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the Linux OS, it will attempt to reserve the range against use by the OS.

7.4.6.3.7.1.2 KCONFIG Options

Description

The Pme device supports various kernel configuration options.

FSL_PME2_PDSRSIZE

Pattern Description and Stateful Rule default table size defined as the number of 128 byte entries. This only takes effect if the device tree node doesn't have the 'fsl,pme-pdsr' property present. Valid range is 74240-1048573 (9.5MB-128MB). The default is 131072 (16MB)

FSL_PME2_SRESIZE

SRE Session Context Entries table default table size defined as the number of 32 byte entries. This only takes effect if the device tree node doesn't have the 'fsl,pme-sre' property present. Valid range is 0-134217727 (0-4GB) and the default is 327680 (10MB)

FSL_PME2_SRE_AIM

Select the inconclusive match mode treatment. When true the alternate inconclusive mode is used. When false the default inconclusive mode is used. The default is off.

FSL_PME2_SRE_ESR

Configures if an End of SUI event will produce a Simple End of SUI report.

FSL_PME2_SRE_CTX_SIZE_PER_SESSION

Configures the default SRE Context Size per Session as a power of 2 (e.g. a value of 5 translated to 32 bytes, whereas a value of 15 translated to 32 kbytes. Valid range is 5 to 17 with a default of 17.

FSL_PME2_SRE_CNR

Configured Number of Stateful Rules as a multiple of 256. Valid range is 0 to 128 with a default of 128 (i.e. 32kbytes).

FSL_PME2_SRE_MAX_INSTRUCTION_LIMIT

Configured maximum number of SRE instructions to be executed per reaction. Valid range is 0 to 65535 with a default of 65535.

FSL_PME2_SRE_MAX_BLOCK_NUMBER

Configured the maximum number of reaction head blocks to be traversed perpattern match event (e.g. a matched pattern or an End of SUI event).Valid range is 0 to 32767 with a default of 32767.

FSL_PME2_DB_QOSOUT_PRIORITY

Configures the /dev/pme_db qos priority level for its output frame queue. Valid range is 0 to 7 with a default of 2.

FSL_PME2_STAT_ACCUMULATOR_UPDATE_INTERVAL

The pme accumulator kernel thread reads the current device statistics and add it to a running counter. The frequency (expressed in milliseconds) of these updates may be controlled. If 0 is specified, no automatic updates are done. Valid range is 0 to 10000 with a default of 3400 (e.g. 3400 milliseconds).

7.4.6.3.7.1.3 SYSFS Attributes

Description

The sysfs interface is used to control global pme configuration Pme device parameters and provides an interface for pme statistics. The path to the device attributes is /dev/fsl-pme-dev. The following attributes are defined:

aim

This read-write field specifies the inconclusive match mode treatment. When this field is set to 1, the "alternate" inconclusive mode is used. When this field is set to 0 the "default" inconclusive mode is used. The default is 0.

bsc/[0-63]

This read-write field specifies the buffer pool size for the specified buffer pool id (0-63). When PME h/w acquires a buffer pool it determines the size of these buffers based on this configuration. A value of 0 causes the PME not to attempt any acquires. Valid range is 0 to 11 encoded as follows: 0 - 0 1 - 64 bytes 2 - 128 bytes 3 - 256 bytes 4 - 512 bytes 5 - 1024 bytes 6 - 2048 bytes 7 - 4096 bytes 8 - 8192 bytes 9 - 16,384 bytes 10 - 32,768 bytes 11 - 65,536 bytes

cdcr

This read-write entry disables the specified Pme caching. This value should generally always be zero (i.e. caching always enabled).

dmcr

This read-write entry is the DXE memory control settings.

ecc1bes

This read-write entry contains status bits indicating which internal SRAM instances have accumulated more single bit errors than their programmed threshold.

ecc2bes

This read-write entry contains status bits indicating which internal SRAM instances have had at least one double bit error detected.

eccaddr

This read-only entry, when a bit becomes set in the ecc1bes or ecc2bes registers, the Pme internal memory, and the address within that memory, on which the error was detected are recorded in this entry.

ecccode

This read-only entry, when a bit becomes set in the ecc1bes or ecc2bes entry, the corresponding ECC code and syndrome are recorded in this entry.

ecr0

This read-only entry captures debug information that is specific to the most recent Status code of the highest severity error that the Pme has seen but not necessarily communicated to the CPU via a report.

ecr1

This read-only entry captures debug information that is specific to the most recent Status code of the highest severity error that the Pme has seen but not necessarily communicated to the CPU via a report.

efqc_int

This read-write entry specifies the delay (an interval) between consecutive PMFA dequeue requests. Valid value are: 0, 64, 128, 256 (default) and 512.

end_of_simple_sui_report

This read-write entry specifies the delay (an interval) between consecutive PMFA dequeue requests. Valid value are: 0, 64, 128, 256 (default) and 512.

end_of_sui_reaction_ptr

This read-write entry specifies the head pointer of the reaction linked list that is to be traversed by the SRE for every End of SUI event. The pointer is an index to a 128-byte block relative to the start of the PDSR table. Valid rang is 0x0 to 0xFFFFF for PME ver 2.0, 0x0 to 0x7FFFF for PME ver 2.1 and 0x0 to 0x3FFFF for PME ver 2.2. The default is 0x0.

esr

This read-write attribute contains the Status code of the highest severity error that the Pme has seen but not necessarily communicated to the CPU via a report.t

faconf

This read-only attribute is the frame agent configuretion.

famcr

This read-write attribute configures he frame agent memory control.

isr

This read-only attribute displays the current interrupt status register value. A non zero value indicates that a serious error was detected and consequently the PME device stops processing input until informed otherwise.

kvlts

This read-write entry specifies the size of the KES Variable Length Trigger from 2 to 16 bytes. Valid range is 0 to 16 with 2 being the default.

liodnr

This read-only entry specifies the LIODN value to use when PM accesses its private data structures.

max_allowed_test_line_per_pattern

This read-write entry specifies maximum allowed test line executions per pattern (scaled by a factor of 8; the actual maximum allowed test line executions is equal to MTE*8). The maximum allowed value for this field is 0x3FFF which translates into a maximum allowed test line execution value of 131,064. Valid range is 0x0 to 0x3FFF with 0x3FFF being the default.

max_chain_length

This read-write entry specifies the maximum chain length. This is the maximum number of allowed entries in confidence collision chains. Valid range is 0x0 to 0x7FFF with 0x7FFF being the default.

max_pattern_evaluations_per_sui

This read-write entry specifies maximum pattern evaluations per SUI (scaled by a factor of 8; the actual maximum allowed pattern evaluations is equal to value*8). A value of 0xFFFF will be treated as an infinite limit. Valid range is 0x0 to 0xFFFF with 0xFFFF being the default.

max_pattern_matches_per_sui

This read-write entry specifies maximum pattern matches per SUI (scaled by a factor of 8; the actual maximum allowed pattern matches is equal to value*8). A value of 0xFFFF will be treated as an infinite limit. Valid range is 0x0 to 0xFFFF with 0xFFFF being the default.

max_pdsr_index

This read-write entry specifies the maximum allocated index into the Pattern Description and Stateful Rule table available to the DXE. Valid range is 0x0 to 0xFFFFC for PME ver 2.0, 0x0 to 0x7FFFC for PME ver 2.1 and 0x0 to 0x3FFFC for PME ver 2.2. The index is 128-byte based.

miace

This read-write attribute is used specify the client read and write ports that are included in the miablc and miabyc counts. It is also used to specify the size of the aligned blocks counted by miablc

miacr

This read-write attribute is used to configure various client read and write port attributes.

pattern_range_counter_idx

This read-write entry specifies the index value targeted for counting pattern matches. Note that this field is only wide enough to allow specification of indices within the PDSR table that the DXE will access. Table entries beyond 0x1FFFF are only accessible by the SRE. Valid range is 0x0 to 0x1FFFF for PME ver 2.0, 0x0 to 0xFFFF for PME ver 2.1 and 0x0 to 0x7FFF for PME ver 2.2. The default is 0x0.

pattern_range_counter_mask

This read-write entry specifies the index mask for pattern_range_counter_index, allowing ranges of addresses to be targeted. A 0 in any bit means a "don't care" in the corresponding pattern_range_counter_index bit. Valid range is 0x0 to 0x1FFFF for PME ver 2.0, 0x0 to 0xFFFF for PME ver 2.1 and 0x0 to 0x7FFF for PME ver 2.2, with 0x0 being the default.

pdsrbah

This read-only attribute is used to provision the start location (upper bit 32 bits) of the PRSR table space in system memory.

pdsrbal

This read-only attribute is used to provision the start location (lower 32 bits) of the PRSR table space in system memory.

pehd

This read-write attribute disables selected Pme (KES, DXE or SRE) error condition detection.

pmtr

This read-write attribute allows programming of a time based latency threshold such that memory read transactions whose latency exceeds the threshold can be measured using the integrated performance monitor.

report_length_limit

This read-write entry specifies the maximum number of bman buffers that can be allocated for a single output report frame. A value of zero disables length limiting. Valid range is 0x0 to 0xFFFF with 0x0 being the default.

rev1

This read-only attribute provides the unique IP block ID for the Pattern Matcher block, as well as major and minor revision numbers. It is displayed as a hexadecimal value.

rev2

This read-only attribute provides the Pattern Matcher block integration and configuration options.

scbarh

This read-only attribute is used to provision the start location (upper bit 32 bits) of the SRE table space in system memory.

scbarl

This read-only attribute is used to provision the start location (lower 32 bits) of the SRE table space in system memory.

smcr

This read-write attribute is used to provide software with control over system memory transaction attributes for the SRE.

src_id

This read-only attribute specifies the source id that the PME uses in system bus transactions.

sre_context_size

This read-only entry specifies the context size per session. This represent the FSL_PME2_SRE_CTX_SIZE_PER_SESSION Kconfig value. Valid range is 0 to 13 encoded as follows: 0 - 0 1 - 32 bytes 2 - 64 bytes ... 13 - 128 bytes

sre_max_block_num

This read-write entry specifies the maximum number of reaction head blocks to be traversed per pattern match event. Valid range is 0x0 to 0x3FFF with 0x3FFF being the default.

sre_max_index_size

This read-only entry specifies the maximum allocated 128-bytes based index in the Pattern Description and Stateful Rule table that is to be accessed by the SRE. Valid range is 0x0 to 0xFFFFC.

sre_max_instruction_limit

This read-write entry specifies the maximum number of SRE instructions to be executed per reaction. Valid range is 0x0 to 0xFFFF with 0xFFFF being the default.

sre_max_offset_ctrl

This read-only entry specifies maximum 32-byte index offset for the SRE context table.

sre_pscl

This read-only entry specifies the prescaler value for the SRE Instruction Operand register $C free running counter

sre_rule_num

This read-only entry specifies the configured number of Stateful Rules as a multiple of 256. The default is 128 (i.e. 32kbytes). This represents the FSL_PME2_SRE_CNR Kconfig value.

sre_session_ctx_num

This read-only entry specifies the maximum number of session contexts. This is derived by dividing the size of the SRE table size by per session configured size.

sw_db

This read-write entry can be used by software to hold a version number for the pattern match database. This attribute is currenly not used.

The PME h/w maintains various statistics. The Pme drivers polls these statistics at a configurable interval. The h/w statistics are read-reset based whereas the driver simply keep a running 64 bit counter. Writing a zero to the sysfs entry will clear both the driver and h/w counter. The sysfs interface to these statistics are located in the *stats* subdirectory. The following statistics are defined:

cmecc1ec

This read-write attribute contains a count of the number of single bit ECC errors that have occurred.

dxcmecc1ec

This read-write attribute contains a count of the number of single bit ECC errors that have occurred.

dxemecc1ec

This read-write attribute contains a count of the number of single bit ECC errors that have occurred.

mia_blc

This read-write attribute represents the number of bus aligned memory block accesses for all memory transactions that reach the system bus.

mia_byc

This read-write attribute represents the number of bytes transferred for all memory transactions that reach the system bus.

rbc

This read-write attribute indicates the number of read bytes.

stnch

This read-write attribute indicates the number of confidence check passed by the confidence mechanism.

stndsr

This read-write attribute indicates the number of Stateful Rule executions caused by DXE generated events.

stnesr

This read-write attribute indicates the number of Stateful Rule executions caused by End of SUI events.

stnib

This read-write attribute indicates the number of input bytes scanned by the Pattern Matcher.

stnis

This read-write attribute indicates the number of SUIs scanned by the Pattern Matcher.

stnob

This read-write attribute indicates the number of output bytes produced in reports by the Pattern Matcher.

stnpm

This read-write attribute indicates the number of pattern matches found by the Data Examination Engine.

stnpmr

This read-write attribute indicates the number of pattern match attempts within the index range defined by the Pattern Range Counter Index Configuration and Pattern Range Counter Mask Configuration attributes.

stns1m

This read-write attribute indicates the number of SUIs with at least one pattern match found within.

stns1r

This read-write attribute indicates the number of SUIs scanned that produced at least one report.

stnth1

This read-write attribute indicates the number of Triggers found by the 1-Byte Trigger mechanism.

stnth2

This read-write attribute indicates the number of Triggers found by the 2-Byte Trigger mechanism.

stnths

This read-write attribute indicates the number of Triggers found by the Special Triggers mechanism.

stnthv

This read-write attribute indicates the number of Triggers found by the Variable Length Trigger mechanism.

tbt0ecc1ec

This read-write attribute indicates the number of single bit ECC errors detected in the 2-byte Trigger 0 Table.

tbt1ecc1ec

This read-write attribute indicates the number of single bit ECC errors detected in the 2-byte Trigger 1 Table.

trunci

This read-write attribute indicates the count for every scan report that is produced where the report data has been truncated.

vlt0ecc1ec

This read-write attribute the number of single bit ECC errors detected in the Variable Length Trigger 0 Table.

vlt1ecc1ec

This read-write attribute indicates the number of single bit ECC errors detected in the Variable Length Trigger 1Table.

The Pme driver which polls these statistics has a configurable polling interval.

stats_ctrl/update_interval

This read-write attribute indicates the frequency (expressed in milliseconds) of the pme stats kernel thread updates. If 0 is specified, no automatic updates are done. Valid range is 0 to 10000 with a default of 4000 (e.g. 4 seconds).

The following attribute configure statistic related attributes:

stats_ctrl/cmecc1th

This read-write attribute contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/dxcmecc1th

This read-write attribute contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/dxemecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/tbt0ecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/tbt1ecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/vlt0ecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/vlt1ecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

7.4.6.3.7.2 PME Database

The /dev/pme_db devices device is used to send Pme pattern matcher database configuration requests. This device requires root permissions and only exists on the control plane. A user may acquire exclusive access to the Pattern Matcher device.

7.4.6.3.7.2.1 PMEIO_EXL_INC

PMEIO_EXL_INC-Acquire and increment exclusivity.

Synopsis

```
        ioctl(fd, PMEIO_EXL_INC);
```

Description

This ioctl() increments the exclusivity "counter". This enabled the user to acquire and keep exclusivity. This is a synchronous and interruptible API.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

EINVAL

The parameters specified are not valid

EINTR

The process was interrupted by a signal.

7.4.6.3.7.2.2 PMEIO_EXL_DEC

PMEIO_EXL_DEC-Decrements reference count and possible release exclusivity.

Synopsis

```
ioctl(fd, PMEIO_EXL_DEC);
```

Description

This ioctl() decrement the exclusivity reference counter. If the reference count reaches zero than exclusivity is also released.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

EINVAL

The parameters specified are not valid

EINTR

The process was interrupted by a signal.

7.4.6.3.7.2.3 PMEIO_EXL_GET

PMEIO_EXL_GET-Retrieve exclusivity reference count

Synopsis

```
ioctl(fd, PMEIO_EXL_GET, int *count);
```

Description

This ioctl() returns the currnent exclusivity counter and writes the result to *count*.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

EINVAL

The parameters specified are not valid

EINTR

The process was interrupted by a signal.

7.4.6.3.7.2.4 PMEIO_PMTCC

PMEIO_PMTCC-Send a synchronous pattern matching database request

Synopsis

```
ioctl(fd, PMEIO_PMTCC, struct pme_db *p);

struct pme_db {

    struct pme_buffer input;

    struct pme_buffer output;

    __u8 flags;

    enum pme_status status;

};

struct pme_buffer {

    void *data;

    size_t size;

};
```

Description

This ioctl() is used to send a synchronous pattern matching database request.

p contains the pme pmtcc command and response. Various input and output formats are supported depending on the input flag settings.

p->input is the contiguous input data buffer. Where *p->input.data* is the pointer to the contiguous user-space buffer and *p->input.size* indicates the number of bytes in this buffer.

p->output is where the resulting output/result is stored and is populated by the ioctl(). The *p->output.size* is updated to reflect the number of bytes written.

p->flags attribute is populated by the ioctl and is bit wise mask indicating operation results:

PME_DB_RESULT_UNRELIABLE-When set, indicates that this input was processed following detection and reporting of a serious Pattern Matcher error. Any report data carried within this input that was produced by Pattern Matcher cannot be considered reliable. Output tagged as Unreliable do not have a valid Status code.

PME_DB_RESULT_TRUNCATED-When set, indicates that the contents of the output data were truncated and are incomplete.

p->status attribute is the status code of the operation. This value is invalid if the PME_DBS_RESULT_UNRELIABLE bit mask was set in *p->flags*. Refer to fsl_pme.h for complete list of possible values:

pme_status_ok-operation was successful

all other values indicate an exception occurred.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.2.5 PMEIO_SRE_RESET

PMEIO_SRE_RESET-Performs an SRE rule reset operation

Synopsis

```
         ioctl(fd, PMEIO_SRE_RESET, struct pme_db_sre_reset *sre_reset);
struct pme_db_sre_reset {
    __u32 rule_vector[PME_SRE_RULE_VECTOR_SIZE];
    __u32 rule_index;
    __u16 rule_increment
    __u32 rule_repetitions;
    __u16 rule_reset_interval;
    __u8 rule_reset_priority;
};
#define PME_SRE_RULE_VECTOR_SIZE 8
```

Description

This ioctl() performs a sre rule reset operation. Exclusivity should be held during this operation in order to perform the operation atomically.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.2.6 PMEIO_NOP

PMEIO_NOP-Send a Pme nop command

Synopsis

```
ioctl(fd, PMEIO_NOP);
```

Description

This ioctl() send a Pme nop command. This has no effect on the Pme device. This is a blocking api. Upon return, this indicates that the response to the nop command has been received.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3 PME Scan

The /dev/pme_scan device is used to execute patterning matching from user space. The device can be used to peform synchronous (the calling thread is blocked until an operations completion) or asynchronous (once an operation has begun, the calling thread is able to perform other processing but needs to query the completion of the operation later). This section specifes the ioctl() values supported by the pme_scan device.

7.4.6.3.7.3.1 PMEIO_SCAN

PMEIO_SCAN-Perform a pattern matching request

Synopsis

```
ioctl(fd, PMEIO_SCAN, struct pme_scan *p);
```

struct pme_scan {

```
    struct pme_scan_cmd cmd;
    struct pme_scan_result result;
};
struct pme_scan_cmd {
    __u32 flags;
    void *opaque;
    struct pme_buffer input;
```

```
        struct pme_buffer output;

    };

    struct pme_scan_result {

        __u8 flags;

        enum pme_status status;

        struct pme_buffer output;

        void *opaque;

    };

    struct pme_buffer {

        void *data;

        size_t size;

    };
```

Description

This ioctl() is used to perform a synchronous pattern matching request. The API will block until the device has completed the request.

p contains the pme scan input command and response. Various input and output formats are supported depending on the input flag settings.

p->cmd.flags attribute defines input and output formats. It's a bit mask of the following values:

PME_SCAN_CMD_RES_BMAN-use Bman for output

PME_SCAN_CMD_STARTRESET- If residue is disabled indicates a Start of Flow. The first data byte of the input will be treated as the Start of Flow for anchored pattern matching purposes. If residue is enabled indicates a Flow Context Reset. The Start of Flow, Sequence Number and Residue Length fields in the Flow Context Record will be reset to 0x0 prior to scanning the input. Empty input data is accepted and may be used as a mechanism to reset Flow Context without scanning a input data.

PME_SCAN_CMD_END-Indicates that the last byte of input data will be considered as the end of Flow. As well, the Flow Context's sequence number and residue length context are reset to zero such that the next data byte on the Flow will be considered a Start of Flow.In order to accommodate protocols such as TCP, where it may not be known until later that the last processed byte was in fact the End of Flow byte, input data with zero length but with the End of Flow indicator set are allowed. In these cases, the session's residue will be recycled through the Pattern Matcher with the End of Flow indication such that any anchored patterns present won't be missed.

The *p->cmd.opaque* is carried through into *p->result.opaque*.

p->cmd.input is the contiguous input data buffer. Where *p->cmd.input.data* is the pointer to the contiguous user-space buffer and *p->cmd.input.size* indicates the number of bytes in this buffer.

p->cmd.output is where the resulting output/result is stored. The caller indicates where the write the result of the scan via this attribute.

p->result is the response of the scan operation and is updated upon return.

p->result.flags attribute is populated by the ioctl and is bit wise mask indicating operation results:

PME_SCAN_RESULT_UNRELIABLE-When set, indicates that this input was processed following detection and reporting of a serious Pattern Matcher error. Any report data carried within this input that was produced by Pattern Matcher cannot be considered reliable. Output tagged as Unreliable do not have a valid Status code.

PME_SCAN_RESULT_TRUNCATED-When set, indicates that the contents of the output data were truncated and are incomplete.

PME_SCAN_RESULT_BMAN-The output was generated from BMan buffers.

p->result.status attribute is the status code of the operation. This value is invalid if the PME_SCAN_RESULT_UNRELIABLE bit mask was set in *p->result.flags*. Refer to fsl_pme.h for complete list of possible values:

pme_status_ok-operation was successful

all other values indicate an exception occurred.

p->result.output is updated with the output generated from the Pattern Matcher device. *p->result.output.data* is *p->cmd.output.data* and p->result.output.size <= *p->cmd.output.size.* In other words the size of the output from the Pattern Matcher may be less than the size that the caller has supplied.

p->result.opaque is set to the corresponding *p->cmd.opaque* value.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3.2 PMEIO_SCAN_W1

PMEIO_SCAN_W1-Perform a single asynchronous pattern matching request

Synopsis

```
         ioctl(fd, PMEIO_SCAN_W1, struct pme_scan_cmd *cmd);
struct pme_scan_cmd {
    __u32 flags;
    void *opaque;
    struct pme_buffer input;
    struct pme_buffer output;
};
struct pme_buffer {
    void *data;
    size_t size;
};
```

Description

This ioctl() is used to perform a single asynchronous pattern matching request. The API will return once the request has been dispatched to the Pattern Matcher device. The resulting response must be retrieved via PMEIO_SCAN_R1 or PMEIO_SCAN_Rn ioctls. The input memory should not be modified by the caller until the corresponding response has been retrieved.

cmd contains the pme scan input command. Refer to PMEIO_SCAN ioctl for a detail description.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3.3 PMEIO_SCAN_R1

PMEIO_SCAN_R1-Retrieve a single Pattern Matcher response.

Synopsis

```
         ioctl(fd, PMEIO_SCAN_R1, struct pme_scan_result *result);
struct pme_scan_result {
    __u8 flags;
    enum pme_status status;
    struct pme_buffer output;
    void *opaque;
};
struct pme_buffer {
    void *data;
    size_t size;
};
```

Description

This ioctl() is used to retrieve a single Pattern Matcher response from a corresponding PMEIO_SCAN_W1 or PMEIO_SCAN_Wn ioctl request. This is a non-blocking api. User should wait for response with select() or poll(), then collect the response(s) with either PMEIO_SCAN_R1 or PMEIO_SCAN_Rn ioctl().

result contains the response from a corresponding PMEIO_SCAN_W1 or PMEIO_SCAN_Wn ioctl(). Refer to PMEIO_SCAN ioctl for a detail description.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3.4 PMEIO_SCAN_Wn

PMEIO_SCAN_Wn-Performs multiple asynchronous pattern matching request

Synopsis

```
         ioctl(fd, PMEIO_SCAN_Wn, struct pme_scan_cmds *cmds);
struct pme_scan_cmds {
    unsigned num;
    struct pme_scan_cmd *cmds;
};
```

Description

This ioctl() is a more generalized version of the PMEIO_SCAN_W1 ioctl(). Permits sending multiple request asynchronously. The results must be retrieved using the SCAN_R1 or SCAN_Rn ioctl(). User's should first call select() or poll() to indicate that responses are ready to be retrieved. The ioctl may not have sent all scan requests: in this case -EINT is returns and *cmds->num* is updated with the number of commands sent.

cmds->cmds is a pointer to an array of pme_scan_cmd structures

cmds->num is the number of elements in the above array.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3.5 PMEIO_SCAN_Rn

PMEIO_SCAN_Rn-Retrieve multiple Pattern Matcher responses synchronously.

Synopsis

```
         ioctl(fd, PMEIO_SCAN_Rn, struct pme_scan_results *results);
struct pme_scan_results {
    unsigned num;
    struct pme_scan_result *result;
};
```

Description

This ioctl() is a more generalized version of the PMEIO_SCAN_R1 ioctl(). Permits retrieving multiple request synchronously.

results->result is a pointer to an array of pme_scan_result structures

result->num is the number of elements in the above array. This value is updated upon return to indicate the actual number of responses retrieved.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3.6 PMEIO_RELEASE_BUFS

PMEIO_RELEASE_BUFS-Release the allocated buffer back to hardware

Synopsis

```
ioctl(fd, PMEIO_RELEASE_BUFS, void *user_mem);
```

Description

This ioctl() release the allocated buffer back to hardware.

user_mem is a pointer to data which has been zero copied mmap'ed. This memory will be released to BMan. For intance this would be *&result.output.data*.

Return Value

Returns 0 on success, -1 with errno set upon error.

Errors

See Also

7.4.6.3.7.3.7 PMEIO_RESETSEQ

PMEIO_RESETSEQ-Resets the flow context sequence number to zero

Synopsis

```
ioctl(fd, PMEIO_RESETSEQ);
```

Description

This ioctl() resets the flow context sequence number to zero. The Start of Flow indicator is also set.

Return Value

Returns 0 on success, -1 with errno set upon error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3.8 PMEIO_RESETRES

PMEIO_RESETRES-Resets the flow context residue

Synopsis

```
ioctl(fd, PMEIO_RESETRES);
```

Description

This ioctl() resets the flow context residue such that no bytes remain in residue.

Return Value

Returns 0 on success, -1 with errno set upon error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3.9 PMEIO_SETSCAN

PMEIO_SETSCAN-Configures a pme context parameters

Synopsis

```
ioctl(fd, PMEIO_SETSCAN, struct pme_scan_params *p);
```

struct pme_scan_params {

```
    __u32 flags;
    struct pme_scan_params_residue {
        __u8 enable;
        __u8 length;
    } residue;
    struct pme_scan_params_sre {
        __u32 sessionid;
        __u8 verbose;
        __u8 esee;
```

```
            } sre;

            struct pme_scan_params_dxe {

                __u16 clim;

                __u16 mlim;

            } dxe;

            struct pme_scan_params_pattern {

                __u8 set;

                __u16 subset;

            } pattern;

        };
```

Description

This ioctl() sets the flow context and scan related attributes. This is a blocking API.

p->flags attribute is a bit wise mask which indicate which group of attributes are being set. The flags can be OR'ed together.

PME_SCAN_PARAMS_RESIDUE-Residue attributes are being set.

PME_SCAN_PARAMS_SRE-SRE attributes are being set.

PME_SCAN_PARAMS_DXE-DXE attributes are being set.

PME_SCAN_PARAMS_PATTERN-Pattern attributes are being set.

p->residue.enable attribute turn residue on or off. If non-zero, residue is on, otherwise residue is off.

p->residue.length attribute value is ignored by this ioctl() but the underlining number of bytes in residue is reset to zero if the *p->residue.enable* to set.

p->sre.sessionid attribute is the index where the per-session context for this Flow will be accessed by SRE in the SRE table. Range: 0x0 ... 0x7FFFFFF.

p->sre.verbose attribute is the PME report verbosity mode. There are 4 levels: 0,1,2 and 3.

p->sre.esee attribute turns on or off the enables the End of SUI event. If turned on the End of SUI Reaction pointer programmed in ESRP (End of SUI Reaction Pointer Register) is to be executed upon End of SUI. A value of zero will turn off this attribute, otherwise it will be on.

p->dxe.clim attribute is the compare limit. Valid range is 0x0 to 0xFFFF. 0xFFFF is treated as infinite.

p->dxe.mlim attribute is the match limit. Valid range is 0x0 to 0xFFFF. 0xFFFF is treated as infinite.

p->pattern.set attribute defines an exclusive grouping of patterns (i.e. no set overlap) that are to be searched simultaneously by the Pattern Matcher. The Pattern Matcher supports 256 (mutually exclusive) pattern sets. When scanning for patterns in the data, the search is restricted to a particular pattern set. Valid range is 0x0 to 0xFF.

p->pattern.subset attribute defines An non-exclusive grouping of patterns (subset overlap permitted) within a given pattern set that are to be searched simultaneously by the Pattern Matcher with or without other subsets. Unlike a pattern set, patterns may be assigned to multiple pattern subsets.The Pattern Matcher supports 16 subsets per pattern set. When scanning for patterns, the search may use a list of subsets. Valid range is 0x0 to 0xFFFF.

Return Value

Returns 0 on success, -1 with errno set upon error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.3.10 PMEIO_GETSCAN

PMEIO_GETSCAN-Retrieves the pme context and scanning attributes.

Synopsis

```
ioctl(fd, PMEIO_GETSCAN, struct pme_scan_params *p);
```

struct pme_scan_params {

```
__u32 flags;
struct pme_scan_params_residue {
    __u8 enable;
    __u8 length;
} residue;
struct pme_scan_params_sre {
    __u32 sessionid;
    __u8 verbose;
    __u8 esee;
} sre;
struct pme_scan_params_dxe {
    __u16 clim;
    __u16 mlim;
} dxe;
struct pme_scan_params_pattern {
    __u8 set;
    __u16 subset;
```

```
            } pattern;

        };
```

Description

This ioctl() gets the flow context and scan related attributes. This is a blocking API.

p->flags is ignored by the ioctl().

p->residue.enable attribute indicates if residue is turn on or off. A value of zero indicates off. A value of non-zero indicates on.

p->residue.length attribute indicates the current number of bytes in residue. This attribute has no meaning when residue is disabled.

p->sre.sessionid attribute is the index where the per-session context for this Flow will be accessed by SRE in the SRE table.

p->sre.verbose attribute is the PME report verbosity mode.

p->sre.esee attribute indicates the state of the End of SUI event. If turned on (non zero) the End of SUI Reaction pointer programmed in ESRP (End of SUI Reaction Pointer Register) is to be executed upon End of SUI. A value of zero indicates off.

p->dxe.clim attribute is the compare limit.

p->dxe.mlim attribute is the match limit.

p->pattern.set attribute defines an exclusive grouping of patterns (i.e. no set overlap) that are to be searched simultaneously by the Pattern Matcher. The Pattern Matcher supports 256 (mutually exclusive) pattern sets. When scanning for patterns in the data, the search is restricted to a particular pattern set.

p->pattern.subset attribute defines An non-exclusive grouping of patterns (subset overlap permitted) within a given pattern set that are to be searched simultaneously by the Pattern Matcher with or without other subsets. Unlike a pattern set, patterns may be assigned to multiple pattern subsets.The Pattern Matcher supports 16 subsets per pattern set. When scanning for patterns, the search may use a list of subsets.

Return Value

Returns 0 on success and * *p* is updated accordingly, -1 with errno set upon error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

EINVAL

The parameters specified are not valid

See Also

7.4.6.3.7.4 PME Kernel API

This describes the kernel API of the PME driver.

7.4.6.3.7.4.1 pme_ctx_init

pme_ctx_init-Initializes a pme context for performing scan or control operations

Synopsis

```
        int pme_ctx_init(
```

```
                struct pme_ctx *ctx, u32 flags, u32 bpid, u8 qosin, u8 qosout,

                enum qm_channel dest, const struct qm_fqd_stashing *stashing);
```

Description

The pme_ctx_init function is used to initialize a new pme context that can be used for performing pattern matching or control operations on the pattern matching device.

The ctx structure will be initialized. The ctx->cb field is required to be set prior to invoking this api.

flags defines context attributes and is a bit mask of the following values:

PME_CTX_FLAG_LOCKED-When either frame queues state are updated, this flag indicates if a spin_lock is used during the update. If this flag is specified, the spin_lock is acquired. This is relavent if different cores will be updating the frame queues state.

PME_CTX_FLAG_EXCLUSIVE-When set, causes the input frame queue to remain in a parked state when the context is enabled. Furthermore, when a frame is enqueued on the input frame queue, the PME's exclusive frame queue control will be enabled. Hence, the context input frame queue is the PME's exclusive frame queue for this enqueue. This mode is only available if the driver is built with control functionality and if the OS has access to the PME's CCSR map.

PME_CTX_FLAG_PMTCC-A pme_context operates in two modes: scan and pmtcc mode. If this flag is specified then pmtcc mode is selected, otherwise scan mode. The mode determines which operations on the pme_context are not permitted. In scan mode pme_ctx_pmtcc() operations are not permitted. In pmtcc mode, pme_ctx_scan(), pme_ctx_ctrl_update_flow() and pme_ctx_ctrl_read_flow() are not permitted.

PME_CTX_FLAG_DIRECT-When in scan mode, a pme_context may have a flow context allocated. If this flag is specified, no flow context is allocated, ortherwise one is. When in direct mode, the following operations are not permitted: pme_ctx_ctrl_update_flow(), pme_ctx_ctrl_read_flow().

PME_CTX_FLAG_LOCAL-Use the current core's dedicated portal channel. The dest argument will be ignored.

The *bpid* value specifies the buffer pool id to be used for any bman generated output.

The *qosin* value specifies the workqueue priority on the PME channel (0-7). Only applies if PME_CTX_FLAG_EXCLUSIVE is not set.

0, 1-High priority

2, 3, 4-Medium Priority

5, 6, 7-Low Priority

The *qsout* value specifies the workqueue priority on the software portal (0-7). Same priorities as described above.

The *stashing* argument configures the desired dequeue stashing behavior. If NULL, no stashing will occur. Refer to the qman api document if non NULL behavior is required.

Return Value

The pme_ctx_init API returns 0 on success, and may return the following error codes:

-ENOMEM if there was not enough memory available to satisfy an allocation request

-EIO

-EBUSY

See Also

pme_ctx_finish()

7.4.6.3.7.4.2 pme_ctx_finish

pme_ctx_finish-Releases all previously allocated resources (e.g. frame queues, memory) from a disabled pme_ctx.

Synopsis

```
        void pme_ctx_finish(struct pme_ctx *ctx);
```

Description

This API is used to indicate a pme_ctx object is no longer required. Once a pme context has been disabled and is no longer required, invoking this api will release all previously allocated resources, such as frame queues, flow context context memory, etc.

See Also

pme_ctx_init()

7.4.6.3.7.4.3 pme_ctx_enable

pme_ctx_enable-Enable a pme_ctx

Synopsis

```
int pme_ctx_enable(struct pme_ctx *ctx);
```

Description

This API is used to change the state of a pme_ctx to the enabled state. The following operations are not permitted while in the enabled state:

pme_ctx_enable()

pme_ctx_reconfigure_tx()

pme_ctx_reconfigure_rx()

pme_ctx_finish()

Return Value

The API will return 0 on success, and may return the following error codes:

-EBUSY

-EINVAL

-EIO

See Also

pme_ctx_disable()

7.4.6.3.7.4.4 pme_ctx_disable

pme_ctx_disable-Disable a pme_context object

Synopsis

```
        int pme_ctx_disable(struct pme_ctx *ctx, u32 flags, struct pme_ctx_ctrl_token
*token);
```

Description

This API disables a previously enabled pme_context.

If the return value is 0, the context is disabled.

If the return value is +1, the context is disabling and the token's completion callback will be invoked when disabling is complete.

Otherwise an error is returned and the context remains enabled.

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in *token* object. The cb (callback) member must be set (i.e. non NULL) when a callback is expected.

Return Value

The API will return 0 on success, +1 when the callback will be invoked or one of the following errors on failure:

-EBUSY

-EINTR

-EIO

See Also

pme_ctx_enable()

pme_ctx_is_disable()

7.4.6.3.7.4.5 pme_ctx_is_disable

pme_ctx_is_disable-query whether a pme context is disabled

Synopsis

```
int pme_ctx_is_disabled(struct pme_ctx *ctx);
```

Description

This API queries whether a pme context is disabled.

Return Value

The API will return >0 if the ctx is disabled. Otherwise 0 is returned is ctx is not disabled.

See Also

pme_ctx_disable()

7.4.6.3.7.4.6 pme_ctx_is_dead

pme_ctx_is_dead-query whether a pme context is dead

Synopsis

```
int pme_ctx_is_dead(struct pme_ctx *ctx);
```

Description

This API queries whether a pme context is dead.

Return Value

The API will return >0 if the ctx is dead. Otherwise 0 is returned is ctx is not dead.

See Also

pme_ctx_is_disable()

7.4.6.3.7.4.7 pme_ctx_reconfigure_tx

pme_ctx_reconfigure_tx-reconfigure transmit frame queue

Synopsis

```
int pme_ctx_reconfigure_tx(struct pme_ctx *ctx, u32 bpid, u8 qosin);
```

Description

This API reconfigure the tx qman frame queue. The pme_ctx must be in a disabled state.

bpid specifies the report buffer pool id that the Pme device shall use when BMan output is generated.

qosin indicates which of the Pme's 8 prioritized workqueues the frame queue should schedule to. This is only applicable if the PME_CTX_FLAG_EXCLUSIVE context flag is not set.

Return Value

The pme_ctx_reconfigure_tx API returns 0 on success, and may return the following error codes:

-EBUSY

-EIO

-EINVAL

See Also

pme_ctx_disable()

7.4.6.3.7.4.8 pme_ctx_reconfigure_rx

pme_ctx_reconfigure_rx-reconfigure receive frame queue

Synopsis

```
int pme_ctx_reconfigure_rx(struct pme_ctx *ctx, u8 qosout, enum qm_channel dest,
const struct qm_fqd_stashing *stashing);
```

Description

This API reconfigure the rx qman frame queue. The pme_ctx must be in a disabled state.

qosout indicates which of the 8 prioritized workqueues the frame queue should be scheduled to on the software portal.

dest specifies which pool channel *or* dedicated channel to use. Ignored if the pme_ctx was initialized with the PME_CTX_FLAG_LOCAL flag.

stashing specifies a stashing object. Refer to qman api for more detail on this object.

Return Value

The pme_ctx_reconfigure_rx API returns 0 on success, and may return the following error codes:

-EBUSY

-EIO

-EINVAL

See Also

pme_ctx_disable()

7.4.6.3.7.4.9 pme_ctx_ctrl_update_flow

pme_ctx_ctrl_update_flow-Updates the associated pme flow context record

Synopsis

```
int pme_ctx_ctrl_update_flow(struct pme_ctx *ctx, u32 flags, struct pme_flow
*params, struct pme_ctx_ctrl_token *token);
```

Description

This API updates the associated pme flow context record in the Pme device according to the values specified in *params*. For instance to enable residue, set the PME_CMD_FCW_RES flags parameter as well as the *params->ren* and *params->rlen* values. To disable residue, set the PME_CMD_FCW_RES flags parameter and set the *params->ren* to 0. The pme_ctx must be in the following state: enabled, flow mode and scan mode(e.g. PME_CTX_FLAG_DIRECT and PME_CTX_FLAG_PMTCC were not specified during initialization).

flags indicates which fields in the flow context are to be updated. These values can be or'ed together with the exception of PME_CTX_OP_RESETRESLEN.

PME_CMD_FCW_RES-Residue related attributes can be updated. These attributes are: *params->ren*, *params->rlen*

PME_CMD_FCW_SEQ-Sequence related attributes can be updated. These attributes are: *params->sos*, *params->seqnum*

PME_CMD_FCW_SRE-Stateful rule related attributes can be updated. These attributes are: *params->srvm, params->esee* and *params->sessionid*

PME_CMD_FCW_DXE-Data Examination related attributes can be updated. These attributes are: *params->clim* and *params->mlim*

PME_CMD_FCW_ALL-All of the above attributes can be updated.

PME_CTX_OP_RESETRESLEN-Applies only to a pme_ctx which has residue enabled. The following attribute is updated: *params->rlen*. This flag should only be used on it's own.

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

params is the pme flow structure. The contents of this structure get copied by this api.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in *token* object.

Return Value

The pme_ctx_ctrl_update_flow API returns 0 on success, and may return the following error codes:

-EBUSY

-EIO

-EINVAL

-ENOMEM

See Also

7.4.6.3.7.4.10 pme_ctx_ctrl_read_flow

pme_ctx_ctrl_read_flow-read the flow record in the Pme device.

Synopsis

```
int pme_ctx_ctrl_read_flow(struct pme_ctx *ctx, u32 flags, struct pme_flow
*params, struct pme_ctx_ctrl_token *token);
```

Description

This API sends a ctrl operation request to read the flow record in the Pme device into the *params* argument. The *ctx* must be in the following state: enabled, flow mode and scan mode(e.g. PME_CTX_FLAG_DIRECT and PME_CTX_FLAG_PMTCC were not specified during initialization).

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

params is a pointer to a flow context object. The params will be updated with upon invocation of the callback specified in the token parameter.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in *token* object.

Return Value

The pme_ctx_ctrl_read_flow API returns 0 on success, and may return the following error codes:

-EINTR

-EBUSY

-EINVAL

See Also

7.4.6.3.7.4.11 pme_ctx_ctrl_nop

pme_ctx_ctrl_nop-sends a Pme nop command

Synopsis

```
int pme_ctx_ctrl_nop(struct pme_ctx *ctx, u32 flags, struct pme_ctx_ctrl_token
*token);
```

Description

This API sends a pme nop command. The *ctx* must be enabled.

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in *token* object.

Return Value

The pme_ctx_ctrl_nop API returns 0 on success, and may return the following error codes:

-EINTR

-EBUSY

See Also

7.4.6.3.7.4.12 cb

cb-Function to invoke when response to a ctrl api (update_flow, read_flow, nop) operation has been returned from Pme hardware.

Synopsis

```
         void (*cb)(struct pme_ctx *ctx, const struct qm_fd *fd, struct pme_ctx_ctrl_token
*token);
void (*ern_cb)(struct pme_ctx *, const struct qm_mr_entry *, struct
pme_ctx_ctrl_token *);
```

Description

This cb function is specified in the pme_ctx_ctrl_token parameter specified in the ctrl apis. When the response to a pme_ctx_ctrl_update_flow(), pme_ctrx_ctrl_read_flow, pme_ctx_ctrl_nop() or pme_ctx_disable() is received this callback is invoked (usually in interrupt context) and appropriate action must be taken by the user. In the case of pme_ctx_disable(), the callback is not invoked if the api returns 0. Since the callback may be in interrupt context user must ensure that proper requirements are met (e.g. sleeping is not permitted, etc).

The ern_cb function is invoked when an enqueue rejection occurs. This may occur before order-restoration or after (eg. if due to congestion or tail-drop). Use the rc code of the mr_entry to determine.

ctx is a pme context structure.

fd is a frame description which has been dequeued from the output frame queue. In other words this is the response from the Pme device.

token is the parameter originally passed in the corresponding pme_ctx_ctrl api. The user is now given ownership of this parameter.

See Also

pme_ctx_ctrl_update_flow()

pme_ctx_ctrl_read_flow()

pme_ctx_ctrl_nop()

pme_ctx_disable()

7.4.6.3.7.4.13 pme_ctx_exclusive_inc

pme_ctx_exclusive_inc-Increment pme exclusivity reference count

Synopsis

```
int pme_ctx_exclusive_inc(struct pme_ctx *ctx, u32 flags);
```

Description

This API aquires Pme exclusivity (if not already held) and increments the reference count in *ctx* . This API can only be invoked on the control plane and only on a ctx that was initialized with the PME_CTX_FLAG_EXCLUSIVE flag. If exclusivity cannot be immediatly acquired the *flags* argument determines whether the API sleeps or not.

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

Return Value

The pme_ctx_exclusive_inc API returns 0 on success, and may return the following error codes:

-ENODEV

-EBUSY

See Also

pme_ctx_exclusive_dec()

7.4.6.3.7.5 pme_ctx_exclusive_dec

pme_ctx_exclusive_dec-Decrement pme exclusivity reference count

Synopsis

```
void pme_ctx_exclusive_dec(struct pme_ctx *ctx);
```

Description

This API decrements the exclusivity reference count and if this is the last reference also releases Pme exclusivity. This API can only be invoked on the control plane and only on a *ctx* that was initialized with the PME_CTX_FLAG_EXCLUSIVE flag.

Return Value

No return value.

See Also

pme_ctx_exclusive_inc()

7.4.6.3.7.6 pme_scan_cb

pme_scan_cb-Function to invoke when response to scan or pmtcc operation has been returned from Pme hardware.

Synopsis

```
        typedef void (*pme_scan_cb)(struct pme_ctx *ctx, const struct qm_fd *fd, struct
pme_ctx_token *token);
```

Description

This callback function is specified prior to the pme_ctx_init() API as part of the *ctx->cb* parameter. When the response to a pme_ctx_scan() or pme_ctx_pmtcc() is received this callback is invoked (usually in interrupt context) and appropriate action must be taken by the user. Since the callback maybe in interrupt context user must ensure that proper requirements are met (e.g. sleeping is not permitted, etc).

ctx is a pme context structure.

fd is a frame description which has been dequeued from the output frame queue. In other words this is the response from the Pme device.

token is the parameter originally passed in the corresponding pme_ctx_scan() or pme_ctx_pmtcc() commands. The user is now given ownership of this parameter. Typically a user may "outcast" this pointer to a larger object (i.e. this token is embedded within a user defined structure which contains additional context data).

See Also

pme_ctx_init()

pme_ctx_scan()

pme_ctx_pmtcc()

7.4.6.3.7.7 pme_scan_ern_cb

pme_scan_ern_cb-Function to invoke when enqueue rejection occurs during a scan or pmtcc operation.

Synopsis

```
        typedef void (*pme_scan_ern_cb)(struct pme_ctx *ctx, const struct qm_mr_entry
*mr, struct pme_ctx_token *token);
```

Description

This error rejection notification callback function is specified prior to the pme_ctx_init() API as part of the *ctx->ern_cb* parameter. When an error rejection notification is received this callback is invoked (usually in interrupt context) and appropriate action must be taken by the user. Since the callback may be in interrupt context user must ensure that proper requirements are met (e.g. sleeping is not permitted, etc).

mr is an ern maessage response structure. The contained ern structure contains the rejection code (*rc*) and the associated frame descriptor (*fd*) which was rejected.

token is the parameter originally passed in the corresponding pme_ctx_scan() or pme_ctx_pmtcc() commands. The user is given ownership of this parameter. Typically a user may "outcast" this pointer to a larger object (i.e. this token is embedded within a user defined structure which contains additional context data).

See Also

pme_ctx_init()

pme_ctx_scan()

pme_ctx_pmtcc

7.4.6.3.7.8 PME_SCAN_ARGS

PME_SCAN_ARGS-Macro that modifies the frame description. Used in the pme_ctx_scan() API

Synopsis

```
        #define PME_SCAN_ARGS(flags, set, subset)
```

Description

Modify a frame descriptor for a pme_ctx_scan api (only modifies *fd->cmd* field).

flags is a bit mask of the following values:

PME_CMD_SCAN_SRVM(n)-Scan Report Verbosity Mode. Where n = {0..3}. This flag is ignored by the Pme device when the context is configured for direct mode.

PME_CMD_SCAN_FLUSH-Instructs the Pme device to flush any cached Flow Context and Residue data to system memory after the Frame is processed. This flag is ignored by the Pme device when the context is configured for direct mode.

PME_CMD_SCAN_SR-Depends on the pme ctx mode and possibly on the state of the flow context. Direct mode-Start of Flow indication. The first data byte of the Frame will be treated as the Start of Flow for anchored pattern matching purposes. Flow mode-Depends if residue is enabled or not. residue disabled: Start of Flow indication. The first data byte of the Frame will be treated as the Start of Flow for anchored pattern matching purposes. residue enabled: Flow Context Reset. The Start of Flow, Sequence Number and Residue Length fields in the Flow Context Record will be reset to 0x0 prior to scanning the Frame. Empty input Frames are accepted and may be used as a mechanism to reset Flow Context without scanning a Frame

PME_CMD_SCAN_E-End of Flow. Depends on the ctx mode. Direct mode: Indicates that the last symbol of the scanned work unit is to be considered an End of Flow. Flow mode: Indicates that the last byte of input data will be considered as the end of Flow. As well, the Flow Context's sequence number and residue length context are reset to zero such that the next data byte on the Flow will be considered a Start of Flow. In order to accommodate protocols such as TCP, where it may not be known until later that the last processed byte was in fact the End of Flow byte, Frames with zero length input data but with the End of Flow indicator set are allowed. In these cases, the session's residue will be recycled through the Pattern Matcher with the End of Flow indication such that any anchored patterns present won't be missed.

set is the pattern set value. An exclusive grouping of patterns (i.e. no set overlap) that are to be searched simultaneously by the Pattern Matcher. The Pattern Matcher supports 256 (mutually exclusive) pattern sets. When scanning for patterns in the data, the search is restricted to a particular pattern set.

subset is the pattern subset value. A non-exclusive grouping of patterns (subset overlap permitted) within a given pattern set that are to be searched simultaneously by the Pattern Matcher with or without other subsets. Unlike a pattern set, patterns may be assigned to multiple pattern subsets. The Pattern Matcher supports 16 subsets per pattern set. When scanning for patterns, the search may use a list of subsets.

See Also

pme_ctx_scan()

7.4.6.3.7.9 pme_ctx_scan

pme_ctx_scan-sends a Pme scan command

Synopsis

```
int pme_ctx_scan(struct pme_ctx *ctx, u32 flags, struct qm_fd *fd, u32 args,
struct pme_ctx_token *token);
```

Description

Send a pme scan command. The *ctx* must be enabled and not in the pmtcc mode. If *ctx.flags* indicate exclusivity then Pme exclusivity is acquired before the request is sent. The API returns zero upon successful enqueue of the *fd*. The corresponding *ctx.cb* function is invoked when the response to the scan request has been dequeued. The *ctx.cb* function may be invoked before this api has completed. The *ctx.cb* is invoked in interrupt context (when applicable).

ctx is an enabled pme context that is in scan mode (e.g. The *ctx* was not initialized with PME_CTX_FLAG_PMTCC).

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

fd is a frame description as defined in the Pattern Matcher Block Guide.

args parameter is produced by the PME_SCAN_ARGS() macro.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in pme ctx object. A caller will typically embed this token object as part of a larger object so as to maintain his own per command data.

Return Value

The pme_ctx_scan API returns 0 on success, and may return the following error codes:

-EINTR

-EBUSY

See Also

pme_ctx_enable()

PME_SCAN_ARGS()

7.4.6.3.7.10 pme_ctx_scan_orp

pme_ctx_scan_orp-sends a Pme scan command with order restoration

Synopsis

```
int pme_ctx_scan_orp(struct pme_ctx *ctx, u32 flags, struct qm_fd *fd, u32 args,
struct pme_ctx_token *token, struct qman_fq *orp_fq, u16 seqnum);
```

Description

This extends the pme_ctx_scan() API to provide order restoration support. The *orp_fq* represents the frame queue descriptor that is to be used as the order restoration point and the *seqnum* is the sequence number to use for order restoration.

ctx is an enabled pme context that is in scan mode (e.g. The *ctx* was not initialized with PME_CTX_FLAG_PMTCC).

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

fd is a frame description as defined in the Pattern Matcher Block Guide.

args parameter is produced by the PME_SCAN_ARGS() macro.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in pme ctx object. A caller will typically embed this token object as part of a larger object so as to maintain his own per command data.

orp_fq is the order restoration point frame queue to be used.

seqnum is the sequence number to be used for order restoration.

Return Value

The pme_ctx_scan_orp API returns 0 on success, and may return the following error codes:

-EINTR

-EBUSY

See Also

pme_ctx_enable()

PME_SCAN_ARGS()

7.4.6.3.7.11 pme_ctx_pmtcc

pme_ctx_pmtcc-sends a Pme pmtcc command

Synopsis

```
int pme_ctx_pmtcc(struct pme_ctx *ctx, u32 flags, struct qm_fd *fd, struct
pme_ctx_token *token);
```

Description

Send a Pme pmtcc command. The *ctx* must be enabled and have been initialized with the PME_CTX_FLAG_PMTCC flag. The corresponding *ctx->cb* function is invoked when the response to the pmtcc request has been received. The *ctx->cb* function may be invoked before this api has completed. The *ctx->cb* is invoked in interrupt context (when applicable). The token parameter is returned via the *ctx->cb* function.

This API will attempt to acquire exclusivity if the *ctx* doesn't already have it. Exclusivity can only be acquired on the control plane.

ctx is an enabled pme context that is in pmtcc mode (e.g. The *ctx* was initialized with PME_CTX_FLAG_PMTCC).

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

fd is a frame description as defined in the Pattern Matcher Block Guide.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in pme ctx object. A caller will typically embed this token object as part of a larger object so as to maintain his own per command data.

Return Value

The pme_ctx_pmtcc API returns 0 on success, and may return the following error codes:

-EINTR

-EBUSY

See Also

pme_ctx_enable()

pme_ctx_init()

7.4.6.3.7.12 pme_attr_set

pme_attr_set-Write to a Pme attribute

Synopsis

```
int pme_attr_set(enum pme_attr attr, u32 val);
```

Description

This API permits the setting of certain Pme attributes as defines by the enum pme_attr set. This API is only available on the control plane.

attr is the Pme attribute to write to.

val is the value to be written

Return Value

The pme_attr_set API returns 0 on success, and may return the following error codes:

-ENODEV

See Also

pme_attr_get()

7.4.6.3.7.13 pme_attr_get

pme_attr_get-Reads from a Pme attribute

Synopsis

```
int pme_attr_get(enum pme_attr attr, u32 *val);
```

Description

This API permits the reading of certain Pme attributes as defines by the enum pme_attr set. This API is only available on the control plane.

attr is the Pme attribute to read.

val is the value read

Return Value

The pme_attr_get API returns 0 on success, and may return the following error codes:

-ENODEV

See Also

pme_attr_set()

7.4.6.3.7.14 pme2_have_control

pme2_have_control-Query if there is access to the Pme CCSR register space

Synopsis

```
int pme2_have_control(void);
```

Description

This API return >0 is there is access to the Pme CCSR register space (i.e. caller is on the control plane). Otherwise 0 is returned. Some apis require the caller to have access to CCSR space (such as pme_attr_get()), this api permits a caller to determine is CCSR space is accessible.

Return Value

The pme2_have _control API returns 0 the caller does not have CCSR access. Otherwise a non zero value is returned.

See Also

7.4.6.3.7.15 pme_stat_get

pme_stat_get-Query a Pme statistics attribute

Synopsis

```
int pme_stat_get(enum pme_attr attr, u64 *value, int reset);
```

Description

This api returnes an accumulated version of the related attribute. At a configured interval, the Pme driver will query all statistic attributes and maintain an accumulated counter (i.e. the value read is added to the current count). This api also permits resetting the accumulated counter. This API is only available on the control plane.

attr is one of

pme_attr_trunci

pme_attr_rbc

pme_attr_tbt0ecc1ec

pme_attr_tbt1ecc1ec

pme_attr_vlt0ecc1ec

pme_attr_vlt1ecc1ec

pme_attr_cmecc1ec

pme_attr_dxcmecc1ec

pme_attr_dxemecc1ec

pme_attr_stnib

pme_attr_stnis

pme_attr_stnth1

pme_attr_stnth2

pme_attr_stnthv

pme_attr_stnths

pme_attr_stnch

pme_attr_stnpm

pme_attr_stns1m

pme_attr_stnpmr

pme_attr_stndsr

pme_attr_stnesr

pme_attr_stns1r

pme_attr_stnob

pme_attr_mia_byc

pme_attr_mia_blc

value is updated to the current accumulated count.

reset indicates is the accumulated count is to be reset after updating value. A value of 0 will not reset the counter. A value of non-zero will reset the counter. In either case the current accmulated count is returned.

This API returns 0 in success.

Return Value

-ENODEV

-EINVAL

See Also

pme2_have_control()

7.4.6.3.8 Control interface

This chapter provides a description of the API of the Pattern Matcher control interface (PMCI).

PMCI module is a Linux user space library that provides C functional interface to send and receive Pattern Matcher control commands to the Pattern Matcher through Pattern Matcher driver software. These commands allow users to program and monitor the Pattern Matcher database on the hardware. The PMCI converts complex PME control commands into PME driver primitives.

In general, PME control interface is used in the following order:

Initialize a PMCI object using pmci_open().

Use pmci_write() to write single or multiple PME control commands. These commands follow the Pattern Matcher protocol (PMP) format.

Use pmci_read() to read responses from commands that generate responses.

Use pmci_close() to close the PMCI object.

The table below lists the files that constitute the PMCI package.

PMCI files

libpmci.a	PMCI functionality
pmci.h	Defines the interface to the PMCI module
pmci.c	Implementation of the PMCI module
genTypes.h	Contains the generic type definitions and is included by pmci.h
pmDefs.h	Defines the interface that is common to PMLL, PMREC, PMSRC, etc., and is included by pmci.h
pmp.h	Contains the definition of the Pattern Matcher Protocol and is included by pmci.h

7.4.6.3.8.1 pmci_open

Initialize a new instance of PME Control Interface channel, and return the handle.

Synopsis

```
#include <pmci.h>
pmci_error_t pmci_open(int channel, handle_t *handle)
```

Description

This function opens a new PME Control Interface channel and returns its handle. Such a handle is required on any subsequent PME Control Interface operations.

channel specifies the DMA channel to use for communication with hardware. The possible values are from between 0 and 3.

handle specifies the variable where the new handle value is returned.

Return Value

handle points to the new PMCI handle if open was successful.

pmci_success_e is returned for a successful open, otherwise one of the following errors is returned:

pmci_unavailable_driver_e

Part of required driver modules not loaded.

pmci_invalid_channel_e

DMA channel was not configured.

7.4.6.3.8.2 pmci_set_option

Apply option to PMCI handle to modify its behaviour.

Synopsis

#include <pmci.h>

```
pmci_error_t pmci_set_option(handle_t pmci_handle, pmci_option_id_t optionId, void
*option, int optionSize)
```

Description

This function provides the ability to change the behavior of the PMCI handle by applying options. The supported options are defined by the pmci_option_id_t type. Currently, the only available option are pmci_option_timeout_e which changes the timeout behaviour for pmci_read(), and pmci_option_batch_buffer_threshold_e which adjust how much buffer memory can be used when batch attribute is enabled.

pmci_handle is the PMCI handle to be modified.

optionId is the behaviour to be changed.

option is the new value of the option

optionSize is the size of *option*, in number of bytes.

Return Value

Returns pmci_success_e for successful operation, otherwise one of the following error codes is returned:

pmci_invalid_handle_e

PMCI handle is invalid.

pmci_invalid_option_code_e

Option id is invalid.

pmci_unavailable_option_e

Option id is invalid.

pmci_invalid_parameters_e

Option value is invalid.

pmci_invalid_option_size_e

Option size is invalid.

7.4.6.3.8.3 pmci_get_option

Retrive the option value previously set by pmci_set_option.

Synopsis

#include <pmci.h>

```
pmci_error_t pmci_get_option(handle_t pmci_handle, pmci_option_id_t optionId, void
*option, int *optionSize)
```

Description

This function provides the ability to retrieve the current values of options used by the PMCI *handle*. These are either values set by the user or default values set upon initialization.

pmci_handle is the PMCI handle

optionId is the identifier id of the option being queried.

option is the returned value of the option being queried.

optionSize is the returned size of *option*, in number of bytes.

Return Value

Returns pmci_success_e upon successful completion, otherwise returns on of the following errors. Upon return, *option will contain the retrieved option value, and *optionSize will contain the size of the value in bytes.

pmci_invalid_handle_e

PMCI handle is invalid.

pmci_invalid_option_code_e

Option id is invalid.

pmci_unavailable_option_e

Option id is invalid.

pmci_invalid_parameters_e

option or optionSize pointer is invalid.

7.4.6.3.8.4 pmci_close

Shut down a PME Control Interface channel.

Synopsis

#include <pmci.h>

```
pmci_error_t pmci_close (handle_t pmci_handle)
```

Description

This function shuts down a PME Control Interface. During the close all resources related to *pmci_handle* will be released.

pmci_handle is the handle of the control interface to be shut down.

Return Value

Returns pmci_success_e for a successful close, otherwise returns one of the following error codes:

pmci_invalid_handle_e

Invalid PMCI handle.

pmci_failure_e

Failed to release some resources.

7.4.6.3.8.5 pmci_write

Write one or more commands to the PME Control Interface.

Synopsis

#include <pmci.h>

```
pmci_error_t pmci_write(handle_t pmci_handle, void *cmds, int cmdsSize)
```

Description

This function is used to write one or more commands to the PME control device. These commands follow the PME command protocol (PMP) format. The PMCI writes these commands into control descriptors created by the PME driver for communicating to the hardware.

Some PME control commands are sent directly to hardware, and some will be handled by PMCI software. A software command may be expanded into multiple hardware commands inside PMCI.

pmci_handle is the PMCI handle in use.

cmd is a pointer to a buffer containing one or more PME commands (in PMP format)..

cmdSize is the total size of all PME commands, in bytes, being sent to the PMCI.

Return Value

Returns pmci_success_e upon successful completion, otherwise returns one of the following error codes:

pmci_invalid_handle_e

PMCI handle is invalid.

pmci_invalid_attribute_id_e

Invalid PM control command.

pmci_failure_e

Invalid PM control command or hardware failure.

pmci_invalid_parameters_e

command buffer is set to NULL, or cmdsSize is 0

7.4.6.3.8.6 pmci_read

Read a notification from PME Control Interface.

Synopsis

#include <pmci.h>

```
pmci_error_t pmci_read(handle_t pmci_handle, pmp_msg_t *notif)
```

Description

This function is used to read command notifications from the PME Control Interface. PME command protocol contains commands that generate a notification. These notifications can be read by calling this function.

pmci_handle is the PMCI handle in use.

notif points to the user defined notification buffer where notifications will be returned. The buffer size should be the size of pmp_msg_t which is defined in pmp.h

Return Value

Returns pmci_success_e upon successful completion, otherwise it returns on of the following error codes:

pmci_invalid_handle_e

PMCI handle is invalid.

pmci_empty_read_e

Read timed out and no notification was read.

pmci_failure_e

Invalid PM control command or hardware failure.

pmci_invalid_parameters_e

notification buffer pointer is set to NULL

7.4.6.3.8.7 pmci_flush

Flush the PME Control Interface and ensure no more data is in flight.

Synopsis

#include <pmci.h>

```
pmci_error_t pmci_flush(handle_t pmci_handle)
```

Description

This function provides the ability to flush the PME driver pipe and ensure no more data is in flight. The call blocks until the PME driver has ensured its command pipeline is empty and that all preceeding commands have not only reached the hardware but has also completed execution.

pmci_handle is the PMCI handle in use.

Return Value

Returns pmci_success_e upon successful completion, otherwise returns one of the following errors:

pmci_invalid_handle_e

PMCI handle is invalid.

pmci_lost_driver_e

PM driver did not perform the operation.

pmci_failure_e

Hardware failure.

7.4.6.3.8.8 pmci_context_clear_by_session_id

Clear the stateful rule context associated to a specific session Id.

Synopsis

#include <pmci.h>

```
pmci_error_t pmci_context_clear_by_session_id(uint32_t sessionId)
```

Description

This utility function provides the ability to clear the stateful rule context associated to a specific session Id. The PME hardware provides the functionality to clear its stateful rule context for a particular session. This function provides users the means to invoke that functionality. It opens a PMCI handle, sends the hardware command to clear the context, and then close the PMCI handle.

session_id is the session id whose associated context information is to be cleared.

Return Value

Returns pmci_success_e upon successful completion, otherwise returns one of the following errors:

pmci_unavailable_driver_e

Driver not loaded or configured.

pmci_failure_e

Driver not configured properly, or HW failure.

pmci_invalid_parameters_e

Bad session id.

7.4.6.3.8.9 pmci_error_string

Returns a string describing the PMCI error code.

Synopsis

#include <pmci.h>

```
const char *pmci_error_string(pmci_error_t code)
```

Description

This utility function returns a printable string corressponding to the pmci error code.

code is the error code returned from other PMCI functions.

Return Value

Returns a string pointer to an appropiate error message.

7.4.6.3.9 PMP message format

The Pattern Matcher protocol (PMP) defines the format of messages that are used to setup and manage the PME database. The following sections describe in detail the various PMP message formats.

7.4.6.3.9.1 PME Message Syntax

This section summarizes the PMP message syntax. Each PMP message has the following format:

Version (8 bits)-version number of the PME control protocol

Type (8 bits)-command type code

Reserved (16 bits)-for future use

Length (32 bits)-total length in bytes, including header bytes

Message ID (64 bits)-command message sequence number, helps to co-relate the response message with the command message

Command (variable)-control command information, specific to each type of message

Commands and Notifications Types

7.4.6.3.9.2 PMP Message Types

There are various types of PMP messages. These messages vary in format and behavior based on their type. Some messages are one-way only and represent commands to the PME whereas other messages expect a response message or notification from the PME.

The following table lists the various types of messages.

Summary of Numerical Assignment of Type Field

Category	Type	Description[14]

Table continues on the next page...

[14] The descriptions that contain a code in parenthesis have the following legend: C = Command, N = Notification, and V = Virtualized (not directly supported by the hardware). Also note that read and get notifications uses the same type as the commands except the most significant bit is set to differentiate between the request and the reply.

Table continued from the previous page...

Table Manipulation	0x00	Read Table Entry (CN)[15]
	0x01	Write Table Entry (C)
	0x02	Reset All Table Entries (V)
Session Context Manipulation	0x08	Clear Session Contexts by Session ID (C)
	0x09	Clear Session Contexts by Rule ID (V)
	0x0c	Clear All Session Contexts (V)
Attribute Manipulation	0x10	Get Attribute (VN)
	0x11	Set Attribute (V)
Debugging	0x1f	Error Indication (N)
Reply Types	0x80-0xff	Reserved for replies (most-significant bit)

7.4.6.3.9.3 TID Information

The read, write and reset messages of PMP require the specification of table identifier (TID), index, and specific entry data sizes. The table below specifies the details of these values.

Table Identifier, Index, and Data Sizes

Table Id	Table Id Name	Start Index	End Index	Entry Data Size
0	pmp_one_byte_trigger_table_id_e	0	0	32
1	pmp_two_byte_trigger_table_id_e	0	511	8
2	pmp_variable_trigger_table_id_e	0	4095	8
3	pmp_confidence_table_id_e	0	18944	4
4	pmp_confirmation_table_id_e	0	65535	128
5	pmp_userDefinedGroupTableId_e	0	0	256
6	pmp_equivalence_table_id_e	0	0	256
7	pmp_session_context_table_id_e	0	1073741823 (B)	32
8	pmp_special_trigger_table_id_e	0	0	32
(A) The number of confirmation entries is configurable.				
(B) The number of session context entries is configurable and relative to the number of sessions and context size.				

[15] Designates commands that map one to one with the PMI commands supported by PME hardware.

7.4.6.3.9.4 PMP message format summary

A summary of all PMP message formats follows. Note that all formats are expressed in network order (known as big-endian byte order). The length field represents the length of the overall command including the type, length, and message identifier.

Additionally, write messages must have data aligned on a word boundary (4-byte), that the table identifier (TID) implicitly indicates the correct data field size. Although the hardware supports multiple write requests, the current version of the protocol does not support this mode. Note that if padding bytes are required, they are placed at the end of the data. Refer to Complete Description of All PMP Commands and Notifications on page 982, for a detailed description.

7.4.6.3.9.4.1 Complete Description of All PMP Commands and Notifications

Read Table Entry

Command	V	T		Length	MsgId	TID	Index			
Notification	V	T		Length	MsgId	TID	Index	Data	. . .	

The command representing the read table entry request has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x00 (pmp_table_read_request_msg_type_e)

Length = 24 (pmp_table_read_request_msg_size_d)

MsgId = Any 64-bit value

TID = Any valid table identifier listed in the TID Information section/>

Index = Any valid index falling in the appropriate range defined in the TID Information section

The notification representing the read table entry reply has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x80 (pmp_table_read_reply_msg_type_e)

Length = 24 + entry data size as defined in the TID Information section/> (see pmp.h for macros defining these)

MsgId = Same 64-bit value as provided by the command

TID = Any valid table identifier listed in the TID Information section/>

Index = Any valid index falling in the appropriate range defined in the TID Information section/>

Data = Whichever value located in the PM hardware table at specified index

The purpose of this function is to retrieve the content of the PME hardware table located at a specific index of the specified table. As a command, it represents the read request. As a notification, it represents the read reply (the response with the relevant data). Mostly used as debugging and diagnostics.

7.4.6.3.9.4.1.1 Write Table Entry

Command	V	T		Length	MsgId	TID	Index	Data	. . .

Table continues on the next page...

Table continued from the previous page...

Notification		INVALID					

The command representing the write table entry request has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x01 (pmp_table_write_request_msg_type_e)

Length = 24 + entry data size as defined in Table 1 (see pmp.h for macros defining these)

MsgId = Any 64-bit value

TID = Any valid table identifier listed in the TID Information section/>

Index = Any valid index falling in the appropriate range defined in the TID Information section/>

Data = Whichever value to store in the PM hardware table at specified index

There are no notifications associated with this function.

The purpose of this function is to store a desired value inside the PME hardware table located at a specific index of the specified table. As a command, it represents the read request. This command does not generate any notifications.

7.4.6.3.9.4.1.2 Reset All Table Entries

Command	V	T		Length	MsgId	TID		
Notification		INVALID						

The command representing the reset all table entries request has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x02 (pmp_table_reset_request_msg_type_e)

Length = 20 (pmp_table_reset_request_msg_size_d)

MsgId = Any 64-bit value

TID = Any valid table identifier listed in the TID Information section/>

There are no notifications associated with this function.

The purpose of this function is to reset all entries of the specified table with zeros. As a command, it represents the reset request. This command does not generate any notifications.

7.4.6.3.9.4.1.3 Clear Session Contexts by Session ID

Command	V	T		Length	MsgId	SessionId	RuleCap	

Table continues on the next page...

Table continued from the previous page...

Notification		INVALID	

The command representing the clear session context by session ID request has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x08 (pmp_ctx_by_session_id_clear_request_msg_type_e)

Length = 24 (sizeof(pmp_ctx_by_session_id_clear_request_msg_t))

MsgId = Any 64-bit value

SessionId = Any 32-bit supported by the system (between 0 and (number of session-1))

RuleCap = 0

There are no notifications associated with this function.

This function clears the digest of the specified session identifier. As a command, it represents the reset request. This command does not generate any notifications.

Each session comprises of a digest, flags and rule context areas. The digest indicates which rule context areas are cleared and which others are not. When resetting context by session identifier, only the digest really needs to be cleared as this invalidates the content of the rule areas.

7.4.6.3.9.4.1.4 Clear Session Context by Rule ID

Command	V	T		Length	MsgId	FirstSesId	SesDepth	NumOfSes	RuleId 0	RuleId 1	. . .	RuleId n
Notification		INVALID										

The command representing the clear session context by rule ID request has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x09 (pmp_ctx_by_rule_id_clear_request_msg_type_e)

Length = Variable based on the number of rules to clear

MsgId = Any 64-bit value

FirstSesId = 0 (first session identifier is determined by the hardware)

SesDepth = 0 (session depth is determined by the hardware)

NumOfSes = 0 (number of session is determined by the hardware)

RuleId0 --RuleIdn = Rule identifiers to clear

There are no notifications associated with this function.

This function clears the bits of the session digest representing the rule identifiers listed in the message for all session identifiers described by the message. As a command, it represents the reset request. This command does not generate any notifications.

Each session comprises of a digest, flags and rule context areas. The digest indicates which rule context areas are cleared and which others are not. When resetting context by rule identifier, only the digest bit(s) of each session really needs to be cleared as this invalidates the corresponding rule areas.

7.4.6.3.9.4.1.5 Clear All Session Contexts

Command	V	T		Length	MsgId			
Notification		INVALID						

The command representing the clear all session contexts request has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x0c (pmp_ctx_all_clear_request_msg_type_e)

Length = 16 (pmp_ctx_all_clear_request_msg_size_d)

MsgId = Any 64-bit value

There are no notifications associated with this function.

This function clears the whole digest of all session in the system. As a command, it represents the clear all request. This command does not generate any notifications.

Each session comprises of a digest, flags and rule context areas. The digest indicates which rule context areas are cleared and which others are not. When resetting all context, only the digest of each session really needs to be cleared as this invalidates all rule areas.

7.4.6.3.9.4.1.6 Get Attribute

Command	V	T		Length	MsgId	Attr. ID		
Notification	V	T		Length	MsgId	Attr. ID	Attr. Data	. . .

The command representing the get attribute request has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x10 (pmp_attribute_get_request_msg_type_e)

Length = 20 (pmp_attribute_get_reply_empty_msg_size_d)

MsgId = Any 64-bit value

Attr. ID = Any valid attribute identifier defined by pmp_attribute_id_t

The notification representing the read table entry reply has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x90 (pmp_attribute_get_reply_msg_type_e)

Length = Variable (pmp_attribute_get_reply_msg_size_d(<attributeDataSize>))

MsgId = Same 64-bit value as provided by the command

Attr. ID = Any valid attribute identifier defined by pmp_attribute_id_t

Attr. Data = Representation of the run-time attribute data

The purpose of this function is to retrieve the content of a PME hardware attribute. As a command, it represents the get attribute request. As a notification, it represents the get attribute reply (the response with the relevant data, and is mostly used for initialization, debugging, and diagnostics.

Available Attributes

Attribute ID	Set	Description
pmp_statistics_attr_id_e	Yes	Get PME statistics (set = reset)
pmp_hardware_revision_attr_id_e	No	Get PME hardware revision
pmp_protocol_revision_attr_id_e	No	Get PMP protocol version
pmp_atomic_attr_id_e	Yes	Set to enable exclusive feature
pmp_batch_attr_id_e	Yes	Enable batch. All pmp commands will be buffered, and then executed when batch attribute is cleared.
pmp_sre_end_of_sui_index_attr_id_e	Yes	Set end-of-SUI head block.
pmp_variable_trigger_size_attr_id_e	Yes	Set variable trigger size.
pmp_confidence_chain_max_length_attr_id_e	Yes	Set maximum length of confidence chain.
pmp_sw_database_signature_attr_id_e	Yes	Set signature for SW database.
pmp_drcc_mask_attr_id_e	Yes	Mark for DRCC select
pmp_drcc_selection_attr_id_e	No	Get DXE Pattern Range Counter Configuration.
pmp_extension_block_num_attr_id_e	No	Get number of extension blocks.
pmp_context_max_num_attr_id_e	No	Get maximum number of contexts.
pmp_context_area_size_attr_id_e	No	Get total size of context area.
pmp_max_stateful_rule_num_attr_id_e	No	Get maximum number of stateful rules.

7.4.6.3.9.4.1.7 Set Attribute

Command	V	T		Length	MsgId	Attr. ID	Attr. Data	. . .

Table continues on the next page...

Table continued from the previous page...

Notification		INVALID					

The command representing the set attribute request has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x11 (pmp_attribute_set_request_msg_type_e)

Length = Variable (pmp_attribute_set_request_msg_size_d(<*attributeDataSize*>))

MsgId = Any 64-bit value

Attr. ID = Any valid attribute identifier defined by pmp_attribute_id_t

Attr. Data = Representation of the run-time attribute data

There are no notifications associated with this function.

The purpose of this function is to assign the content of a PME hardware attribute. As a command, it represents the set attribute request. This command does not generate any notifications. Most attributes are used for initialization. See the table in Get Attribute command for available attributes.

7.4.6.3.9.4.1.8 Error Indication

Command		INVALID					
Notification	V	T	Length	MsgId	ErrorId		

This function does not exist as a command.

The notification representing the error indication has the following field assignment:

V = 1 (PMP_CURRENT_VERSION)

T = 0x1f (pmp_error_indication_msg_type_e)

Length = 20 (sizeof(pmp_error_indication_msg_t))

MsgId = Any 64-bit value if can associate the error with a command, zero otherwise

Error = 0xffffffff

The purpose of this function is to notify the managing software about an error caused by the PM hardware or detected by the PMCI or PME driver. It is invalid as a command. As a notification, it represents the error indication to communicate with the managing software.

7.4.7 Decompression Compression Engine (DCE)

7.4.7.1 DCE Drivers Release Notes

Description

This document describes DCE software for the DCE hardware block that is part of the QorIQ family of SoCs. This is a preliminary release of the DCE software. It is expected that the APIs will be updated for future releases.

Linux

The DCE driver software includes a Linux kernel driver. The driver provides a set of kernel level APIs.

The driver includes the following functionality:

DCE Kernel Driver Interface

The DCE kernel driver APIs provide a callback based interface to the DCE. The driver provides APIs to perform either stateless (chunk) based (de)compression or stateful (stream) based (de)compression. The driver internally co-ordinates commands to the DCE and corresponding results from the DCE. The chunk interface is meant for inline (de)compression where each DCE operation is on a complete and independent piece of information. The stream interface is is designed to (de)compress many related pieces of information (e.g. a file).

DCE FLIB interface

The DCE FLIB interface provides a consistent interface to the CCSR registers, the memory defined DMA structures and to the dce_flow software object.

DCE Configuration interface

The DCE configuration interface is an encapsulation of the DCE CCSR register space and the global/error interrupt source. This is expected to be managed only by (and visible to) a control-plane operating system,

DCE User-space Interface

There is a debugfs interface available for device debugging. No other userspace interface is available. Debugfs provides easy access to DCE memory map registers space. See the *DPAA Reference Manual* for the “DCE Individual Register Memory Map”. e.g.

```
0x000 DCE_CFG — DCE configuration
0x03C DCE_IDLE— DCE Idle status Register
0x3F8 DCE_IP_REV_1 — DCE IP Block Revision 1 register
```

Mount debugfs to explore DCE status:

```
mount -t debugfs none /sys/kernel/debug
root@t4240qds:/dev/shm# cat /sys/kernel/debug/dce/ccsrmem_addr
DCE register offset = 0x0
root@t4240qds:/dev/shm# cat /sys/kernel/debug/dce/ccsrmem_rw
DCE register offset = 0x0
value = 0x00000003        <-DCE configuration, x03= Enable. Block is operational, Frame Queues are
consumed.
root@t4240qds:/dev/shm# echo 0x03c > /sys/kernel/debug/dce/ccsrmem_addr
root@t4240qds:/dev/shm# cat /sys/kernel/debug/dce/ccsrmem_rw
DCE register offset = 0x3c
value = 0x00000001        <- DCE Idle status Register, 1 = idle
root@t4240qds:/dev/shm# echo 0x3f8 > /sys/kernel/debug/dce/ccsrmem_addr
root@t4240qds:/dev/shm# cat /sys/kernel/debug/dce/ccsrmem_rw
DCE register offset = 0x3f8
value = 0x0af00101        <-match default value of “0x0AF0_0101”
```

Functionality

Configuration

The DCE device is configured via device-tree nodes and by some compile-time options controlled via Linux's Kconfig system. See the “DCE Kernel Configure Options” section for more info.

Debugfs Interface

The DCE has a debugfs interface available to assist in device debugging. The code can be built either as a loadable module or statically.

Module Loading

The driver can be statically built or as a dynamically loadable module.

DCE Kernel Configure Options

Common Kernel Configure Options	Description
CONFIG_STAGING	Required in order to make “staging” drivers such as DCE available.
CONFIG_FSL_DCE	Required to build DCE support.
CONFIG_FSL_DCE_CONFIG	Compiles in dce device driver support.
CONFIG_FSL_DCE_DEBUGFS	Compiles in support for debugfs interface for the DCE.
CONFIG_FSL_DCE_TESTS	Compiles DCE test code.

Compile-time Configuration Options

The "Kernel Configure Options" above describe the compile-time configuration options for the kernel.

Source Files

Linux

Source Files	Description
drivers/staging/fsl_dce/fsl_dce_chunk.h	The DCE driver APIs for chunk based (de)compression
drivers/staging/fsl_dce/fsl_dce_stream.h	The DCE driver APIs for stream based (de)compression
drivers/staging/fsl_dce/flib/*.*	The DCE flib interface
drivers/staging/fsl_dce/flib/dce_regs.h	The DCE CCSR register macros. Used in conjunction with bitfield_macros.h macros.
drivers/staging/fsl_dce/flib/dce_defs.h	The DCE dma defined memory structures.
drivers/staging/fsl_dce/flib/dce_flow.h	Object which defines the transport mechanism with the DCE engine. This object encompasses the QMan frame queues required to communicate with the DCE. The chunk and stream object use the flow object as a base.
drivers/staging/fsl_dce/dce_debugfs.*	The DCE debugfs interface
drivers/staging/fsl_dce/tests/performance_simple/*.*	Test which demontrates the DCE throughput performance using single input files. Refer to local README file for more details.

Build Procedure

The procedure is a standard SDK build.

Test Procedure

Refer to drivers/staging/fsl_dce/tests/performance_simple/README for detailed descriptions of sample DCE throughput performance test.

Known Bugs, Limitations, or Technical Issues

- The APIs have been tested in the context of the performance test applications.
- It is possible that in future releases additions and or modification to APIs may occur.

Supporting Documentation

1. T4240 QorIQ Advanced Multiprocessing Processor Reference Manual
2. Queue Manager, Buffer Manager API reference Manual

7.5 Enhanced Secured Digital Host Controller (eSDHC)

7.5.1 eSDHC Driver User Manual

Description

The enhanced SD Host Controller(eSDHC) provides an interface between the host system and MMC/SD cards.

The eSDHC supports 1/4-bit data modes and bus clock frequency up to 50MHz

Module Loading

The eSDHC device driver support either kernel built-in or module.

U-boot Configuration

Runtime options

Env Variable	Env Description	Sub option	Option Description
hwconfig	Hardware configuration for u-boot	setenv hwconfig sdhc	Enable esdhc for the kernel

Kernel Configure Options

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`<*>  MMC/SD/SDIO card support --->`
`<*>  MMC block device driver`
`(8)  Number of minors per block device`
`[*]  Use bounce buffer for simple hosts`
	Enables SD/MMC block device driver support

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description
*** MMC/SD/SDIO Host Controller Drivers *** <*> Secure Digital Host Controller Interface support <*> SDHCI platform and OF driver helper [*] SDHCI OF support for the NXP eSDHC controller	Enables NXP eSDHC driver support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_MMC	y/n	n	Enable SD/MMC bus protocol
CONFIG_MMC_BLOCK	y/n	y	Enable SD/MMC block device driver support
CONFIG_MMC_BLOCK_MINORS	integer	8	Number of minors per block device
CONFIG_MMC_BLOCK_BOUNCE	y/n	y	Enable continuous physical memory for transmit
CONFIG_MMC_SDHCI	y/n	y	Enable generic sdhc interface
CONFIG_MMC_SDHCI_PLTFM	y/n	y	Enable common helper function support for sdhci platform and OF drivers
CONFIG_MMC_SDHCI_OF_ESDHC	y/n	y	Enable NXP eSDHC support

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,esdhc'
reg	integer	Required	Register map

Default node:

```
qoriq-esdhc-0.dtsi:
sdhc: sdhc@114000 {
        compatible = "fsl,esdhc";
        reg = <0x114000 0x1000>;
        interrupts = <48 2 0 0>;
        clock-frequency = <0>;
};

For special platform (T1040 as example):
/include/ "qoriq-esdhc-0.dtsi"
        sdhc@114000 {
                compatible = "fsl,t1040-esdhc", "fsl,esdhc";
                fsl,iommu-parent = <&pamu0>;
                fsl,liodn-reg = <&guts 0x530>; /* eSDHCLIODNR */
                sdhci,auto-cmd12;
                rcpm-wakeup = <&rcpm 0x00000080>;
        };
```

```
NOTE: For different platform, the compatilbe can be different.
      And the property "sdhci, auto-cmd12" is option.
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/mmc/host/sdhci.c	Linux SDHCI driver support
drivers/mmc/host/sdhci-pltfm.c	Linux SDHCI platform devices support driver
drivers/mmc/host/sdhci-of-esdhc.c	Linux eSDHC driver

User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name
iozone	IOzone is a filesystem benchmark tool. The benchmark generates and measures a variety of file operations.The benchmark tests file I/O performance for the following operations: Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread ,mmap, aio_read, aio_write.	iozone

Verification in U-boot

The u-boot log:

```
=> mmcinfo
Device: FSL_ESDHC
Manufacturer ID: 3
OEM: 5344
Name: SD02G
Tran Speed: 25000000
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 2032664576
Bus Width: 4-bit
=> mmc read 0 10000 0 1
MMC read: dev # 0, block # 0, count 1 ... 1 blocks read: OK
=> mmc part 0
Partition Map for MMC device 0  --   Partition Type: DOS
Partition     Start Sector     Num Sectors     Type


    1                 16         3970032       b
```

Verification in Linux

Add environment value

```
=> setenv hwconfig sdhc
```

The booting log

```
......
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
mmc0: SDHCI controller on ffe2e000.sdhci-of [ffe2e000.sdhci-of] using PIO
......
mmc0: new SD card at address 87e2
mmcblk0: mmc0:87e2 SD02G 1.89 GiB
 mmcblk0: p1
```

Check the disk

```
~ # fdisk -l /dev/mmcblk0


disk /dev/mmcblk0: 2032 MB, 2032664576 bytes

63 heads, 62 sectors/track, 1016 cylinders

Units = cylinders of 3906 * 512 = 1999872 bytes

Device Boot      Start       End      Blocks  Id System

/dev/mmcblk0p1      1        1016     1984217   b Win95 FAT32   ~ #
```

Mount the file system and operate the card.

```
~ #

~ # mkdir /mnt/sd

~ # mount -t vfat /dev/mmcblk0p1 /mnt/sd

~ # ls /mnt/sd/

vim

~ # cp /bin/busybox /mnt/sd

~ # ls /mnt/sd

busybox  vim

~ # umount /mnt/sd

~ # mount -t vfat /dev/mmcblk0p1 /mnt/sd
```

```
~ # ls /mnt/sd

busybox  vim

~ #
```

Benchmarking

```
~ #

~ # # iozone -Rab ./iosdresult/result -i 0 -i 1 -f test -n

512M -g 1G -r 64K

Iozone: Performance Test of File I/O

Version $Revision: 3.263 $

Compiled for 32 bit mode.

Build: linux-arm



Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins

Al Slater, Scott Rhine, Mike Wisner, Ken Goss

Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,

Randy Dunlap, Mark Montague, Dan Million,

Jean-Marc Zucconi, Jeff Blomberg,

Erik Habbinga, Kris Strecker, Walter Wong.



Run began: Wed Feb 16 20:33:04 2011



Excel chart generation enabled

Auto Mode

Using minimum file size of 524288 kilobytes.

Using maximum file size of 1048576 kilobytes.

Record Size 64 KB

Command line used: iozone -Rab ./iosdresult/result -i 0 -i 1 -f test -n 512M -g 1G -r 64K

Output is in Kbytes/sec
```

```
Time Resolution = 0.000005 seconds.

Processor cache size set to 1024 Kbytes.

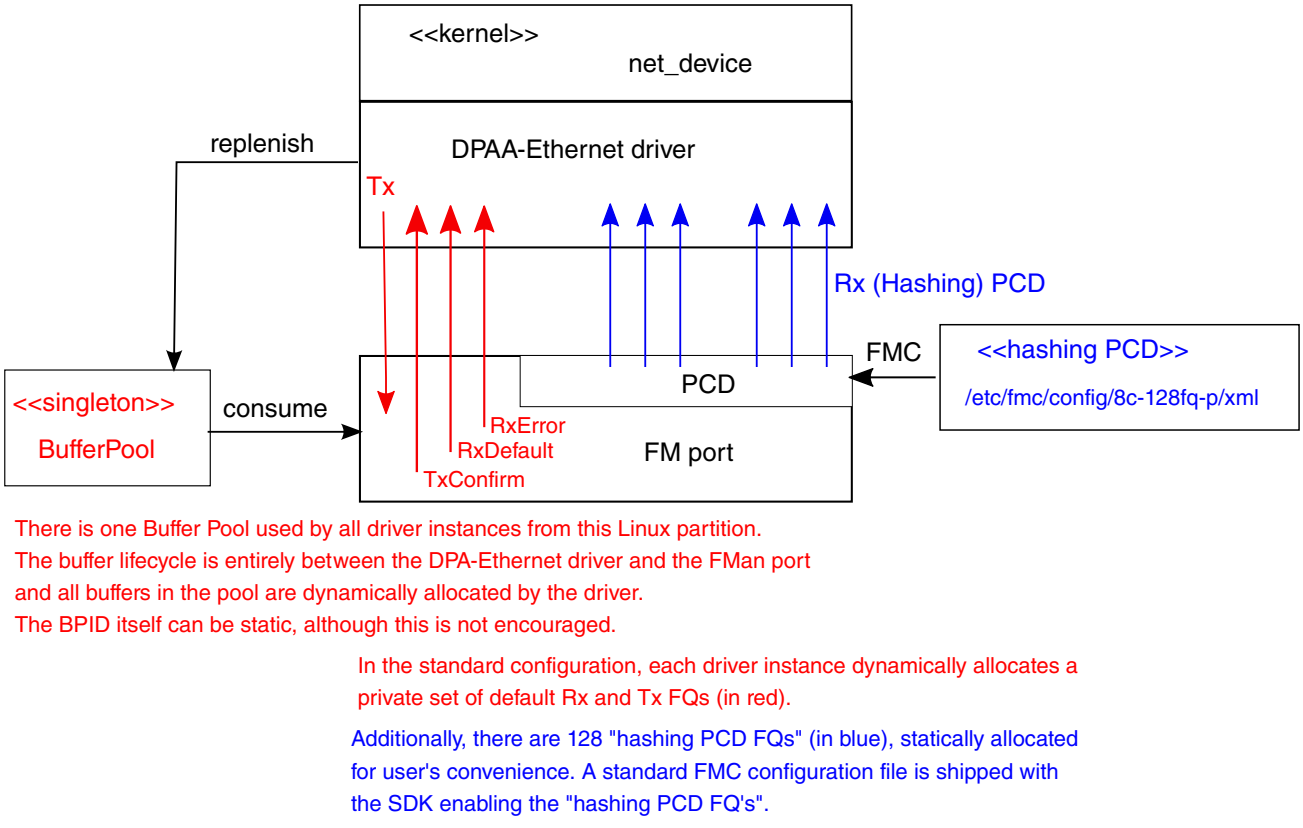
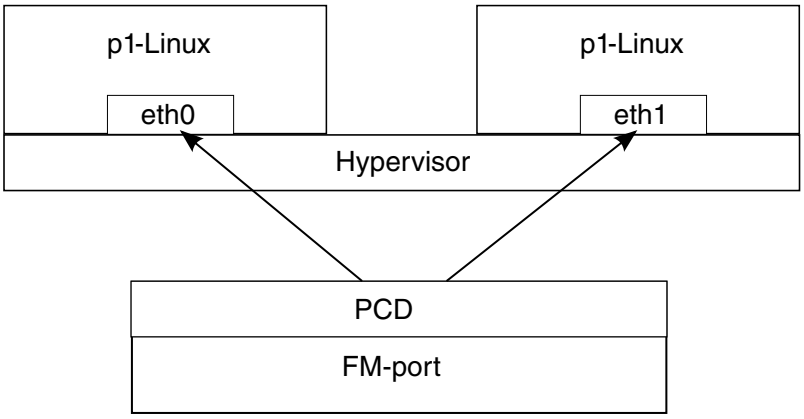
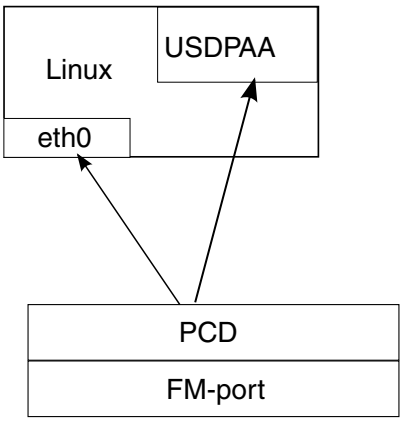
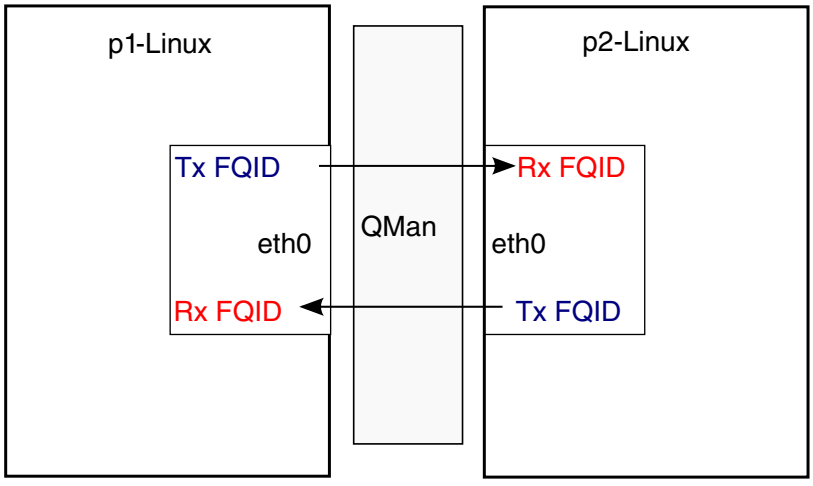
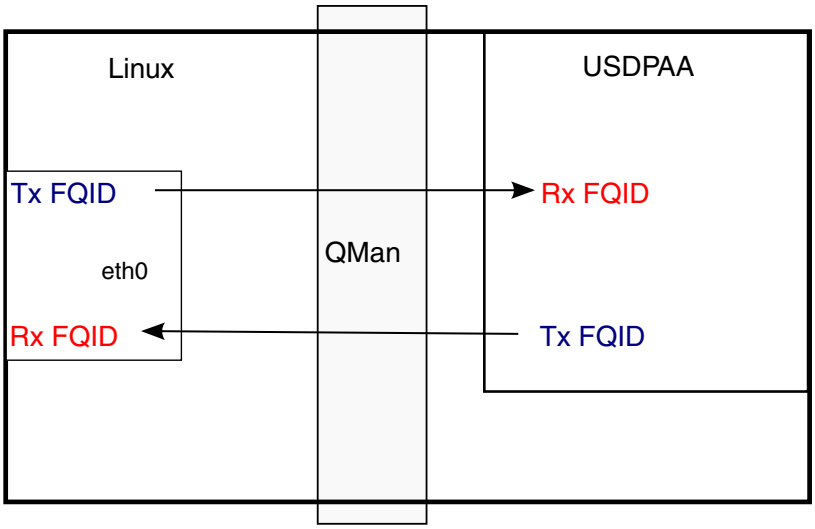
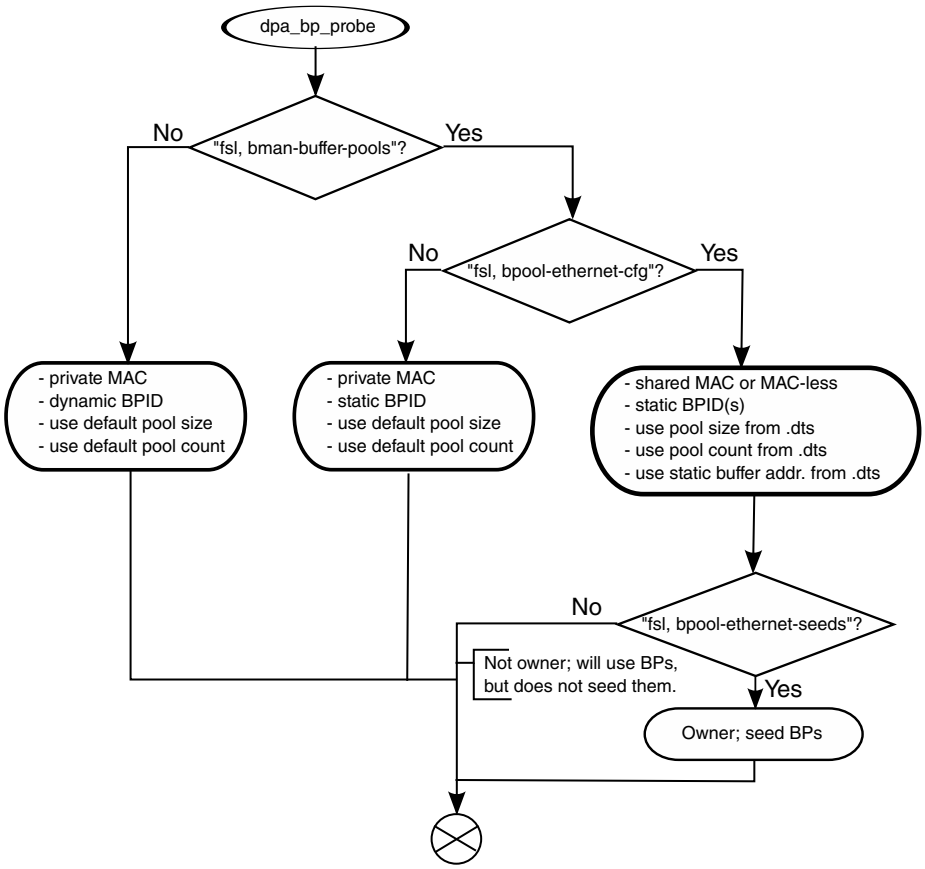
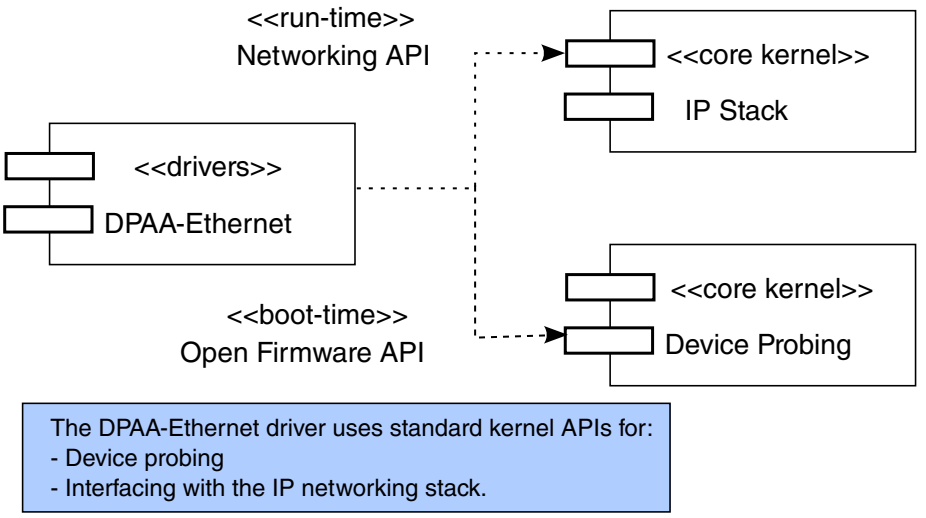
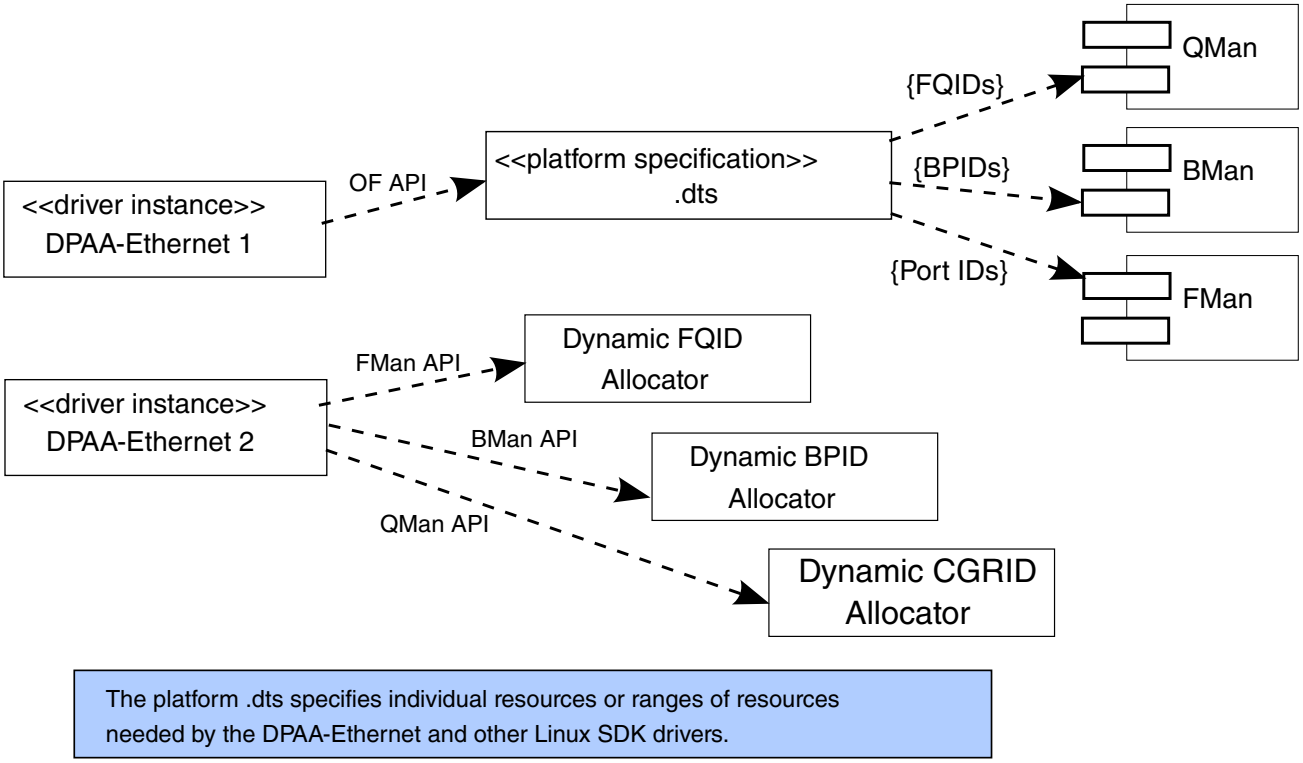
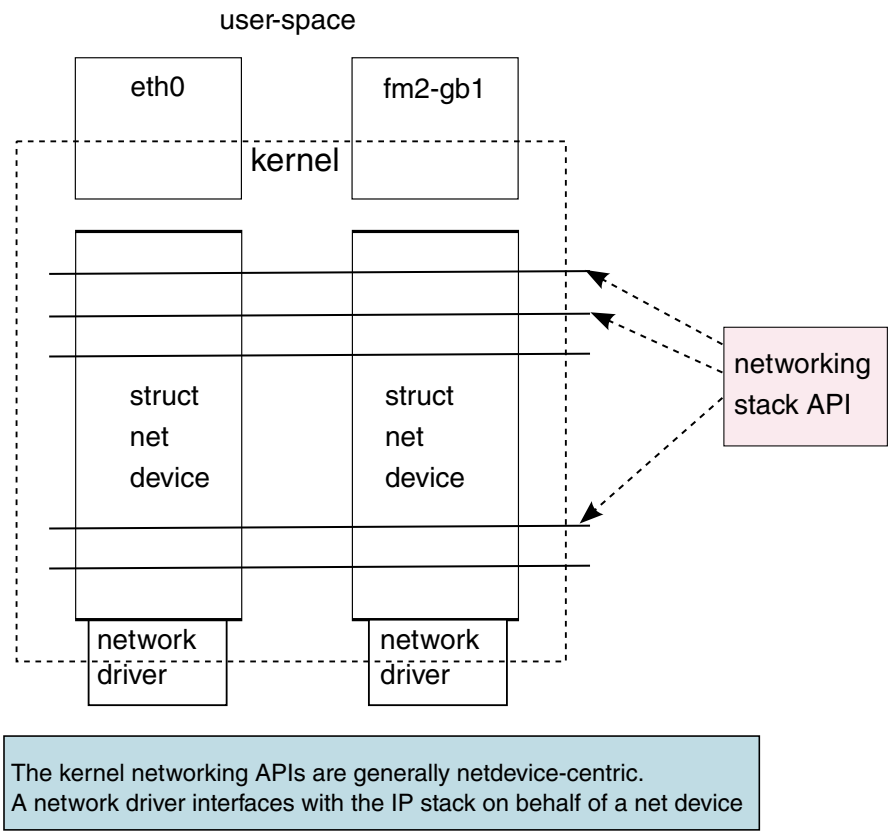
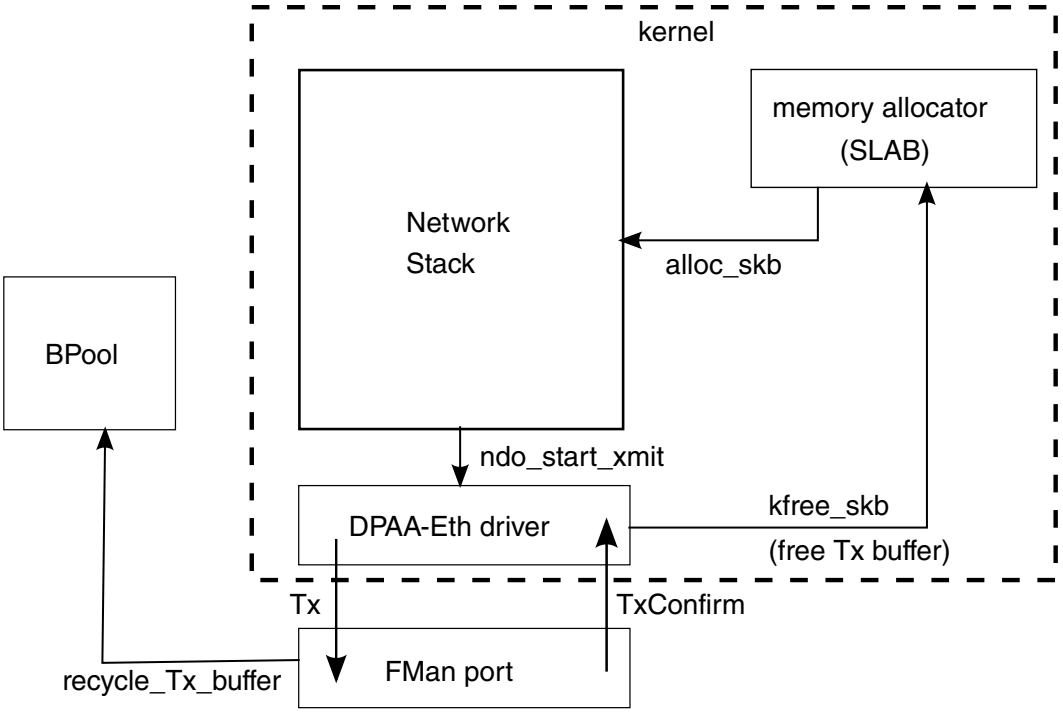
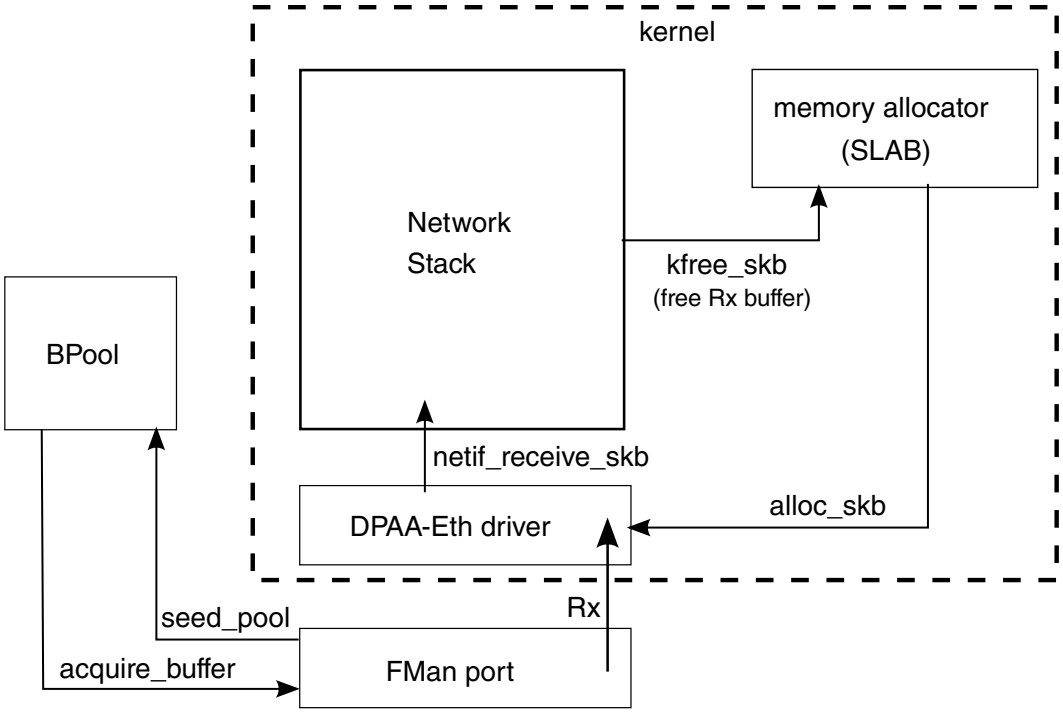
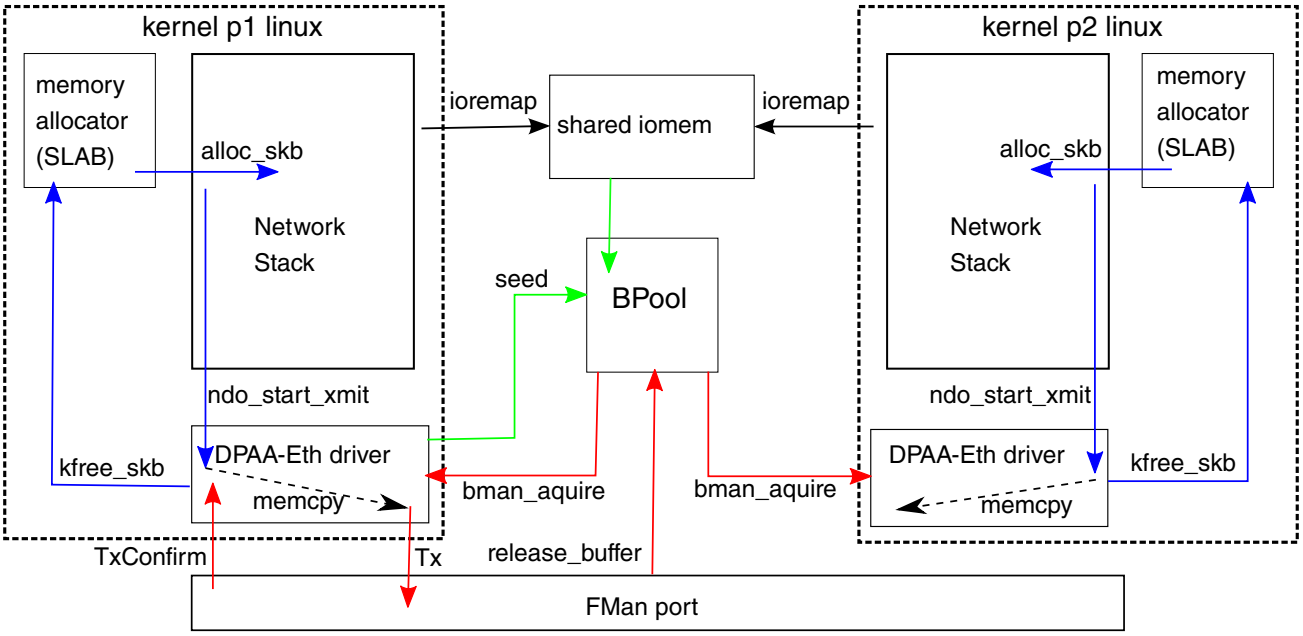
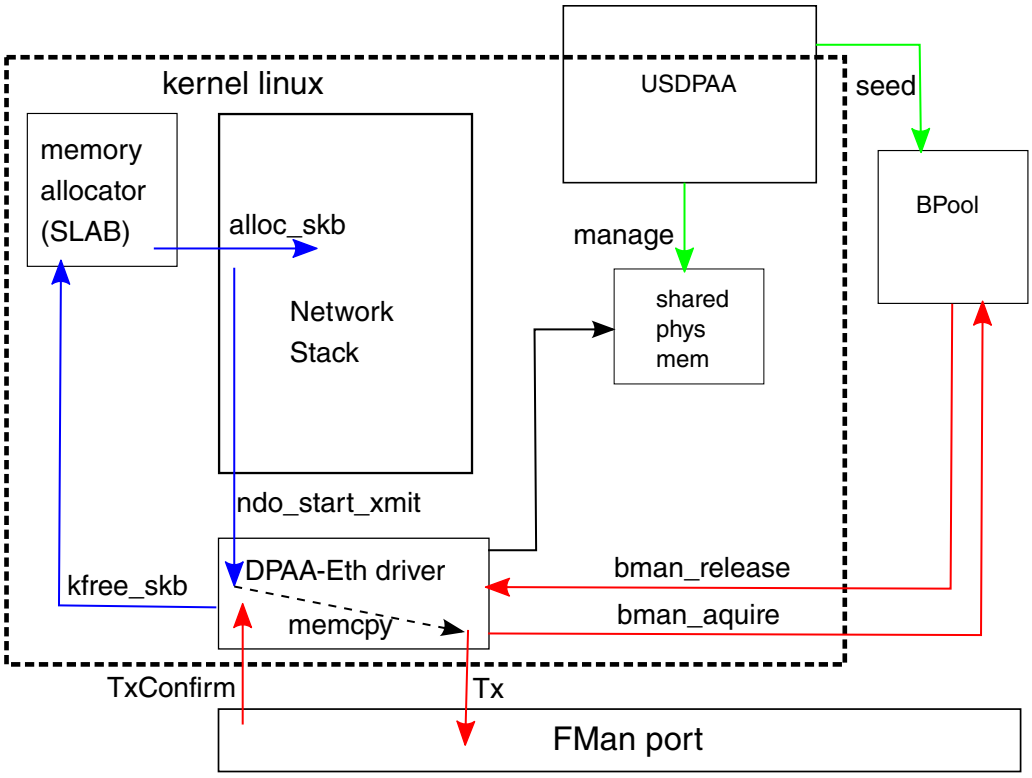
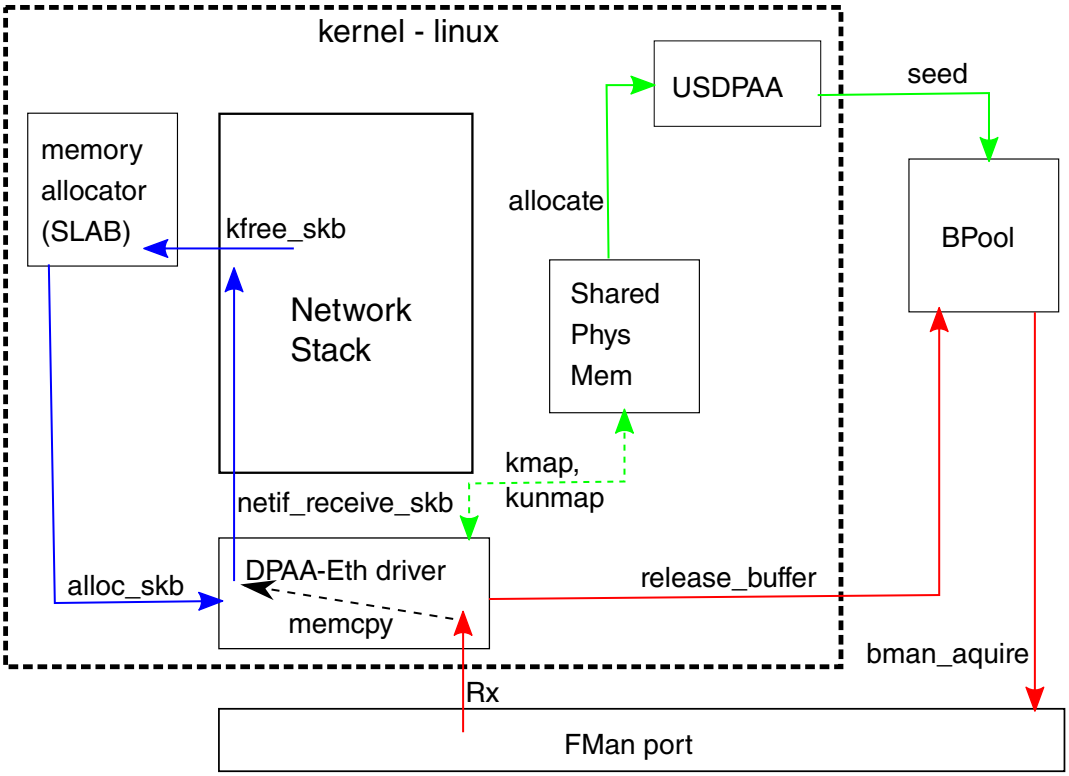
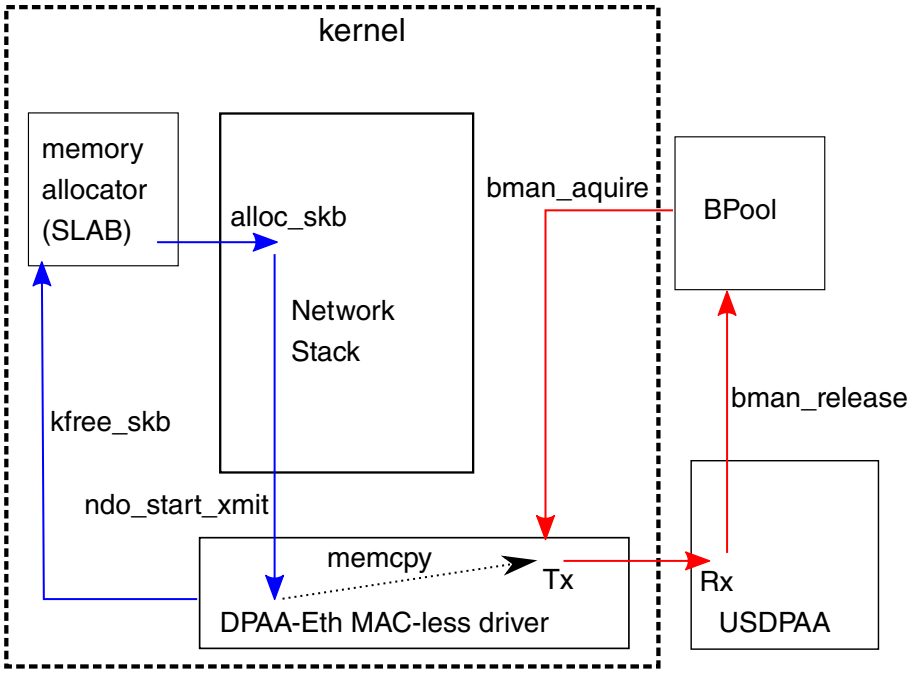
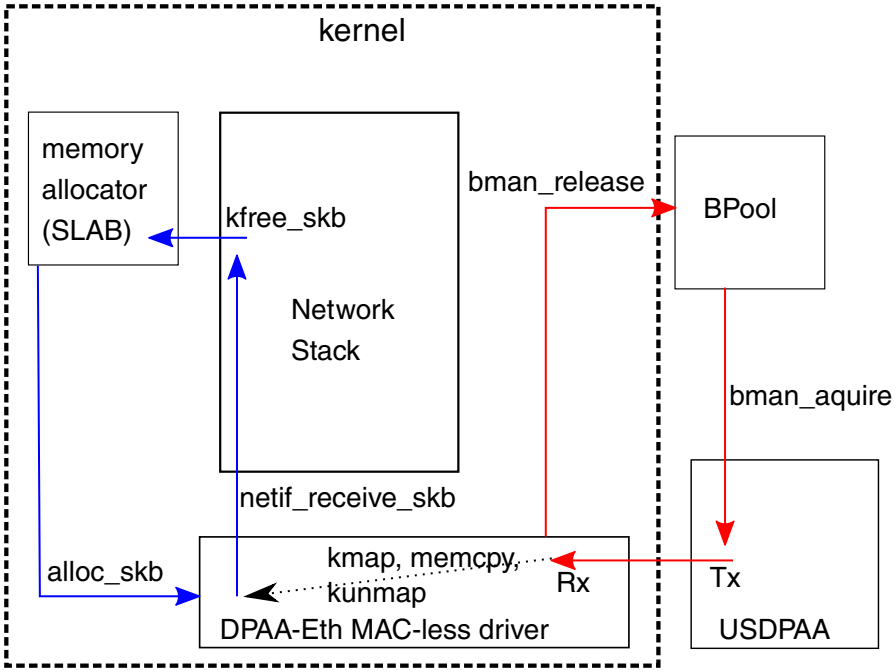
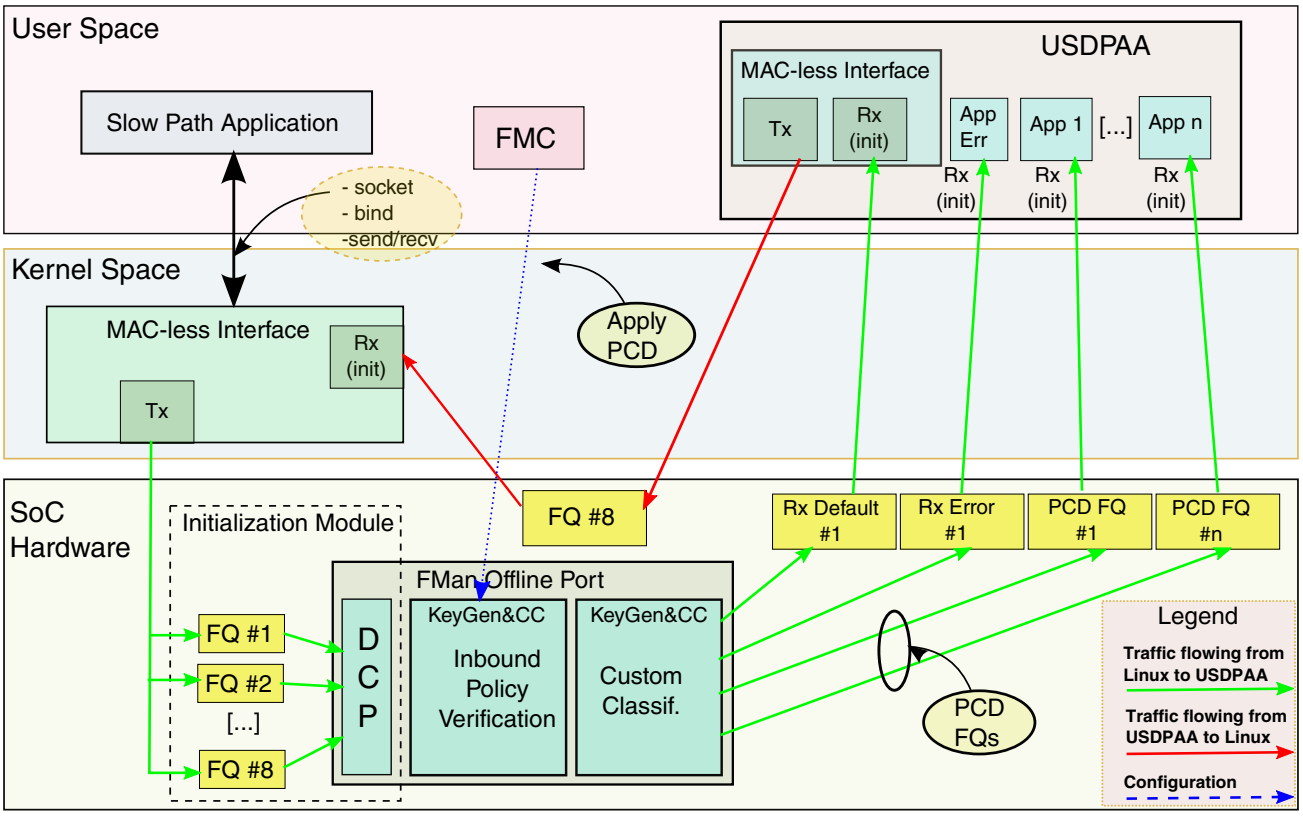
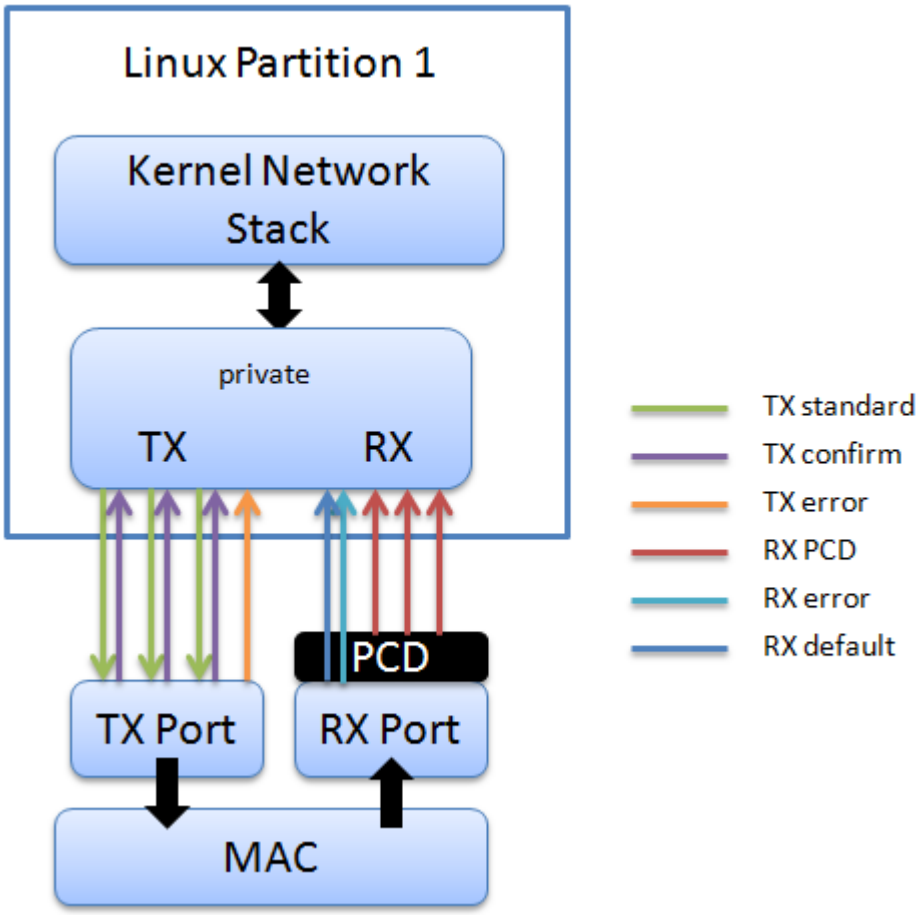
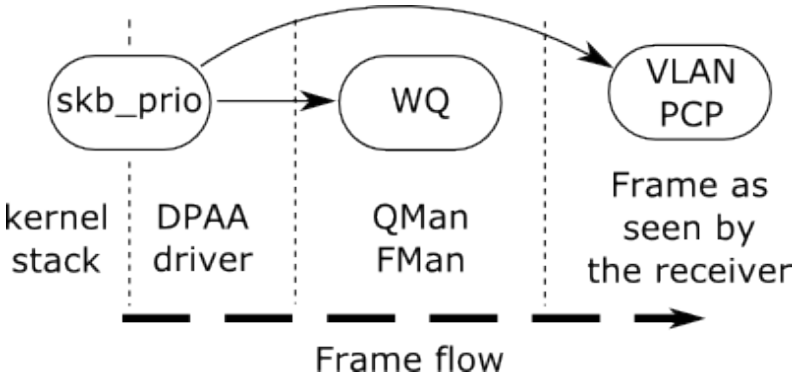
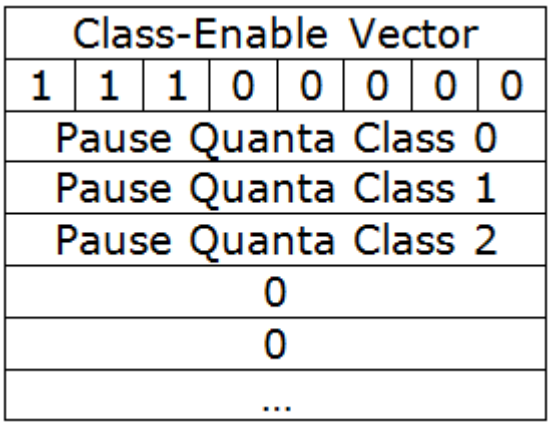
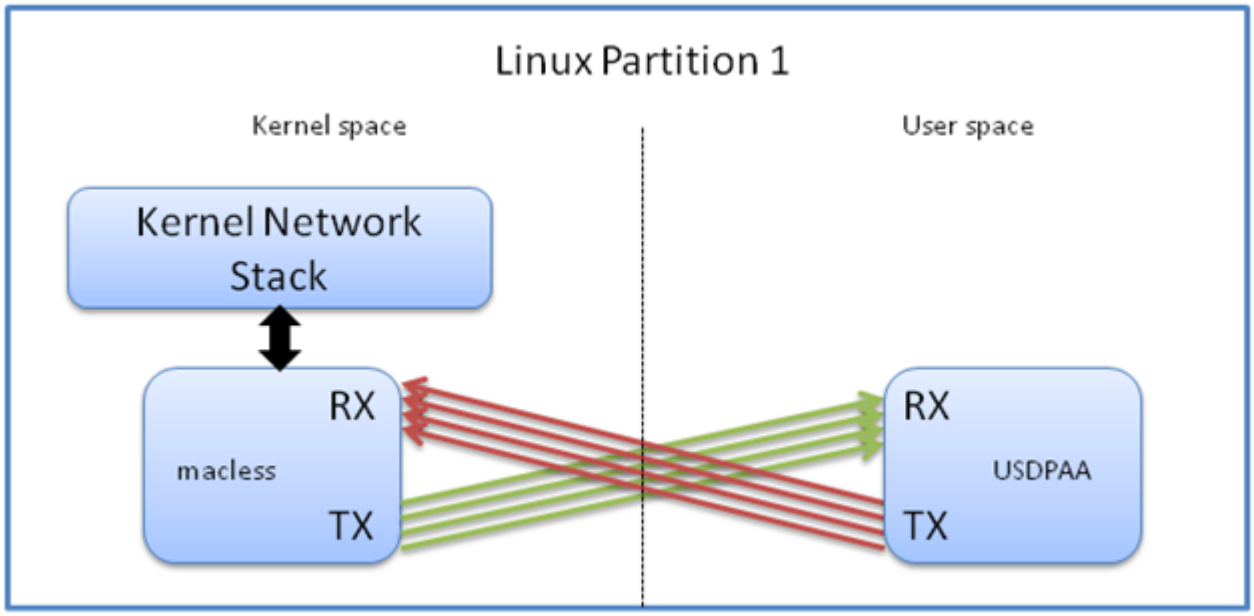
Processor cache line size set to 32 bytes.

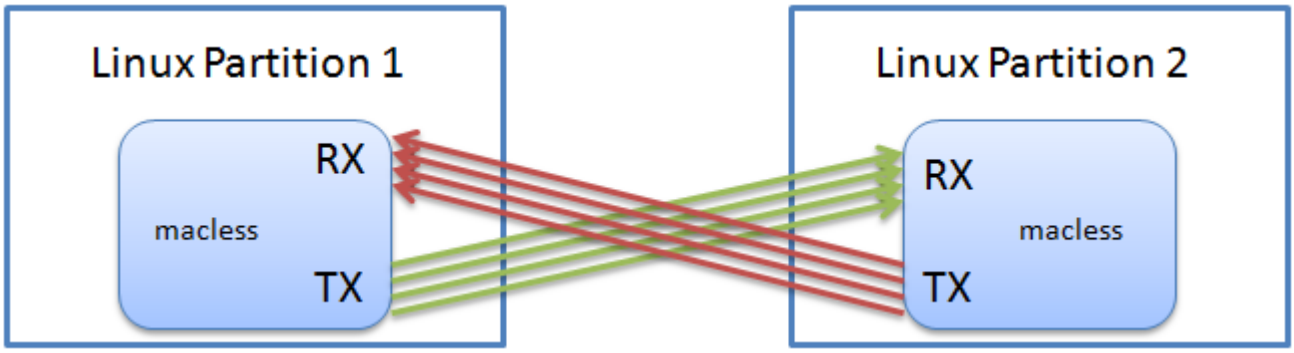
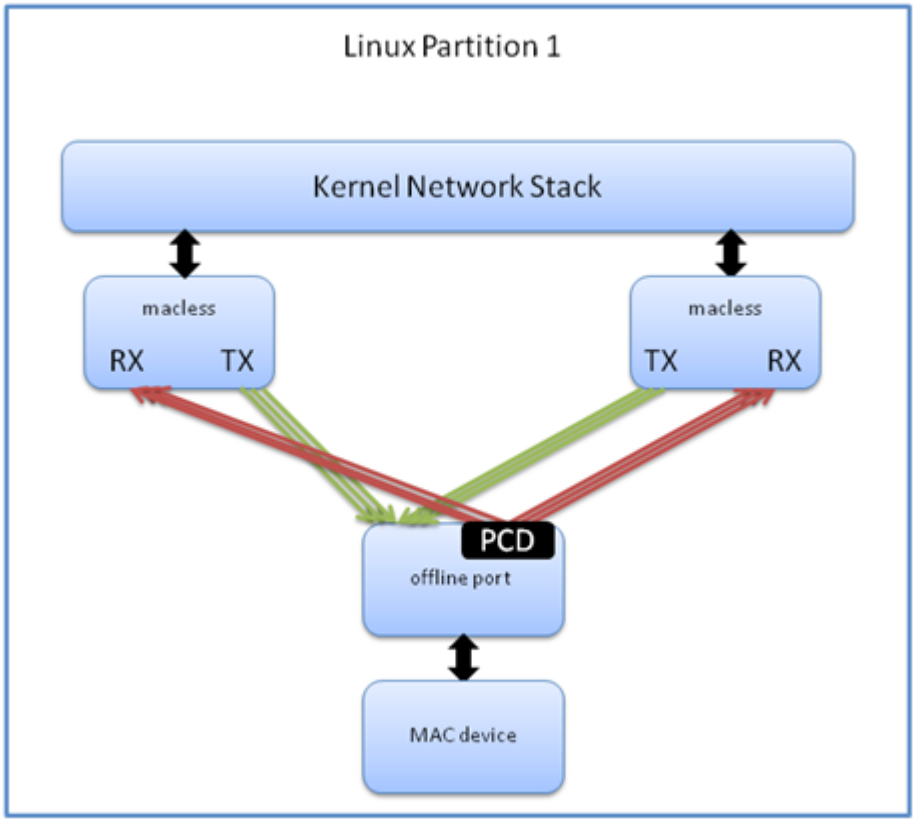
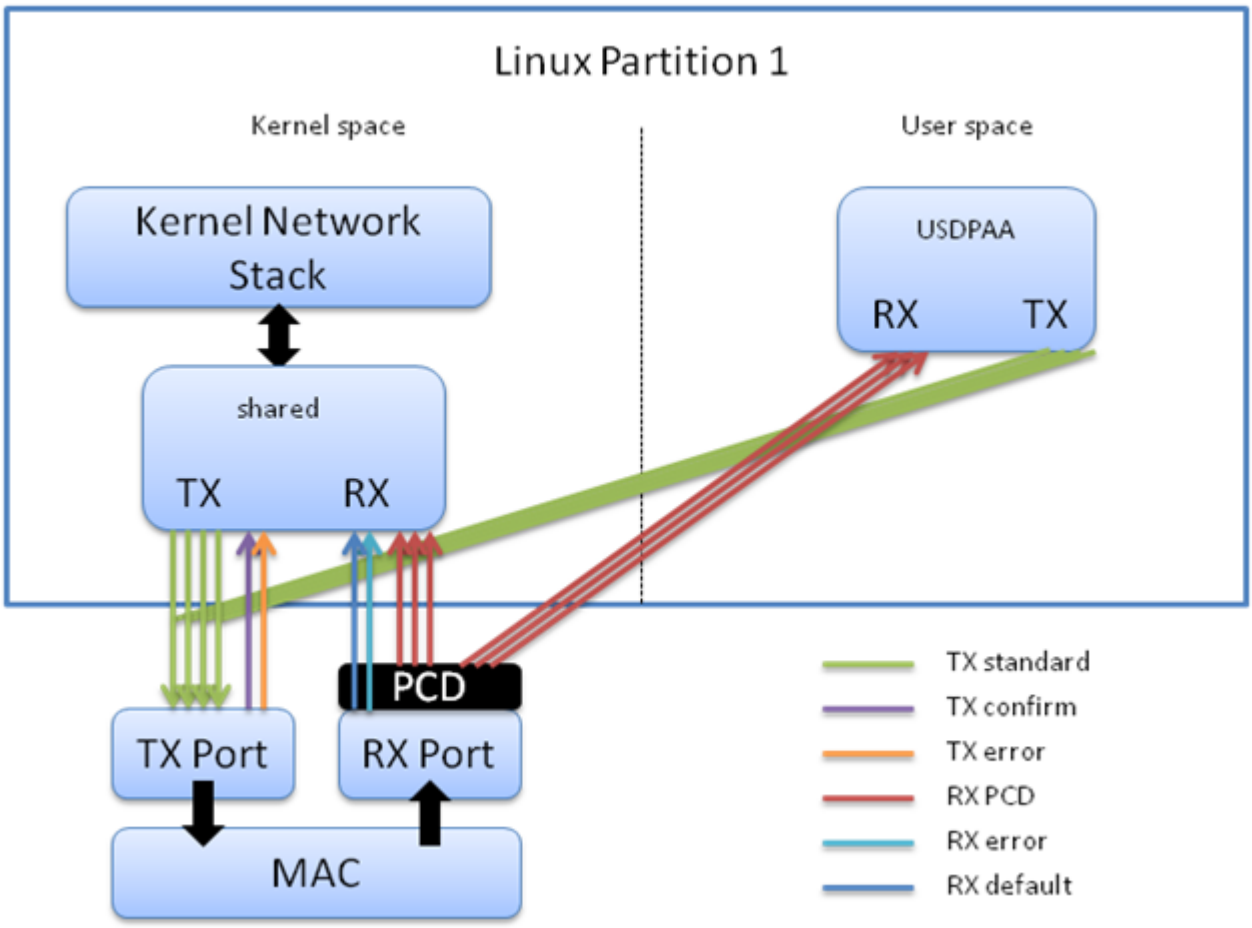
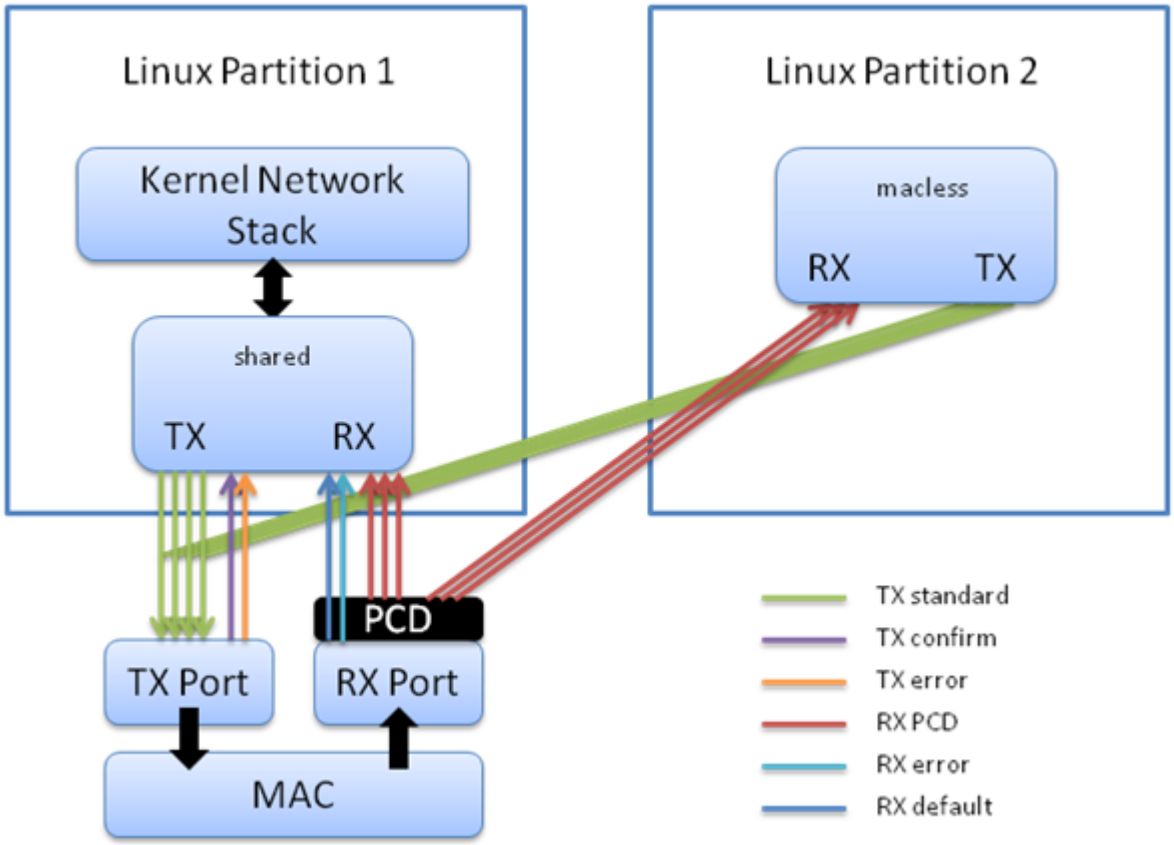
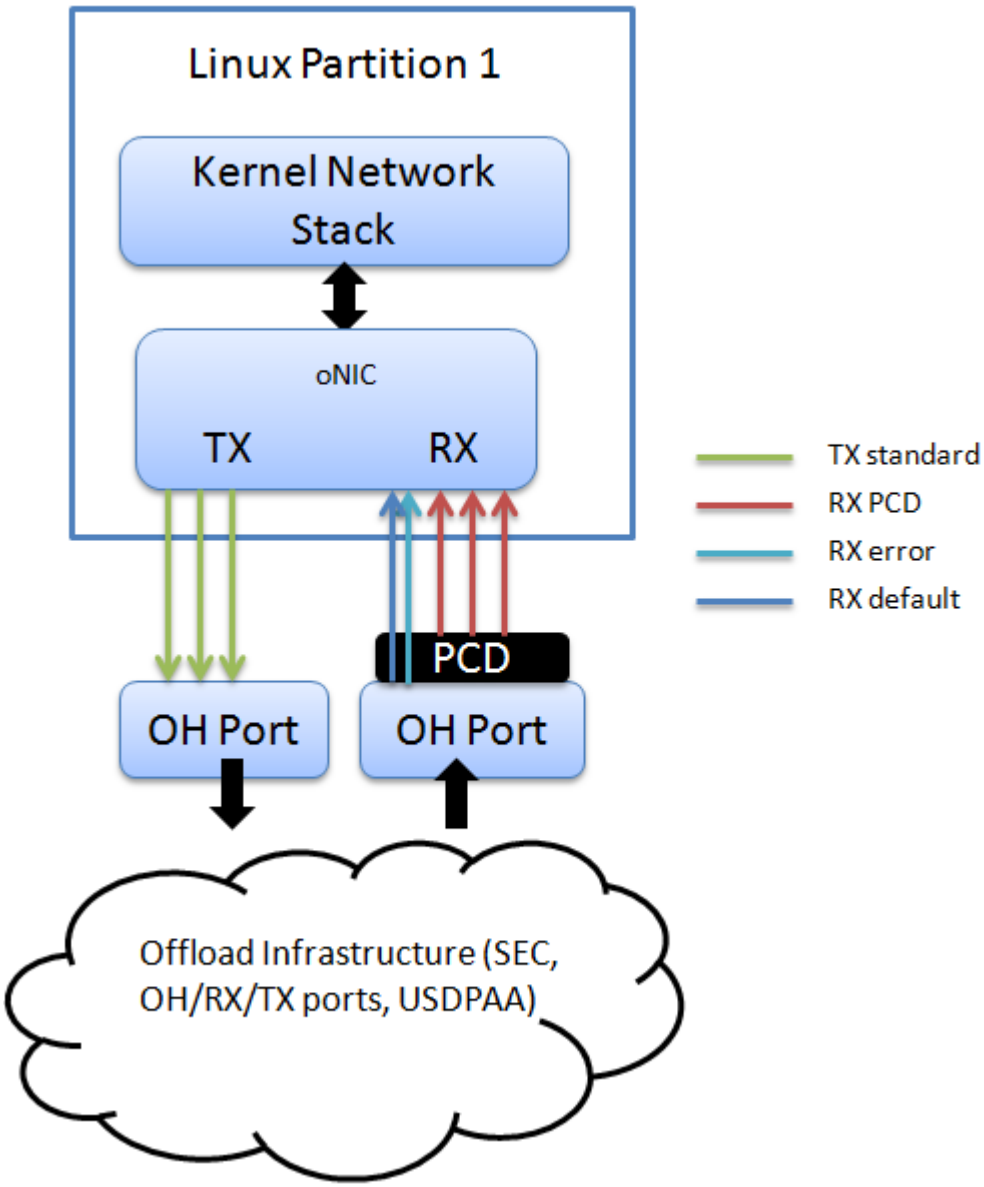
File stride size set to 17 * record size.

random random bkwd record stride

KB reclen write rewrite read reread read write read rewrite read fwrite frewrite fread freread

524288 64 7040 7253 371022 372079

1048576 64 6537 6566 9857 10203
```

Known Bugs, Limitations, or Technical Issues

1. Call trace when running `iozone` to test SD card performace on some platforms

   Workaround: increase the timeout value (in kernel configuration) and decrease the dirty_ratio in proc file system.

   a. menuconfig:

      Kernel hacking

      (xxx) Default timeout for hung task detection (in seconds)

      NOTE
      The xxx may be 400 seconds or greater

   b. Modify 'proce file system':

      ```
      echo xx > /proc/sys/vm/dirty_ratio
      echo xx > /proc/sys/vm/dirty_background_ratio
      ```

      NOTE
      The xx may be 10 or 5, which meas 10% or 5%, the default is 20%.

2. The platform whose card is required to work on a special transfer mode which is not FS or HS mode needs a special rcw. (e.g. rcw_66_15_1800MHz_emmc_ddr.rcw is for t2080qds eMMC DDR mode. Because of pin multiplexing with SPI, SPI would not work when eMMC card works on DDR mode)
3. According to SD specifications, only a power cycle could reset the SD card working on UHS-I speed mode. However, our boards couldn't provide a power cycle for SD card even with reset. This initializes SD UHS-I card to use high speed mode after board reset when the card is on UHS-I speed mode. The workaround is using power off/on instead of reset when using SD UHS-I card.

7.6 Ethernet

7.6.1 Linux Ethernet Driver for DPAA 1.x Family

7.6.1.1 Linux DPAA 1.x Ethernet Primer

Understanding the high-level concepts of the Linux driver

7.6.1.1.1 Introduction

An overview of the DPAA-Ethernet network driver, in the more generic context of Linux device drivers.

The primary concepts of the DPAA-Ethernet driver architecture are presented without going into such intricate details as device-tree configuration or code structure. The current document is neither a Linux Device Drivers tutorial, nor a replacement to the **Linux Ethernet Driver** document in the SDK, but a quick start guide which provides context for users.

The DPAA-Ethernet driver software shipped with the standard QorIQ Linux SDK is described.

7.6.1.1.2 Intended Use Cases

This chapter presents a high-level view of the standard use-cases (based on the QorIQ Linux SDK requirements) that the DPAA-Ethernet driver currently supports.

7.6.1.1.2.1 Private Net Devices

This is the primary driver and the one which currently delivers the best performance. Characteristics of this driver are:

- The private driver is a multiqueue driver - it uses 1 TX queue per CPU
- All private interfaces use a single BPID - usually dynamically allocated
- The FQIDs for the common types of queues - RX, TX, RX Error, TX Error, TX Confirm - are dynamically allocated
- The Hashing/PCD frame queues are hardcoded in the device tree. The private driver imports the PCD configuration from device tree at startup
- The above resources are allocated and visible only to the private driver

In the case of private interfaces, all network traffic takes place between the Linux kernel and the physical FMan port private to that partition.



Figure 146. Network Traffic Between the Linux Kernel and the Physical FMan Port

7.6.1.1.2.2 Shared-MAC Net Devices

A shared-MAC device is one that can be used from two (or potentially more) Linux and/or USDPAA partitions. A shared-MAC encompasses one of the following partitioning scenarios:



Figure 147. Two (or more) Linux separate partitions, under control of the TOPAZ hypervisor.



Figure 148. One Linux and one USDPAA running in the same partition

Points to emphasize

- In all cases, there is exactly one physical MAC. Linux always “owns” the MAC, which it initializes.
- A shared-MAC between two Linux interfaces (say, eth0 and eth1) from the same partition is **not supported** unless in a hypervised scenario (and has never been intended as a use-case, anyway).

Configuration and ownership of a shared-MAC is asymmetric and driven by a number of hardware and software constraints.

The following constraints and design assumptions apply to Buffer Pools in a shared-MAC device:

- FMan v2 (not supporting virtual storage profiles) picks the buffer to store the Rx frame based solely on the ingress frame size, regardless of the result of PCD.
- In shared-MAC devices used by Linux and USDPAA, to avoid the need for the USDPAA fastpath to remap the ingress buffers, the same Buffer Pool is shared between Linux and USDPAA.
- In shared-MAC devices used by two Linux partitions (no USDPAA involved), it is necessary that the Linux guests have the same true-physical to guest-physical mappings, in order for FMan-DMA to work seamlessly regardless of the destination partition. Moreover, that presumes that the buffer space seen by the two partitions is identical, which comes down to the Buffer Pools being physically shared between the two Linux partitions.

- Sharing a Buffer Pool between two or several Linux and/or USDPAA partitions requires that the BPID be statically defined (identically hard-coded) in the `.dts` configurations of all partitions.
- The convention between Linux and USDPAA is that USDPAA initializes and seeds the shared Buffer Pool. Linux dynamically remaps ingress buffers received on a shared-MAC, copies them into a buffer dynamically allocated in its own memory space, then releases the ingress buffer back into the shared Buffer Pool.
- In the case of a shared-MAC between two Linux partitions (in a hypervised scenario), only one partition will initialize and seed the shared Buffer Pool. That partition is determined by means of a special property in the `hv.dts` partition configuration. Refer to **Linux Ethernet Driver - Buffer Pools** for details.

The following constraints and design assumptions apply to Frame Queues in a shared-MAC device as seen by the Linux DPAA-Ethernet (i.e. not USDPAA) driver:

- Rx traffic is driven to either partition via a PCD configuration applied on the shared physical port. This means that, for Linux partitions using a shared-MAC, at least one Rx Frame Queue has to be statically declared in the `.dts`. That may be the Rx Default Frame Queue and/or the (automatically initialized) 128 core-affine **Hashing PCD Frame Queues**.
- In the Linux-Linux shared-MAC scenario, all Tx Frame Queues must be statically specified and be the same in both partitions. The reasoning is explained in the **Linux Ethernet Driver - Virtual/Shared Controller** document. .
- In the Linux-USDPAA shared-MAC scenarios, the Linux partition will always initialize its own Tx Frame Queues, be they dynamically or statically allocated. It is up to the USDPAA application to choose its own Tx FQIDs.
- The Tx Confirm Frame Queues are always used in shared-MAC scenarios; all egress buffers are confirmed. See **Shared MAC: Tx** packet lifecycle for details.

7.6.1.1.2.3 MAC-less Net Devices

A MAC-less device, also called a Virtual Controller in the **Linux Ethernet Driver - Virtual/Shared Controller**, appears to Linux as a regular net device (Ethernet interface). It is “virtual” in the sense that it does not have a physical FMan port (be it an online or an OH port) underlying, but that is transparent to the Linux kernel and userspace applications – only the DPAA-Ethernet driver is aware of the difference.

The figure below shows the typical MAC-less use-case is as a communication device between two Linux partitions:



Typical MAC-less setup between two Linux partitions
(Both under control of the hypervisor)

Figure 149. Communication device between two Linux partitions

… or between Linux and USDPAA:



MAC-less is supported between Linux and USDPAA running in the same partition.

Figure 150. Communication device between one Linux and one USDPAA in the same partition

From the DPAA resource configuration standpoint, a MAC-less net device is very similar to a **Shared MAC: Tx**. The same constraints apply to Buffer Pools and Frame Queues as in the case of a shared-MAC, with one notable difference:

- Because the Tx FQs of a MAC-less device always sink into another partition (or USDPAA) instead of a physical FMan port, there is the convention that the DPAA-Ethernet driver of a MAC-less node **only initializes its Rx FQs**. In other words, each partition initializes its own Rx FQs, because it has to bind local dequeue callbacks to them.

In a MAC-less setup, one endpoint's Tx FQIDs are the other endpoint's Rx FQIDs and vice versa.

While the previous diagrams have shown typical MAC-less use-cases, one can design more complex scenarios by interposing various processing blocks between the two MAC-less endpoints. Refer to the **Extended Use Cases** chapter for a discussion.

7.6.1.1.2.4 Choosing the Current Use Case

The following activity diagram describes the configuration selection logic in the DPAA-Ethernet driver, based on properties found in the `.dts` specification:



Figure 151. Configuration Selection Logic

7.6.1.1.3 The DPAA-Eth View of the World

This section presents the primary concepts behind the DPAA-Ethernet driver design.

As a Linux driver, one of DPAA-Ethernet driver's main goals is proper integration with the Linux kernel ecosystem. As a hardware device driver, the DPAA-Ethernet driver integrates functions of several DPAA IP blocks, within the scope of the defined/supported use cases.

7.6.1.1.3.1 The Linux Kernel API's

The DPAA-Ethernet drivers interface with the Linux kernel via the latter’s networking stack APIs. This is a strong requirement, mandated by the integration with the Linux kernel.

Another type of interaction with the kernel code is at boot-time, via the Open-Firmware API. That API is used to parse the PowerPC platform device tree and discover the hardware modules that need to be configured. In particular, the DPAA-Ethernet drivers use the platform device tree to discover:

- What net devices to probe and what type of hardware is underlying those devices;
- Which DPAA resources are involved: FQIDs, BPIDs, CGRIDs, FMan port IDs.



Figure 152. Platform Device Tree

Generally, we prefer drivers configurations to be dynamic and transparent to the rest of the system. Among the benefits of dynamic resource allocations, we count:

- Portability of the drivers across multiple QorIQ platforms;
- Seamless support of platform changes (e.g. via booting with different RCWs);
- Seamless support of multiple partitions under the control of a hypervisor;
- Cohabitation with other DPAA drivers (e.g. a SEC driver) in the SDK.

In certain scenarios, however, configurations are statically defined and are extracted directly from the .dts specification. This is for instance the case of shared MAC and MAC-less devices.



Figure 153. Shared MAC and MAC-less Devices

7.6.1.1.3.2 The Driver's Building Blocks

This chapter presents the main structures and data entities with which the DPAA-Ethernet driver operates.

The driver's building blocks are part of the interfaces of the driver with the relating components, i.e.:

- The kernel’s IP stack;
- The DPAA hardware blocks and their drivers.

The DPAA Ethernet driver is actually a series of drivers, each tailored for a specific use case:

- Private driver - maps Linux kernel network interfaces to physical ports
- Shared driver - gives simultaneous control of a physical port to the Linux kernel and user space applications
- Macless driver - although it appears to the Linux kernel as a regular net device, it does not control a physical port. It is typically used as a virtual communication device between two Linux partitions
- Proxy driver - an interface through which the Linux kernel configures a physical port, only to pass its control to user space applications
- Offline port driver - controls Offline Parsing / Host Command port (OH)

Each of the above drivers has its own code base and implements the Linux kernel API by providing its own callback functions.

7.6.1.1.3.2.1 Net Devices

A net device (`struct net_device` in C representation) is the fundamental structure of any Linux network device driver.

A net device describes a (physical or virtual) device capable of sending and receiving packets over a (virtual or physical) network. All incoming and outgoing traffic is accounted and processed on behalf of the net device it comes or goes on.

Each supported type of net device has its own kernel driver. If there are several such devices present in a system, there will be as many device driver instances.

A net device is accessible to the Linux user via the standard tools, such as ‘ifconfig’ or ‘ethtool’.

Not all net devices have real underlying hardware; tunnel endpoints, for examples, are represented by net devices but are not directly backed by hardware. Same holds for drivers such as “bonding” or “dummy”.

It is worth emphasizing, however, that **every** Linux interface is represented by a net device. This is a fundamental design aspect of all Linux networking drivers, including DPAA-Ethernet. One can describe the Linux IP stack as being a **netdev-centric** construction. Nearly all of the kernel networking APIs receive a `struct net_device` as a parameter. The `net_device` structure is the handle through which the driver and the network stack communicate.

The following diagram illustrates what has just been described:



Figure 154. Every Linux Interface is Represented by a Net Device

7.6.1.1.3.2.2 Frame Queues

The Frame Queue is one of the fundamental concepts of DPAA. In the case of DPAA-Ethernet, it is the main interface between the network driver and the hardware blocks.

Ingress frames received by the DPAA-Ethernet driver on one of the Frame Queues it is servicing are sent to the IP stack on behalf of the net device structure that the driver is associated with. Conversely, outgoing frames coming from the IP stack into the driver are enqueued to one of the egress Frame Queues.

NOTE

Depending on its configuration (see usecase), the DPAA-Ethernet driver may initialize some of its Frame Queues and make assumptions over what entity initializes the others.

7.6.1.1.3.2.3 Buffer Pools

Buffer pool configuration is another fundamental part of the DPAA-Ethernet driver design.

Unlike the Frame Queue utilization – which is more flexible – the Buffer Pool utilization is conditioned by several design assumptions:

- The source and ownership of the ingress frame buffers are presumed by the DPAA-Ethernet driver. The exact allocation logic depends on the driver configuration (see usecase), but in all cases the driver makes hard assumptions on how the buffers were allocated.

  For instance, the “private MAC” driver configuration seeds the Buffer Pools at predefined checkpoints on the Rx path. There are also buffer utilization counters maintained by the driver, which influence the buffer allocation logic.

  The “shared MAC” and “MAC-less” driver configurations only work with the Buffer Pools hard-coded in the platform `.dts`.

- The layout of incoming frames is also presumed by the driver. Depending on its configuration (see usecase), the driver expects the frame layout to conform to its own allocation logic. The actual buffer layout is outside the scope of this document and should not be assumed upon by driver users.
- The existence of a static Buffer Pool configuration in the platform `.dts` determines the driver configuration. The DPAA-Ethernet driver currently assumes that, if a static Buffer Pool is configured in its device tree node (as number/size/address of buffers), then it is describing a “shared MAC” or a “MAC-less” configuration.

7.6.1.1.4 DPAA Resources Initialization

The rationale behind the “what”s, “why”s and “how”s of DPAA resource initializations made by the DPAA-Ethernet driver are presented. This description does not go into the full detail of driver configuration (please refer to the **Linux Ethernet Driver** for that level of detail).

7.6.1.1.4.1 What, Why and How Resources are Initialized

DPAA resources initialized by the various configurations of the DPAA-Ethernet driver are:

- FQs and FQIDs (where static config applies);
- BPs and BPIDs (where static config applies);
- Buffers (not quite “DPAA” resources, rather “system” resources);;
- CGRs (CGRIDs are always dynamic);
- FMan’s online ports (note: the offline ports are configured by a different driver than DPAA-Ethernet).

Frame Queues and Buffer Pools have been covered at length in the previous chapters of this document. CGRs are of lesser interest from the initialization viewpoint.

FMan’s online ports are initially probed by the FMan Driver (FMD) and later in the boot process they are configured by the DPAA-Ethernet driver instances according to the specifications in the `.dts`.

7.6.1.1.4.2 Hashing/PCD Frame Queues

Among the Frame Queues initialized by the DPAA-Ethernet driver, there is a predefined set of 128 core-affined Rx FQs, automatically initialized by the driver for user’s convenience. They are there because most performance-enhanced setups must use a PCD configuration; to that end, the standard QorIQ Linux SDK provides a “hashing PCDs” configuration that can be applied by the user via the FMC tool. Since FMC does not support dynamic FQID specification in its .xml configuration files, the “hashing PCD” Frame Queues also have static, hard-coded FQIDs.

Furthermore, apart from the core-affined Rx FQs, there is another set of 128 core-affined Rx FQs, which have a higher priority than the former. They are named throughout this documentation "Rx PCD High Priority Frame Queues". Likewise, the queues in this set are also core-affined and have static, hard-coded FQIDs.

For details about the “hashing PCD” Frame Queues and the Rx PCD High Priority Frame Queues, refer to the **Linux Ethernet Driver - Core Affined Queues**.

7.6.1.1.5 The (Simplified) Life of a Packet

This chapter presents a packet’s lifecycle in various configurations of the DPAA-Ethernet driver.

7.6.1.1.5.1 Private Net Device: Tx



Figure 155. DPAA-Ethernet driver enqueues the packet to the FMan port

Arrows in the diagram above represent the direction of the buffer/packet flow.

A packet on the egress path is allocated by the network stack using the kernel's standard memory allocator. The DPAA-Ethernet driver enqueues the packet to the FMan port with an indication to recycle the buffer if possible. If recycling is not possible, the DPAA-Ethernet driver itself frees the buffer memory back to the kernel's allocator, when Tx delivery is confirmed by FMan.

7.6.1.1.5.2 Private Net Device: Rx



Figure 156. Buffers on the Ingress Path

Buffers on the ingress path are acquired by FMan directly from a Buffer Pool which was seeded by the DPAA-Ethernet driver. Buffer layout is important to the driver, which assumes ownership on the BP. Arrows in the diagram above represent the direction of the buffer/packet flow.

7.6.1.1.5.3 Shared MAC: Tx

The following diagram presents the lifecycle of an egress buffer/packet in the case of a shared-MAC device used by two Linux partitions (under the control of the Hypervisor, not shown in the picture):



Figure 157. A Shared-MAC Device Used by Two Linux Partitions

There are essentially two buffer circuits in this scenario:

- The in-stack socket buffer lifecycle (colored in blue);
- The bpool buffer lifecycle (colored in red).

Socket buffers are dynamically allocated from the kernel memory. The DPAA-Ethernet driver memcopies them in buffers acquired from the Buffer Pool, then releases the socket buffers back into the kernel memory.

The shared Buffer Pool is seeded by one (and only one) of the Linux partitions, and is never again replenished by the software. Memory used for seeding the shared Buffer Pool is statically defined in the guest `.dts` configuration and is presented to the kernel as device memory (not as physical memory). Both DPAA-Ethernet drivers statically ioremap the entire shared Buffer Pool space (as per the static configuration in the .dts) at probe time. No run-time mapping/unmapping is effected afterwards.

After transmission, FMan confirms the frame and the Linux DPAA-Ethernet driver releases the buffers back into the Shared Buffer Pool.

Similarly to the above, the next diagram the lifecycle of an egress buffer/packet in the case of a shared-MAC device used by Linux and USDPAA:



Figure 158. Shared-MAC Device Used by Linux and USDPAA

7.6.1.1.5.3.1 Shared MAC: Rx

The diagram below presents the lifecycle of a buffer/packet in the case of a shared-MAC device used by Linux and USDPAA running in the same partition.



Figure 159. Shared-MAC Device Used by Linux and USDPAA

The Linux driver is the one to initialize the shared FMan port, but it is the USDPAA partition that allocates and seeds the buffers in the Shared Buffer Pool. To that end, USDPAA uses a block of physical memory reserved at boot time, such that the kernel can still map it, but not allocate from it (via either the page allocator or the object cache allocator).

Each incoming frame is dynamically mapped by the DPAA-Ethernet driver. As in the case of shared-MAC Tx, a memcopy is involved in the driver, from each incoming frame into a newly allocated socket buffer (sk_buff). The DPAA-Ethernet driver subsequently unmaps the buffer and releases it back into the shared Buffer Pool, for later reuse.

One should note the invariant that, as in all MAC-less and shared-MAC scenarios, the Shared Buffer Pool is initialized and seeded exactly once, at init-time; at run-time, the total number of in-flight buffers and buffers available in the pool is constant. One must preallocate a large enough number of buffers in the Shared Buffer Pool, such that to prevent its run-time depletion.

NOTE

A similar diagram can be obtained in the case of a shared-MAC between two different Linux partitions, with the principal change that the shared memory from which the Buffer Pool is seeded is seen as iomem (rather than physical memory), and is statically ioremapped all at once, at boot-time.

7.6.1.1.5.3.2 MAC-less Net Devices: Tx

The following diagram presents the lifecycle of a buffer/packet in the case of a MAC-less device used for communication from Linux to USDPAA:



Figure 160. MAC-less Device Used for Communication from Linux to USDPAA

7.6.1.1.5.3.3 MAC-less Net Devices: Rx

The following diagram presents the lifecycle of a buffer/packet in the case of a MAC-less device used for communication from USDPAA to Linux:



Figure 161. a MAC-less Device Used for Communication from USDPAA to Linux

7.6.1.1.6 Advanced Drivers Use Cases

This chapter is meant as an initial guideline toward building more complex use-cases, based on the predefined built-in capabilities of the DPAA-Ethernet driver that have been explained so far. (Note: Such use-cases are not currently part of the standard Linux QorIQ SDK.)

7.6.1.1.6.1 MAC-less Over OH (Linux-USDPAA

One can interpose an OH port between a Linux MAC-less net device and a USDPAA MAC-less interface, where the Linux-to-USDPAA path goes through the OH port, while the USDPAA-to-Linux path is direct:



Figure 162. Linux MAC-less net device and a USDPAA MAC-less interface

In the current implementation, the DPAA Ethernet Driver initializes Linux MAC-less Rx FQs (= USDPAA's MAC-less Tx FQs).

Because there is no symmetric MAC-less endpoint to initialize the Linux endpoint's Tx FQs (as in the case of a standard MAC-less Linux-Linux setup), a separate kernel module is necessary in order to:

- Initialize the Linux MAC-less Tx FQs; and
- Add them to the DCP of the Offline Port.

The diagram shows:

Table 186. DPAA Ethernet Driver initializes FQs

#	Description	Detail
1	A MAC-less Linux interface with:	• 8 Tx FQs – initialized by the Offline Port Initialization Module and placed in the OH port's DCP; • 8 Rx FQs – initialized by the MAC-less interface and connected to the Tx FQs of the USDPAA MAC-less interface.
2	A MAC-less USDPAA interface with:	• 8 Tx FQs – initialized by the Linux MAC-less interface. (Note: USDPAA currently asserts that there are exactly 8 Tx FQs. This can be changed if a different number of frame queues are requested). • 8 Rx FQs - initialized by the MAC-less interface. Unlike in the standard Linux-Linux MAC-less scenario, these are different from the Tx FQs of the Linux MAC-less interface, because of the interposing OH port. The Offline Port's Rx Default and Rx Error FQs (and possibly the PCD FQs) use some of these FQIDs.
3	An Offline Port Initialization Module.	Responsible with initializing the 8 Tx FQs of the Linux MAC-less interface and connecting them to the DCP of the OH port.

The common idea of this design is that the entity which uses the queues for Rx is responsible with their initialization and (for software portals) specifying their dequeue callbacks. On the kernel side, this is the responsibility of the MAC-less interface (DPAA-Ethernet driver) and on USDPAA the responsibility is of the application that uses the FQs.

7.6.1.1.6.2 MAC-less Over OH (Linux-Linux)

Similarly to the previous use-case, one can use an Offline Port's interface with the QMan for interposing an OH between two MAC-less endpoints and effectively build a MAC-less-over-OH net device.

7.6.1.1.6.3 ARP Handling in Shared MAC

1. Linux-Linux: Currently (FMan-v2), ARP packets arriving on a shared-MAC interface are (in absence of relevant PCD configuration) routed to the Linux partition.
2. Linux-USDPAA: This is entirely handled by USDPAA infrastructure. Please refer to the USDPAA documentation.

7.6.1.1.6.4 Multicast Support in Shared MAC

This is not a supported feature in the FMan-v2- and FMan-v3-based DPAA-Ethernet drivers.

7.6.1.1.7 Appendix A: Infrequently Asked Questions

Table 187. Q and A

#	Question	Answer
1	How do I send a frame up the network stack?	The frame-processing network stack only exists in the context of a net device. So, “sending a frame into the stack” is an inaccurate statement: the frame must first be associated to a net device, and then the respective instance of the Ethernet driver will deliver the frame to the stack, on behalf of that net device. To achieve that, the frame must arrive via the physical device that underlies the driver (or via a MAC-less device’s ingress FQs).
2	Can I allocate a buffer and inject it as a frame into a private interface’s ingress queues?	This is probably a mistake. The DPAA-Ethernet driver makes hard assumptions on buffer ownership, allocation and layout. In addition, the driver expects FMan Parse Results to be placed in the frame preamble, at an offset which is implementation-dependent. In short, while a carefully crafted code might work, it would make for *very* brittle design, and hard to maintain, too. **NOTE** A MAC-less interface (potentially modified to interpose an OH port), however, will support this case, as long as: a) the buffer originates from the interface’s shared Buffer Pool; and b) the frame is a valid Ethernet frame.
3	But can I acquire a buffer directly from a private interface’s Buffer Pool, and inject it as such into the private interface’s Rx FQs?	While this works on a MAC-less or shared-MAC interface, it is not an intended use-case for private interfaces.
4	What format must an ingress frame have, from the standpoint of the DPAA-Ethernet driver and the Linux kernel stack?	The DPAA-Ethernet driver is expected to perform an initial validation of the ingress frame, but does not look at the Layer-2 fields directly. The current kernel networking code does make a check on the MAC addresses of the frame and the protocol (Ethertype) field. One should not make assumptions on such details of frame processing, because the kernel stack implementation is not bound by any contract.

7.6.1.1.8 Appendix B: Frequently Asked Questions

The right place to look at the driver FAQs is the **Linux Ethernet Driver** document in the SDK.

7.6.1.2 Linux DPAA 1.x Ethernet Drivers

7.6.1.2.1 Introduction

This document describes the Linux drivers which enable support for Ethernet on processors with the Datapath Acceleration Architecture (DPAA). The focus is on the theory and operation behind using Ethernet. It provides only limited discussion of the BMan, QMan, and FMan, describing instead the layer of software which allows all of these to interoperate. For the purposes of this document, all these drivers will be referred to as the DPAA Ethernet Drivers. Enablement, configuration and debugging for all DPAA Ethernet Drivers are described in this document.

Purpose

The DPAA Ethernet Driver is meant to manage the use of the Datapath hardware for communication via the Ethernet protocol. This includes facilities for:

- Allocating buffer pools and buffers
- Allocating frame queues
- Assigning frame queues and buffer pools to specified FMan ports
- Transferring packets between frame queues and the Linux stack
- Controlling Link Management features

Overview

Ethernet in the Datapath is realized by interconnecting BMan, QMan, and FMan. The primary interaction for the DPAA Ethernet Driver is between the kernel and the QMan. Ethernet frames are delivered to the driver from the frame queue via the QMan portal, and the driver delivers Ethernet frames to the outgoing frame queue via the QMan portal. For some use cases, that is the only interaction.

Usually, the frame queues are connected to a FMan port. Each FMan port has two queues which must be assigned to them: a default queue and an error queue. This assignment can be specified in the device tree, or created dynamically by the driver on initialization.

The Ethernet frames are often stored in buffers which are associated with a BMan buffer pool. The driver sets up this pool, and either seeds it with buffers, or maps the buffers which are put into the pool. Depending on the pool, the buffers may be allocated and freed by the kernel during network activity, or they may be allocated once, and return to the pool when not in use by the Datapath hardware.

DPAA Ethernet Driver types

The complexity of DPAA allows a variety of possible use cases. Although speed is the key factor for performance in most use cases, customization and usability are preferred in others. Building a single Ethernet driver to cover all these tasks was, in the ever-changing world of the Linux kernel, a task that was becoming harder day-by-day. Instead we divided the single basic block into several specialized and simpler Ethernet drivers which can be combined in complex configurations:

- The Private DPAA Ethernet Driver resembles the common Linux Ethernet driver. It is highly improved for performance and uses all the features that DPAA offers;
- The Macless DPAA Ethernet Driver and the Shared DPAA Ethernet Driver flavors are used in virtualized multi-partition scenarios (with the Topaz Hypervizor) or in custom user-space traffic analyzer use cases;
- The Proxy DPAA Ethernet Driver will do the entire preliminary work in scenarios where all the control is passed to user-space, bypassing the standard Linux kernel standard.
- The Offload NIC Ethernet Driver is similar to the Macless Driver and can be used in the same scenarios. In addition, it has offloading capabilities that the Macless Driver lacks, and uses Offline Parsing/Host Command Ports.
- The MACsec Driver is used to configure the FMan MACsec hardware block that is capable of offloading the IEEE 802.1AE's protocol features.

7.6.1.2.2 Private DPAA Ethernet Driver

The Private DPAA Ethernet Driver manages the network interfaces which are fully owned by the Linux partition who runs them. Therefore, it is possible to take advantage of the DPAA facilities in order to increase the performance in both termination and forwarding scenarios.

The Private DPAA Ethernet Driver will be further referenced as the Private Driver.



7.6.1.2.2.1 Configuration

This chapter present the configuration options for the Private DPAA Ethernet Driver.

7.6.1.2.2.1.1 Device Tree Configuration

The compatible string used to define a private interface in device tree is „fsl,dpa-ethernet". The default structure for the device tree node that specifies a Private interface should be similar to the below snippet of a B4860QDS device tree node:

```
ethernet@4 {
        compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet";
        fsl,fman-mac = <&fm1mac5>;
};
```

"fsl,fman-mac" is the reference to the MAC device connected to this interface. This property is used to determine which RX and TX ports are connected to this interface.

Buffer pools

A single buffer pool is currently defined and used by all the Private interfaces. The buffer pool ID is dynamically allocated and provided by the Buffer Manager. The number and size of the buffers in the pool is decided internally by the Private driver therefore no device tree configuration is accepted.

Frame queues

The frame queues are allocated by the Private driver with IDs dynamically allocated and provided by the Queue Manager. The frame queues can also be statically defined using two additional device tree properties:

```
ethernet@0 {
        compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet";
        fsl,fman-mac = <&fm1mac5>;
        fsl,qman-frame-queues-rx = <0x100 1 0x101 1 0x180 128>;
        fsl,qman-frame-queues-tx = <0x200 1 0x201 1 0x300 8>;
};
```

Within the example above, a value of 0x100 was assigned to the RX error frame queue ID and 0x101 to the RX default frame queue ID. In addition, 128 PCD frame queues raging between 0x180-0x1ff are defined and assigned to the core-affined portals in a round-robin fashion.

There is exactly one RX error and one RX default queue hence a value of "1" for the frame count. Optionally, one can specify a value of "0" for the base to instruct the driver to dynamically allocate the frame queue IDs.

Within the example above, a value of 0x200 was assigned to the TX error queue ID and 0x201 to the TX confirmation queue ID. The third entry specifies the queues used for transmission.

If the qman-frame-queues-rx and qman-frame-queues-tx are not present in the device tree, the number of dynamically allocated TX queues is equal to the number of cores available in the partition.

7.6.1.2.2.1.2 Bootargs

Two bootarg parameters are defined for the Frame Manager driver but they also influence the behavior of the Private driver:

- fsl_fm_max_frm
- fsl_fm_rx_extra_headroom

fsl_fm_max_frm

The Frame Manager discards both Rx and Tx frames that are larger than a specific Layer2 MAXFRM value. The DPAA Ethernet driver won't allow one to set an interface's MTU too high such that it would produce Ethernet frames larger than MAXFRM. The maximum value one can use as the MTU for any interface is (MAXFRM - 22) bytes, where 22 is the size of an Eth+VLAN header (18 bytes), plus the Layer2 FCS (4 bytes).

Currently, the value of MAXFRM is set at boot-time and cannot be changed without rebooting the system.

The default MAXFRM is 1522, allowing for MTUs up to 1500. If a larger MTU is desired, one would have to reboot and reconfigure the system as described next. The maximum MAXFRM is 9600

The MAXFRM can be set in two ways:

- as a Kconfig option (CONFIG_FSL_FM_MAX_FRAME_SIZE):

```
Device Drivers
+-> Network device support
  +-> Ethernet driver support
    +-> Freescale devices
       +-> Frame Manager support
          +-> Freescale Frame Manager (datapath) support
            +-> Maximum L2 frame size
```

- as a bootarg:

- In no-HV scenarios: In the u-boot environment, add "fsl_fm_max_frm=<your_MAXFRM>" directly to the "bootargs" variable.
- In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying "fsl_fm_max_frm=<your_MAXFRM>;".

Note that any value set directly in the kernel bootargs will override the Kconfig default. If not explicitly set in the bootargs, the Kconfig value will be used.

Symptoms of Misconfigured MAXFRM

MAXFRM directly influences the partitioning of FMan's internal MURAM among the available Ethernet ports, because it determines the value of an FMan internal parameter called "FIFO Size". Depending on the value of MAXFRM and the number of ports being probed, some of these may not be probed because there is not enough MURAM for all of them. In such cases, one will see an error message in the boot console.

fsl_fm_rx_extra_headroom

Configure this to tell the Frame Manager to reserve some extra space at the beginning of a data buffer on the receive path, before Internal Context fields are copied. This is in addition to the private data area already reserved for driver internal use. The option does not affect in any way the layout of transmitted buffers. The default value (64bytes) offers best performance for the case when forwarded frames are being encapsulated (e.g. IPSec).

The RX extra headroom can be set in two ways:

- as a Kconfig option (CONFIG_FSL_FM_RX_EXTRA_HEADROOM):

```
Device Drivers
+-> Network device support
  +-> Ethernet driver support
    +-> Freescale devices
      +-> Frame Manager support
        +-> Freescale Frame Manager (datapath) support
          +-> Add extra headroom at beginning of data buffers
```

- as a bootarg:
  - In no-HV scenarios: In the u-boot environment, add "fsl_fm_rx_extra_headroom=< your_rx_extra_headroom>" directly to the "bootargs" variable.
  - In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying " fsl_fm_rx_extra_headroom=<your_rx_extra_headroom>;".

7.6.1.2.2.1.3 Kconfig Options

The private driver has a number of parameters which can be tuned at compile time from menuconfig. These can be found in:

```
Device Drivers
    +- Network Device Support
        +- Ethernet Driver Support
            +- Freescale Devices
                +- DPAA Ethernet
```

FSL_DPAA_ETH_JUMBO_FRAME - "Optimize for jumbo frames"

Optimizes the DPAA Ethernet driver throughput for large frames termination traffic (e.g. 4K and above).

On FMan v2 platforms this option requires modifications to the device tree, otherwise the ping will not work. Include 'qoriq-fman-0-chosen-fifo-resize.dtsi' in 'qoriq-fman-0.dtsi' and 'qoriq-fman-1-chosen-fifo-resize.dtsi' in 'qoriq-fman-1.dtsi'

```
/include/ "qoriq-fman-0-chosen-fifo-resize.dtsi"
```

Using this option in combination with small frames increases significantly the driver's memory footprint and may even deplete the system memory. Also, the skb truesize is altered and messages from the stack that warn against this are bypassed.

This option is not available on LS1043A platforms.

FSL_DPAA_1588 - "IEEE 1588-compliant timestamping"

Enables IEEE1588 support code.

FSL_DPAA_TS - "Linux compliant timestamping"

Enables Linux API compliant timestamping support.

FSL_DPAA_ETH_USE_NDO_SELECT_QUEUE - "Use driver's Tx queue selection mechanism"

The DPAA-Ethernet driver defines a ndo_select_queue() callback for optimal selection of the egress FQ. That will override the XPS support for this netdevice. If for whatever reason you want to be in control of the egress FQ-to-CPU selection and mapping, or simply don't want to use the driver's ndo_select_queue() callback, then unselect this and use the standard XPS support instead.

FSL_DPAA_ETH_MAX_BUF_COUNT - "Maximum nuber of buffers in private bpool"

Defaults to "128". The maximum number of buffers to be by default allocated in the DPAA-Ethernet private port's buffer pool. One needn't normally modify this, as it has probably been tuned for performance already. This cannot be lower than DPAA_ETH_REFILL_THRESHOLD.

FSL_DPAA_ETH_REFILL_THRESHOLD - "Private bpool refill threshold"

Defaults to "128". The maximum number of buffers to be by default allocated in the DPAA-Ethernet private port's buffer pool. One needn't normally modify this, as it has probably been tuned for performance already. This cannot be lower than DPAA_ETH_REFILL_THRESHOLD.

FSL_DPAA_CS_THRESHOLD_1G - "Egress congestion threshold on 1G ports"

The size in bytes of the egress Congestion State notification threshold on 1G ports. Ranges from 0x1000 to 0x10000000. Defaults to 0x06000000. This option can help when:

- the device stays congested for a prolonged time (risking the netdev watchdog to fire - see also the tx_timeout module param)
- preventing the Tx cores from tightly-looping (as if the congestion threshold was too low to be effective)

This might also implies some risks:

- affecting performance of protocols such as TCP, which otherwise behave well under the congestion notification mechanism
- running out of memory if the CS threshold is set too high

FSL_DPAA_CS_THRESHOLD_10G - "Egress congestion threshold on 10G ports"

The size in bytes of the egress Congestion State notification threshold on 10G ports. Ranges from 0x1000 to 0x20000000. Defaults to 0x10000000.

FSL_DPAA_INGRESS_CS_THRESHOLD - "Ingress congestion threshold on FMan ports"

The size in bytes of the ingress tail-drop threshold on FMan ports. Defaults to 0x10000000. Traffic piling up above this value will be rejected by QMan and discarded by FMan.

FSL_DPAA_ETH_DEBUGFS - "DPAA Ethernet debugfs interface"

This option compiles debugfs code for the DPAA Ethernet driver.

FSL_DPAA_ETH_DEBUG - "DPAA Ethernet Debug Support"

This option compiles debug code for the DPAA Ethernet driver.

7.6.1.2.2.1.4 ethtool Options

The private driver implements the following ethtool operations

```
-a --show-pause
    Queries the specified Ethernet device for pause parameter information.
-A --pause
    Changes the pause parameters of the specified private devices.
    rx on|off
        Specifies whether RX pause should be enabled.
    tx on|off
        Specifies whether TX pause should be enabled.
-k --show-features
    Lists the offloadable DPAA driver features. Specifies which features can be changed.
-K --features
    Changes a driver feature.
    feature on|off
        Specifies weather a certain feature should be enabled.
-s --change
    msglvl N
    msglvl type on|off ...
        Sets the driver message type flags by name or number. type names the type of message to
enable or disable; N  specifies  the new flags numerically.
-S --statistics
    Shows driver statistics and counters: interrupt counter, packet counters, error counters,
congestion state, and more.
--show-eee
    Shows the Energy-Efficient Ethernet configurations.
--set-eee
    Configures the EEE behavior.
```

7.6.1.2.2.2 Features

This chapter present the Private DPAA Ethernet Driver features.

7.6.1.2.2.2.1 Congestion Management

QMan offers 3 methods of managing congestion:

- WRED
- congestion state tail drop (CSTD)
- FQ tail drop (FQTD)

The Private driver implements CSTD both on TX and on RX. When the number of bytes residing in a TX FQ congestion group reaches a congestion threshold (high watermark), the QMan rejects any further incoming frames, until the sum of all the frames contained in the congestion groups drops under a low watermark, which is 7/8 of the high watermark. The high watermark can be configured from menuconfig. See section "Kconfig options" for more details.

7.6.1.2.2.2.1.1 Bootargs

Two bootarg parameters are defined for the Frame Manager driver but they also influence the behavior of the Private driver:

- fsl_fm_max_frm

- fsl_fm_rx_extra_headroom

fsl_fm_max_frm

The Frame Manager discards both Rx and Tx frames that are larger than a specific Layer2 MAXFRM value. The DPAA Ethernet driver won't allow one to set an interface's MTU too high such that it would produce Ethernet frames larger than MAXFRM. The maximum value one can use as the MTU for any interface is (MAXFRM - 22) bytes, where 22 is the size of an Eth+VLAN header (18 bytes), plus the Layer2 FCS (4 bytes).

Currently, the value of MAXFRM is set at boot-time and cannot be changed without rebooting the system.

The default MAXFRM is 1522, allowing for MTUs up to 1500. If a larger MTU is desired, one would have to reboot and reconfigure the system as described next. The maximum MAXFRM is 9600

The MAXFRM can be set in two ways:

- as a Kconfig option (CONFIG_FSL_FM_MAX_FRAME_SIZE):

```
Device Drivers
+-> Network device support
  +-> Ethernet driver support
    +-> Freescale devices
       +-> Frame Manager support
          +-> Freescale Frame Manager (datapath) support
            +-> Maximum L2 frame size
```

- as a bootarg:
  - In no-HV scenarios: In the u-boot environment, add "fsl_fm_max_frm=<your_MAXFRM>" directly to the "bootargs" variable.
  - In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying "fsl_fm_max_frm=<your_MAXFRM>;".

Note that any value set directly in the kernel bootargs will override the Kconfig default. If not explicitly set in the bootargs, the Kconfig value will be used.

Symptoms of Misconfigured MAXFRM

MAXFRM directly influences the partitioning of FMan's internal MURAM among the available Ethernet ports, because it determines the value of an FMan internal parameter called "FIFO Size". Depending on the value of MAXFRM and the number of ports being probed, some of these may not be probed because there is not enough MURAM for all of them. In such cases, one will see an error message in the boot console.

fsl_fm_rx_extra_headroom

Configure this to tell the Frame Manager to reserve some extra space at the beginning of a data buffer on the receive path, before Internal Context fields are copied. This is in addition to the private data area already reserved for driver internal use. The option does not affect in any way the layout of transmitted buffers. The default value (64bytes) offers best performance for the case when forwarded frames are being encapsulated (e.g. IPSec).

The RX extra headroom can be set in two ways:

- as a Kconfig option (CONFIG_FSL_FM_RX_EXTRA_HEADROOM):

```
Device Drivers
+-> Network device support
  +-> Ethernet driver support
    +-> Freescale devices
      +-> Frame Manager support
        +-> Freescale Frame Manager (datapath) support
          +-> Add extra headroom at beginning of data buffers
```

- as a bootarg:

- In no-HV scenarios: In the u-boot environment, add "fsl_fm_rx_extra_headroom=< your_rx_extra_headroom>" directly to the "bootargs" variable.
- In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying " fsl_fm_rx_extra_headroom=<your_rx_extra_headroom>;".

7.6.1.2.2.2.2 Scatter/Gather Support

On the Rx path, the first S/G entry is used to build the skb linear part and the other entries are used as fragments.

The Private driver can access the egress skbufs allocated in high memory (e.g. mapped directly from user-space, as is the case of the sendfile() system call). This eliminates the kernel need to copy such skbufs into newly-allocated low memory buffers, allowing zero-copy on the egress path.

7.6.1.2.2.2.3 Jumbo Frames Support

Termination traffic with large frames performs better if only linear skbs (and single buffer frames) are used. The driver has the option to allocate Rx buffers large enough to accommodate the entire frame (of max 9.6K).

This option needs to be used with caution, as the memory footprint can be a real problem when small frames are used.

The option can be enabled from the menuconfig option:

```
Device Drivers
    +-> Network Device Support
        +-> Ethernet Driver Support
            +-> Freescale Devices
                +-> DPAA Ethernet
                    +-> Optimize for jumbo frames
```

In addition to enabling this feature from menuconfig, the user is required to set the L2 maximum frame size to 9600, otherwise the configuration is not valid. This can be achieved by either setting fsl_fm_max_frm=9600 in the bootargs, or configuring CONFIG_FSL_FM_MAX_FRAME_SIZE from menuconfig. For more details see Bootargs on page 1016.

7.6.1.2.2.2.4 GRO/GSO Support

Generic Receive Offload (GRO) is tied to NAPI support and works by keeping a list of GRO flows per each NAPI instance. These flows can then "merge" incoming packets, until some termination condition is met or the current NAPI cycle ends, at which point the flows are flushed up the protocol stack. Flows merging several packets share the protocol headers and coalesce the payload (without memcopying it). This results in a CPU load decrease and/or network throughput increase. Packets which don't match any of the stored flows (in the current NAPI cycle) are sent up the stack via the normal, non-GRO path.

GRO is commonly supported in hardware as a set of "GRO assists", rather than full packet coalescing. The following features count as GRO assists:

- RX hardware checksum validation
- Receive Traffic Distribution (RTD)
- Multiple RX/TX queues
- Receive Traffic Hashing
- Header prefetching
- Header separation
- Core affinity
- Interrupt affinity

Note: With the exception of header separation, the DPAA platforms feature all other hardware assists. Most notably, they are implicitly achieved through the mechanisms that accompany PCDs.

Generic Segmentation Offload (GSO) is also a well-established feature in the Linux kernel. Normally, a TCP segment is composed in the Layer 4 of the Linux stack, based on the current MSS (Maximum Segment Size) connection setting. It has been observed, though, that delaying segmentation is a better approach in terms of CPU load, because fewer headers are processed. Linux has taken an optimization approach, called GSO, whereby the L4 segments are only composed just before they are handed over to the L2 driver.

GRO and GSO support are available by default in the Private driver and can be independently switched on and off at runtime, via *ethtool -k*.

Note: Older versions of ethtool don't support this. Ethtool version 3.0 does - and possibly others before it, too.

Generic optimizations that enhance the driver's performance in the general case also apply to the GRO/GSO-enabled driver. PCD support is therefore recommended in this regard. We have found that these optimizations yield the best results on 10Gbps traffic, and to a lesser extent (if any) on 1Gbps traffic. TCP tests, especially, can benefit from GRO by shedding CPU load and upping the network throughput. The improvements are the more visible with smaller network MTU - with MTU=1500 and below, the benefits are higher, while starting from MTU=4k they are no longer observable.

One optimization that boosts GSO performance is the zero-copy egress path. That is available thanks to the *sendfile()* system call, which may be used instead of the plain *send()* syscall, and which certain benchmark applications know about. Netperf for instance has *sendfile* support in its *TCP_SENDFILE* tests.

GRO and GSO are no panacea, one-button-fix-all kind of optimization. While under most circumstances they should be transparent (this being why GRO is by default enabled in the Linux kernel), there are scenarios and configurations where they may in fact under-perform. Traffic on 1Gbps ports sees little benefit from GRO/GSO. Also, if the Private Driver detects that PCDs are not in place, GRO is automatically by-passed.

7.6.1.2.2.2.5 Transmit Packet Steering

The Private driver exposes to the Linux networking stack a TX-multiqueue interface. This provides the stack with better control of the transmission queues and reduces the need for locking. The user may also control the mapping of egress FQs to the CPUs via a standard Linux feature called Transmit Packet Steering (XPS) and documented here: http://lwn.net/Articles/412062/

NOTE

The kernel transmission queues are different entities than the Private driver Frame Queues.

The Private driver, however, matches the two realms by mapping the DPAA FQs onto kernel's own queue structures. To that end, the Private driver provides a standard callback (net-device operation, or NDO) called ndo_select_queue(), which the stack can interrogate to find out the specific queue mapping it needs for transmitting a frame. The existence of that NDO (which is otherwise optional) overrides the kernel queue selection via XPS. This is why the Private driver provides a compile-time choice to disable the ndo_select_queue() callback, leaving it to the stack to choose a transmission queue.

To use the Private driver's builtin ndo_select_queue() callback, select the Kconfig option **FSL_DPAA_ETH_USE_NDO_SELECT_QUEUE**.

To disable the Private driver's queue selection mechanism and use XPS instead, unselect this Kconfig option. Further on, the users can configure their own txq-to-cpu mapping, as described in the LWN article above.

7.6.1.2.2.2.6 TX and RX Hardware Checksum

Introduction

The FMan block supports calculation of the L3 and/or L4 checksum for certain standard protocols.

This can be used, on the TX path, for calculating the checksum of the outgoing frame, and on the RX path, for validating the L3/L4 checksum of the incoming frame and making classification, or distribution decisions.

TX Checksum Support

On TX, the checksum computation is enabled on a per-frame basis by the Private driver. The TX checksum support for standard protocols is as follows:

Table 188. TX checksum support

Header	IPv4	IPv6	Other
IP header	yes	not available	no
TCP header	yes	yes	no
UDP header	yes	yes	no

NOTE

IP Header checksum capability also exists in SEC block (see IPSEC).

NOTE

Ethernet CRC is calculated on a per frame basis during frame transmission.

NOTE

The main precondition for TX checksum to be enabled in hardware is that IP tunneling must not be present (i.e., not GRE, not MinEnc, not IPIP). Other conditions pertain to the validity and integrity of the frame.

RX Checksum Support

This feature is disabled by default. In order to enable RX checksum computation for supported protocols, a PCD scheme must be applied to the respective RX port. In the current release, L3 and L4 are both enabled if a PCD is applied.

If enabled, L3 and L4 checksum validation is performed for TCP, UDP and IPv4.

NOTE

Controlling this feature via ethtool is not yet supported.

7.6.1.2.2.7 Pause Frames Flow Control

FMan supports IEEE 802.3x flow control. Whenever the FMan RX FIFO threshold is exceeded, FMan transmits PAUSE frames to the other peer on the link. In Linux, the transmission and reception of PAUSE frames can be enabled or disabled using ethool.

To display PAUSE frames settings in use for an interface

```
ethtool -a intf_name
```

Triggering PAUSE frames ON/OFF

PAUSE frames can be enabled/disabled on RX/TX using ethtool -A, like in the following examples

```
ethtool -A intf_name rx on
ethtool -A intf_name tx off
ethtool -A intf_name rx off tx off
```

Autonegotiation

Starting with SDK 1.6, the DPAA Private driver supports PAUSE frame autonegotiation.

When autonegotiation is enabled and the user enables/disables PAUSE frames on RX/TX, these will not automatically be triggered on/off. Instead, the local and the peer PAUSE symmetric/asymmetric capabilities will be considered. If the peer does not match the local capabilities, the following commands may have no effect:

```
ethtool -A intf_name rx on
ethtool -A intf_name rx off
ethtool -A intf_name tx on
ethtool -A intf_name tx ff
```

When autonegotiation is disabled, ethtool settings override the result of link negotiation.

PAUSE frame autonegotiation can also be enabled/disabled using ethtool -A

```
ethtool -A intf_name autoneg on
ethtool -A intf_name autoneg off
```

FMAN v3 platforms

On the following platforms: T4, B4, and T1040, 802.1Qbb Priority Flow Control is used instead of 802.3x PAUSE frames. The ethtool controls are the same, but the structure of the frames is different.

Find about DPAA Private support of PFC in the following section:

Priority Flow Control on page 1024

7.6.1.2.2.8 Priority Flow Control

Beginning with SDK 1.6, the DPAA Ethernet Driver offers experimental support for IEEE standards 802.1Qbb (Priority Flow Control) and 802.1p.

These standards aim to implement lossless Ethernet, in which the highest-priority classes of traffic benefit from maximum bandwidth and minimum delay. Up to 8 classes of service can be used, but only a minimum of 3 is required.

The terms “Class of Service (CoS)” and “priority” will be used interchangeably in this section.

802.1Qbb PFC frames are available only on platforms with FMan v3, namely T4, B4 and T1040. For the other platforms 802.3x PAUSE frames are used instead for Ethernet flow control.

Enabling PFC Support

To enable PFC support, enable the following options from menuconfig

```
Device Drivers
+ Network device support
  + Ethernet driver support
    + Freescale devices
      + Frame Manager support
        + Freescale Frame Manager (datapath) support
          + FMan PFC support (EXPERIMENTAL)
            + (3)     Number of PFC Classes of Service
            + (65535) The pause quanta for PFC CoS 0
            + (65535) The pause quanta for PFC CoS 1
```

```
            + (65535) The pause quanta for PFC CoS 2
```

The number of Classes of Service can range between 1 and 4. It defines the number of Work Queues used and the number of priorities that are set when a PFC frame is issued. 3 is the default value. Changing this value also changes the number of WQs and priorities.

The pause time can be adjusted for each CoS individually.

Enabling and disabling CoS and their pause time is unavailable at runtime. It is only possible at compile time in this release.

Selecting the Class of Service

When PFC support is enabled, the egress traffic flowing on a DPAA Private interface is distributed on the first 3 Work Queues of a TX port, namely WQ0, WQ1 and WQ2.

These function in strict priority. WQ0 has the highest priority and WQ2 the lowest priority. FMan cannot dequeue frames from WQ1 unless WQ0 is empty and from WQ2 unless WQ1 and WQ0 are empty.

The work queue a frame will be enqueued on is determined from the socket buffer priority. skb_prio is just an internal tag that the kernel applies to the frames on the egress path and is not visible to the receiver.

skb_prio	CoS
0	0
1	1
≥2	2

The default skb_prio is 0, which means all frames will be distributed to WQ0. skb_prio can be modified using a number of methods, including traffic control.

To edit a socket buffer's priority using tc, one needs to enable the following options from menuconfig.

```
Networking support
+ Networking options
  + QoS and/or fair queueing
    + Multi Band Priority Queueing (PRIO)
    + Elementary classification (BASIC)
    + Universal 32bit comparisons w/ hashing (U32)
    + Extended Matches
      + U32 key
    + Actions
      + SKB Editing
```

The following commands assign a skb_prio of 1 to traffic destined to TCP and UDP port 5000 and implicitly direct it on WQ1.

```
tc qdisc del dev fm1-mac9.0 root
tc qdisc add dev fm1-mac9.0 root handle 1: prio
tc filter add dev fm1-mac9.0 parent 1: protocol ip u32 match ip dport 5000 action skbedit
priority 1
```

VLAN tagging

In order to be classified by the receiver according to 802.1p the egress traffic must be VLAN tagged, with the Class of Service contained in the PCP field. The PCP priority is also determined from skb_prio.

```
# create a subinterface of fm1-mac9, with VLAN ID 0
vconfig add fm1-mac9 0
# all frames tagged with skb_prio 1, will have PCP priority of 1.
vconfig set_egress_map fm1-mac9.0 1 1
```

If no mapping is specified the PCP field will be set to 0 by default.

The dependence between skb_prio, work queues and VLAN PCP priority:



Receiving PFC Frames

Unlike ordinary 802.3x PAUSE frames, PFC frames can selectively pause a certain priority/CoS.

WQ0 responds to PFC frames that have priority 0 set. Example: When a PFC frame arrives containing priority 0 and having a 100 pause time for priority 0, WQ0 i.e. all traffic from CoS 0 is ignored for dequeing for 100 bit times, and dequeing is done from WQ1 and WQ2.

Generating PFC frames

All DPAA Private interfaces share a single buffer pool which accounts for the buffers in which the frames are stored upon receiving.

When the Buffer Pool reaches the refill/depletion threshold, PFC frames are sent back to the sender in order to pause frames transmission and thus avoid frame loss.

FMan sends PFC frames that pause all Classes of Traffic defined. The only difference between the classes is the pause time.

The pause time can be configured from menuconfig. A pause time of 0 disables that Class of Service.

When the common buffer pool depletes, issued PFC frames look like this.



FMan issues either 802.1Qbb PFC or 802.3x PAUSE frames depending on the platform, but there is no difference in controlling their transmission and reception via ethtool. For more detatils, see the chapter on PAUSE frames support.

Enabling and disabling PFC using ethtool

FMan issues either 802.1Qbb PFC or 802.3x PAUSE frames depending on the platform, but there is no difference in controlling their transmission and reception via ethtool. For more detatils, see the chapter on PAUSE frames support.

Pause Frames Flow Control on page 1023

7.6.1.2.2.9 Core Affined Queues

The driver automatically creates 128 core-affined queues, intended to be used as RX PCD frame queues. These frame queues can be used in PCD configuration files to process certain types of frames on particular CPUs. In order to enhance the PCD files creation, the /etc/fmc/config/ directory from rootfs contains the default configuration and policy files for each platform.

The driver calculates the frame queue IDs based on the address of the MAC registers corresponding to the port using the following formula:

((MAC register address) & 0x1fffff) >> 6

Following are the values for various QorIQ DPAA platforms:

Table 189. FMAN v2 devices core affined queues

Interface	FQID base	P4080	P5020	P5040	P3041	P2041	T1040 (FMAN v3)
fm1-gb0	0x3800	Y	Y	Y	Y	Y	Y
fm1-gb1	0x3880	Y	Y	Y	Y	Y	Y
fm1-gb2	0x3900	Y	Y	Y	Y	Y	Y
fm1-gb3	0x3980	Y	Y	Y	Y	Y	Y
fm1-gb4	0x3a00		Y	Y	Y		Y
fm1-10g	0x3c00	Y	Y	Y			

Table continues on the next page...

Table 189. FMAN v2 devices core affined queues (continued)

Interface	FQID base	P4080	P5020	P5040	P3041	P2041	T1040 (FMAN v3)
fm2-gb0	0x7800	Y		Y			
fm2-gb1	0x7880	Y		Y			
fm2-gb2	0x7900	Y		Y			
fm2-gb3	0x7980	Y		Y			
fm2-gb4	0x7a00			Y			
fm2-10g	0x7c00	Y		Y			

Table 190. FMAN v3 devices core affined queues

Interface	FQID base	T4240	T4160	B4860	B4420	T2080
fm1-mac1	0x3800	Y	Y	Y	Y	Y
fm1-mac2	0x3880	Y	Y	Y	Y	Y
fm1-mac3	0x3900	Y	Y	Y	Y	Y
fm1-mac4	0x3980	Y	Y	Y	Y	Y
fm1-mac5	0x3a00	Y	Y	Y		
fm1-mac6	0x3a80	Y	Y	Y		
fm1-mac9	0x3c00	Y	Y	Y		Y
fm1-mac10	0x3c80	Y		Y		Y
fm2-mac1	0x7800	Y	Y			
fm2-mac2	0x7880	Y	Y			
fm2-mac3	0x7900	Y	Y			
fm2-mac4	0x7980	Y	Y			
fm2-mac5	0x7a00	Y	Y			
fm2-mac6	0x7a80	Y	Y			
fm2-mac9	0x7c00	Y	Y			
fm2-mac10	0x7c80	Y				

These queues are assigned to cores in a round-robin fashion. For instance, if there are 8 cores, 0x3800 will be serviced by core 0, 0x3801 by core 1, 0x3808 by core 0, etc. Currently, if one specifies extra RX PCD queues in the device tree, these queues will **also** be assigned in this round-robin fashion.

High Priority Core Affined Queues

Starting with SDK 2.0, a new set of RX PCD frame queues has been added, to aid in implementing complex traffic management scenarios. This set of frame queues has a higher priority than the normal RX PCD frame queues, and as such, traffic coming in on these frame queues has a higher precedence than the traffic coming on on the default RX PCD frame queues. One scenario where this is useful is the back-to-back IPsec testing scenario, where the encrypted traffic (RX) is desirable to have a higher priority than the plain text traffic.

The driver calculates the high priority frame queue IDs based on the address of the MAC registers corresponding to the port using the following formula:

65536 + ((MAC register address) & 0x1fffff) >> 6

Following are the values for various QorIQ DPAA platforms:

Table 191. FMAN v2 devices core affined queues

Interface	FQID base	P4080	P5020	P5040	P3041	P2041	T1040 (FMAN v3)
fm1-gb0	0x13800	Y	Y	Y	Y	Y	Y
fm1-gb1	0x13880	Y	Y	Y	Y	Y	Y
fm1-gb2	0x13900	Y	Y	Y	Y	Y	Y
fm1-gb3	0x13980	Y	Y	Y	Y	Y	Y
fm1-gb4	0x13a00		Y	Y	Y		Y
fm1-10g	0x13c00	Y	Y	Y			
fm2-gb0	0x17800	Y		Y			
fm2-gb1	0x17880	Y		Y			
fm2-gb2	0x17900	Y		Y			
fm2-gb3	0x17980	Y		Y			
fm2-gb4	0x17a00			Y			
fm2-10g	0x17c00	Y		Y			

Table 192. FMAN v3 devices core affined queues

Interface	FQID base	T4240	T4160	B4860	B4420	T2080
fm1-mac1	0x13800	Y	Y	Y	Y	Y

Table continues on the next page...

Table 192. FMAN v3 devices core affined queues (continued)

Interface	FQID base	T4240	T4160	B4860	B4420	T2080
fm1-mac2	0x13880	Y	Y	Y	Y	Y
fm1-mac3	0x13900	Y	Y	Y	Y	Y
fm1-mac4	0x13980	Y	Y	Y	Y	Y
fm1-mac5	0x13a00	Y	Y	Y		
fm1-mac6	0x13a80	Y	Y	Y		
fm1-mac9	0x13c00	Y	Y	Y		Y
fm1-mac10	0x13c80	Y		Y		Y
fm2-mac1	0x17800	Y	Y			
fm2-mac2	0x17880	Y	Y			
fm2-mac3	0x17900	Y	Y			
fm2-mac4	0x17980	Y	Y			
fm2-mac5	0x17a00	Y	Y			
fm2-mac6	0x17a80	Y	Y			
fm2-mac9	0x17c00	Y	Y			
fm2-mac10	0x17c80	Y				

7.6.1.2.3 Ethernet Advanced Drivers

This current chapter describes advanced DPAA Ethernet driver configurations that can used instead of the default Private DPAA Ethernet driver.

NOTE

The DPAA Ethernet Advanced drivers are supported on the PPC platforms only.

7.6.1.2.3.1 Macless DPAA Ethernet Driver

Macless DPAA Ethernet Driver is a virtual Ethernet driver, hence not using an Ethernet controller to transmit frames. This type of DPAA Ethernet driver is a lightweight version of Private DPAA Ethernet Driver that does not have MAC device control primitives, but maintains the same structure. Macless DPAA Ethernet Driver is used with many different names, “macless”, “MAC-less” or previous SDK name, “Virtual Ethernet Driver”. This DPAA Ethernet Driver is used in simple scenarios, like communication between USDPAA and Linux or communication between two Linux partitions or more complex ones, like Offloading Architecture and Shared MAC scenarios. All these scenarios are presented in the Configuration chapter.

7.6.1.2.3.1.1 Configuration

The main configuration options are offered, like in any other embedded Linux Ethernet driver, by the device tree configuration and the common interface offered by the Linux kernel (ifconfig, ethtool, etc.). All configuration capabilities are presented in this chapter.

7.6.1.2.3.1.1.1 Device Tree Configuration

The Macless DPAA Ethernet Driver has *fsl,dpa-ethernet-macless* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies a Macless interface in a B4860QDS device tree looks like:

```
ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-macless", "fsl,dpa-ethernet-macless";
    fsl,bman-buffer-pools = <&bp10>;
    fsl,qman-frame-queues-rx = <0xfa0 0x8>;
    fsl,qman-frame-queues-tx = <0xfa8 0x8>;
    local-mac-address = [00 11 22 33 44 55];
};
```

The properties that a Macless Driver device tree node can have are:

- *fsl,bman-buffer-pools* - a list of buffer pools used by this interface. The Macless DPAA Ethernet Driver will use only static defined buffer pools because the frames are transmitted between different memory spaces. See Note 2 for more details;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues that will transmit frames to the Macless Driver. These queues are initialized as PCD queues. By default there are 8 Rx queues because there are 8 CPUs on B4860QDS boards;
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues that fetch the frames from Macless Driver to the next hardware device (TX Port, O/H Port) or to another Macless Driver. These queues will only be created, but not initialized. It is the role of the next software module in the data path to initialize these queues;
- *local-mac-address* - this is a virtual L2 address used to identify the driver in the Linux kernel network stack.

Note 1: The Macless DPAA Ethernet Driver does not have a MAC device to control. Because of this the “fsl,fman-mac” property is missing from the device tree specification. This property exists for all other DPAA Ethernet Drivers.

Note 2: A static defined buffer pool should be declared as exemplified below for a B4860QDS board:

```
bp7: buffer-pool@7 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <7>;
    fsl,bpool-ethernet-cfg = < 0 256 0 192 0 0x40000000>;
    fsl,bpool-thresholds = <0x400 0xc00 0x0 0x0>;
};
```

Although the above device tree node is a representation for the BMan Driver, *fsl,bpool-ethernet-cfg* property is parsed solely by the DPAA Ethernet Driver that has a reference to this buffer pool. This property has the following meaning:

```
fsl,bpool-ethernet-cfg = <count size base_address>;
```

- *count* - represents the number of buffers from the buffer pool;
- *size* - buffer size;
- *base_address* - physical address of the buffer pool. Because two Linux partition have different memory spaces, this physical address will be mapped in both partitions. In scenarios where a single partition is used, this address will be invalid, particularly 0, and a dynamic mapping from user-space to kernel-space will be done.

The example above declares a buffer pool with buffer pool ID 7, and describes a pool with 256 192-byte buffers, occupying the memory region from 0x40000000 to 0x40000000 + 256*192. The reason there are two numbers per each of *count*, *size*, and *base_address* is that we support 36-bit addresses on the P4080 (and 64-bit on P5020). It should be noted that the size of those parameters should be set by the root node's *#address-cells* and *#size-cells* properties. The *fsl,bpool-ethernet-seeds* property is there to tell the driver which is using the buffer pool whether to seed the pool with the declared buffers, or not.

The above generic device tree structure can be modified, depending on the scenario that Macless Ethernet Driver is used into. These scenarios are:

Communication inside a single Linux partition between USDPAA and Linux Network Stack through a Macless DPAA Ethernet Driver.

For some applications, USDPAA needs the benefits and flexibility of Linux networking capabilities. To connect to the Linux Network Stack, it uses a Macless DPAA Ethernet Driver. This scenario is represented in the following picture:



The device tree configuration should be similar to the one below extracted from a B4860QDS device tree configuration:

```
ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-macless", "fsl,dpa-ethernet-macless";
    fsl,bman-buffer-pools = <&bp16>;
    fsl,qman-frame-queues-rx = <0xfa0 0x8>;
    fsl,qman-frame-queues-tx = <0xfa8 0x8>;
    local-mac-address = [00 11 22 33 44 55];
};

bp16: buffer-pool@16 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <0x10>;
    fsl,bpool-ethernet-cfg = <0x0 0x800 0x0 0x6c0 0x0 0x0>;
    fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
};
```

In the above device tree snippet, there are two device tree nodes, one for the Macless DPAA Ethernet Driver and one for the static definition of the buffer pool used in the transmission. USDPAA will also parse these nodes and it will manage the buffer pool used in the communication. Additionally, because a Macless DPAA Ethernet Driver initializes only its RX frame queues, USDPAA has the additional role of initializing Macless DPAA Ethernet Driver's TX frame queues also. Another important configuration is the base address of the buffer pool. Because the communication occurs inside a single Linux partition, there is no need for the buffer pool to advertise its physical address, hence the 0 address representing an invalid physical address. The mapping of the Macless kernel memory space to USDPAA's user-space memory space is done through the kernel *kmap*/*kunmap* primitives.

Communication between two Linux partitions through a Macless DPAA Ethernet Driver

With the help of the NXP virtualization solution, called Topaz, two or more partitions can run on the same board. Every communication between partitions is supervised by Topaz. Although network communication between partitions is possible through external connections, one direct connection can be created inside the board using Macless DPAA Ethernet Driver

and hardware frame queues. To create a fast networking communication between partitions, two or more Macless DPAA Ethernet Drivers should be used as depicted below:



The TX frame queues from one partition should be the RX frame queues from the other partition. The device tree nodes in each partition are represented below.

First partition's device tree representation:

```
dpa-ethernet@10 {
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet-macless";
    fsl,qman-frame-queues-rx = <0x4000 8>;
    fsl,qman-frame-queues-tx = <0x4008 8>;
    local-mac-address = [02 00 c0 a8 6f fe];
    fsl,bman-buffer-pools = <&bp10>;
};
bp10: buffer-pool@10 {
    compatible = "fsl,b4080-bpool", "fsl,bpool";
    fsl,bpid = <10>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80000000>;
    fsl,bpool-ethernet-seeds;
};
```

Device tree from the second partition:

```
dpa-ethernet@10 {
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet-macless";
    fsl,qman-frame-queues-rx = <0x4008 8>;
    fsl,qman-frame-queues-tx = <0x4000 8>;
    local-mac-address = [02 00 c0 a8 79 fe];
    fsl,bman-buffer-pools = <&bp10>;
};
bp10: buffer-pool@10 {
    compatible = "fsl,b4080-bpool", "fsl,bpool";
    fsl,bpid = <10>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80000000>;
};
```

In the above configuration there are 8 TX frame queues and 8 RX frame queues used in interchangeable roles in each partition. Both interfaces share one statically defined buffer pool. The same buffer pool definition should be used in both partitions, except for the *fsl,bpool-ethernet-seeds* property. In order to easily map from one memory address space from one partition to another partition a valid physical base address must be configured. The mapping from physical address to each partition's virtual kernel address is done with *ioremap* kernel primitive. The seeding of the buffer pool will be done by the Macless DPAA Ethernet Driver from the partition that has *fsl,bpool-ethernet-seeds* in its device tree node. In the above configuration, the first partition will seed the buffer pool.

Communication inside a Linux partition using Offline Ports and Macless DPAA Ethernet Driver

DPAA has many hardware offload capabilities. One of them is the Offline Parsing/Host Command Port. This hardware portal can fetch traffic from multiple destinations (such as Macless DPAA Ethernet Drivers) and send traffic through many configurable destinations, like MAC devices. For example, this scenario can be used to offload into DPAA stack functionality, like IPSec, IP fragmentation and reassembly or TCP segmentation and reassembly.



Because the Macless DPAA Ethernet Driver does not have an Ethernet controller attached, it can be used in many hardware configuration scenarios similar to the one used in the figure above. In order to do that, special attention should be paid to Frame Queues configuration and the Buffer Pool configuration.

Communication of different Linux partitions through a single MAC device, using Macless DPAA Ethernet Driver and Shared DPAA Ethernet Driver

In this scenario, a new type of DPAA Ethernet Driver is needed. This is called Shared DPAA Ethernet Driver and its configuration is described in Shared DPAA Ethernet Driver configuration chapter. Shared DPAA Ethernet Driver together with Macless DPAA Ethernet Driver are used in a typical shared MAC scenario, described in the Shared DPAA Ethernet Driver chapter.

7.6.1.2.3.1.1.2 Configuration available through Linux interfaces

Because Macless driver does not manage a MAC device, some *ethtool* and *ifconfig* options will not be available. All Layer 3 configurations, like setting an IP address, are generally available.

7.6.1.2.3.1.2 Features

This chapter describes the features of the Macless DPAA Ethernet Driver, their configuration options, fine-tuning and known limitations.

MAC device control capability

As of SDK 1.5, the Macless DPAA Ethernet Driver is capable of controlling a MAC device on behalf of an user-space application, like USDPAA. This will add an additional control mechanism to the amount of customization available through

Macless DPAA Ethernet Driver. For example, in the scenario depicted in *Communication inside a Linux partition using Offline Ports and Macless DPAA Ethernet Driver* scenario, it is desired that one of the Macless DPAA Ethernet Drivers should control the MAC device.

Some of the MAC devices are initialized by the Proxy DPAA Ethernet Driver to be used by USDPAA. By setting the following attribute in the device tree specification of the Macless DPAA Ethernet Driver

```
proxy = <&proxy1>;
```

a reference to the MAC device initialized by the Proxy driver can be obtained in the Macless driver. This will allow basic operations regarding L2 address of the MAC device using Macless driver Linux interface. Currently, the supported operations are enablement and disablement of the MAC device and setting multicast or unicast addresses.

Scatter/Gather

The current implementation supports DPAA Scatter/Gather only on the RX path. Therefore, the Macless DPAA Ethernet Driver will know how to handle Scatter/Gather frames, but the driver does not advertise TX Scatter/Gather capability to the Linux Network Core and only linear socket buffers will be accepted on the TX path.

Hardware checksum

As previously presented in the Private DPAA Ethernet Driver's Features chapter, the FMan supports computation of the L3 and/or L4 checksum for certain protocols (mainly TCP/IP and UDP/IP). Because the same transmission procedure is used to activate hardware checksum in both Private and Macless drivers, the same limitations presented in the *Private DPAA Ethernet Driver* chapter apply to Macless driver. Currently, at frame reception, the Macless Driver is not able to fetch the checksum computed by FMan.

7.6.1.2.3.2 Shared DPAA Ethernet Driver

Shared DPAA Ethernet Driver is similar to Macless Driver, only it has a MAC device in its control. Although it has similar structures with Private DPAA Ethernet Driver, it does not have the same optimization and feature set available. This is because it can be used in pair with a Macless Driver from another Linux partition or it can be used in pair with USDPAA. Therefore, it needs portability and easiness, qualities that Private DPAA Ethernet Driver sacrifices in exchange for higher performance. Shared DPAA Ethernet Driver is also known as *shared* or *Shared Controller* in the previous SDK release. In this document Shared DPAA Ethernet Driver will be referred as *Shared Driver*. This flavor of DPAA Ethernet Driver is used in two scenarios. First, when a MAC device is shared between different partitions under a Hypervisor and second, when a MAC device is shared with a User Space DPAA application in a single Linux partition. These use cases are presented in the Configuration chapter below.

7.6.1.2.3.2.1 Configuration

The main configuration options are offered by the device tree configuration and common Linux interfaces (*ifconfig*, *ethtool*, etc.). All configuration capabilities are presented in this chapter.

7.6.1.2.3.2.1.1 Device Tree Configuration

The Shared Driver has the *fsl,dpa-ethernet-shared* string as compatible string in the device tree. Therefore, the standard structure for the device tree node that specifies a Shared interface should be similar to the below snippet of a B4860QDS device tree node:

```
ethernet@9 {
    compatible = "fsl,b4860-dpa-ethernet-shared", "fsl,dpa-ethernet-shared";
    fsl,fman-mac = <&fm1mac10>;
    fsl,bman-buffer-pools = <&bp17>;
    fsl,qman-frame-queues-rx = <0x5e 1 0x5f 1 0x2000 3>;
    fsl,qman-frame-queues-tx = <0 1 0 1 0x3000 8>;
};
```

Following are the properties of a Shared Driver's device tree node:

- *fsl,fman-mac* - this is the MAC device reference that is in Shared Driver's control;

- *fsl,bman-buffer-pools* - as in Macless Driver specification this is a list of buffer pools used by this interface. Since the frames are transmitted between different memory spaces, Shared Driver will use only static defined buffer pools in both use cases;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues from which the Shared Driver dequeues frames. Because the Shared Driver has a MAC device in its control, the device tree format of the frame queues is similar to the Private Driver's device tree format. The MAC device must have at least two types of frame queues specified at initialization. These are the error frame queue which is the frame queue on which received erroneous frames will be placed and the default frame queue which is the frame queue where the frame will be placed if no other frame queue is selected for transmission. The next batch represents the PCD queues. These queues will be used by the PCD rules configured by the user. The above device tree example defines one error queue with ID 0x5e, one default queue with ID 0x5f and 3 configurable PCD queues, 0x2000, 0x2001, 0x2002;
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues used by the Shared Driver send the frames to the MAC device. As in Private DPAA Ethernet Driver, these are the TX error frame queue, TX confirmation frame queue and the standard TX frame queues. In the above example, a value of 0 for TX error and TX confirmation queues enables the dynamic allocation of values for queues' IDs, letting the QMan assign available values. Besides one dynamic TX error and one dynamic TX confirmation queue, 8 standard TX queues will be created, with IDs between 0x3000 and 0x3007. It is recommended to specify up to NR_CPUs frame queues and have a direct mapping between frame queue and CPU. In the above example, B4860QDS has 8 CPUs.

Note 1: The static defined buffer pool has the same representation as defined in the Macless Driver Configuration chapter. As stated above, the Shared Driver will be used in two different scenarios. Each scenario involves entities that have different memory address spaces, but must share the buffer pools' configuration. This is the reason for not using dynamic buffer pools, like in Private DPAA Ethernet Driver.

Following are the scenarios in which Shared Driver can be used:

USDPAA and Linux stack communicating through a single MAC device, using Shared Driver inside a single Linux partition

USDPAA applications handle only a certain type of traffic, the rest of the packets being handled by Linux Network Stack. The simplest solution is to have a Shared Driver that knows about the traffic division. In order to split the traffic, PCD rules must be applied on the incoming port. This scenario is presented below.



The device tree configuration should be similar to the one below extracted from a B4860QDS device tree configuration.

```
ethernet@9 {
    compatible = "fsl,b4860-dpa-ethernet-shared", "fsl,dpa-ethernet-shared";
    fsl,bman-buffer-pools = <&bp17>;
    fsl,qman-frame-queues-rx = <0x5e 1 0x5f 1 0x2000 3>;
    fsl,qman-frame-queues-tx = <0 1 0 1 0x3000 8>;
    fsl,fman-mac = <&fm1mac10>;
};
bp17: buffer-pool@17 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <17>;
    fsl,bpool-ethernet-cfg = <0 2048 0 1728 0 0>;
    fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
};
```

In the above device tree snippet, there are two device tree nodes, one for the Shared Driver and one for the static definition of the buffer pool used for transmission. In this scenario, USDPAA allocates the buffer pool used in the communication, therefore the above device tree nodes will be parsed by USDPAA also. One important configuration is the base address of the buffer pool. Because the communication occurs inside a single Linux partition, there is no need for the buffer pool to advertise its physical address, hence the 0 address representing an invalid physical address. Because the buffers are allocated by USDPAA, these are in the virtual address space of USDPAA. The mapping of the USDPAA's user-space memory space to kernel space is done through kernel *kmap*/*kunmap* primitives.

Shared communication between two Linux partitions through a single MAC device using a Shared Driver and Macless Driver

In scenarios with multiple Linux partitions managed by a Hypervisor, it is sometimes desired to split one MAC device's traffic between partitions. To achieve this, one partition should use the Shared Driver which will manage the MAC device and the other Linux partition should use the Macless Driver as depicted below.



In scenarios with multiple Linux partitions managed by a Hypervisor, it is sometimes desired to split one MAC device's traffic between partitions. To achieve this, one partition should use the Shared Driver which will manage the MAC device and the other Linux partition should use the Macless Driver as depicted below.

First partition:

```
dpa-ethernet@4 {
    compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet-shared";
    fsl,qman-frame-queues-rx = <0x340 1 0x341 1 0x320 8>;
    fsl,qman-frame-queues-tx = <0x342 1 0x343 1 0x300 8>;
    fsl,bman-buffer-pools = <&part1_bp11>;
};
part1_bp11: buffer-pool@11 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <11>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80036000>;
    fsl,bpool-ethernet-seeds;
};
```

Second partition:

```
dpa-ethernet@20 {
    compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet-macless";
    fsl,qman-frame-queues-rx = <0x350 8>;
    fsl,qman-frame-queues-tx = <0x300 8>;
    local-mac-address = [02 00 c0 a8 a1 fe];
```

```
    fsl,bman-buffer-pools = <&part2_bp11>;
};
part2_bp11: buffer-pool@11 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <11>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80036000>;
};
```

In the above configuration the standard 8 TX queues from the first partition are the same as the standard 8 TX queues from the second partition. Both interfaces share one static buffer pool and the same buffer pool definition is used in both partitions. As in other multi-partition scenarios, the buffers are allocated from the physical memory, indicated by the base address parameter. In the above configuration, the base address is 0x80036000. When a frame arrives to one partition, its address should be mapped to partition's kernel address space. The mapping from physical address to each partition's virtual kernel address is done with *ioremap* kernel primitive. The seeding of the buffer pool will be done by the Shared Driver from the first partition as it has *fsl,bpool-ethernet-seeds* in its device tree node.

7.6.1.2.3.2.1.2 Configuration available through Linux interfaces

The Shared Driver has the same data structures as the Private Driver, therefore it has the same configuration capabilities available through standard Linux interfaces (*ethtool*, *ifconfig*, etc.) as the Private Driver.

7.6.1.2.3.2.2 Features

The same transmission primitives are used both by the Shared and the Macless driver. Therefore, as stated in the Macless Driver's *Features* chapter, the Shared Driver is capable of receiving Scatter/Gather frames and is able to offload checksum computation on transmission. On reception, Shared Driver is not able to fetch the checksum computed by FMan.

7.6.1.2.3.3 Proxy DPAA Ethernet Driver

USDPAA applications achieve high speeds for particular types of traffic, bypassing the processing done by the Linux Network Stack. These applications reside in user-space and need kernel-space configuration in order to initialize the necessary hardware modules used along the data path. The configuration of buffer pools and frame queues and initialization of MAC devices used by the USDPAA are delegated to the Proxy DPAA Ethernet Driver. This type of DPAA Ethernet Driver runs once at system startup and does not have a data structure at run-time like the other DPAA Ethernet Drivers. Proxy DPAA Ethernet Driver is used with many different names, *proxy* or *Initialization Manager*, as it was known in the previous SDK release. In this document, it will be referred to as *Proxy Driver*.

7.6.1.2.3.3.1 Configuration

The only configuration for this type of DPAA Ethernet Driver can be made through device tree node configuration.

7.6.1.2.3.3.1.1 Device Tree Configuration

The Proxy Driver has *fsl,dpa-ethernet-init* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies a Proxy interface in a B4860QDS device tree is depicted below:

```
ethernet@8 {
    compatible = "fsl,b4860-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,fman-mac = <&fm1mac10>;
    fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
    fsl,qman-frame-queues-rx = <0x5c 0x1 0x5d 0x1>;
    fsl,qman-frame-queues-tx = <0x7c 0x1 0x7d 0x1>;
};
```

All the above properties are the same as those used by Shared Driver that has a MAC device reference:

- *fsl,fman-mac* - this is the MAC device reference that will be initialized by the Proxy Driver;

- *fsl,bman-buffer-pools* - each port needs a buffer pool to get buffers from. Since it has a MAC device reference, Proxy Driver will need a statically defined buffer pool. This buffer cannot be dynamically created because the Proxy Driver does not know the memory address space of the software entity that will use the initialized infrastructure.
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues. The device tree format of this property is the same as the format used in Private and Shared Drivers. The MAC device must have at least two queues specified at initialization. These are:
  - error queue which is the queue on which received erroneous frames will be placed;
  - default queue which is the queue where the frame will be placed if none of the PCD queues is selected for transmission;
  - PCD queues are optional. These queues will be used by the PCD rules configured by the user. In the above device tree example no PCD queues are defined.
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues used by the software entity to send the frames to the MAC device. As in Private or Shared Drivers, these are TX error queue, TX confirmation queue and standard TX queues. In the above example, there are no standard TX queues.

Note 1: The statically defined buffer pools have the same device tree structure as that used for other DPAA Ethernet Drivers.

The classical scenario where a Proxy Driver is used is the one in which it initializes the MAC device on behalf of USDPAA inside a single Linux partition. Another scenario is where the initialization is made on behalf of another software entity or even another Linux partition.

7.6.1.2.3.3.2 Features

Proxy Driver is used only to initialize some of the hardware modules and does not have advanced features like other DPAA Ethernet Drivers.

7.6.1.2.3.4 Offload NIC Ethernet Driver

Offload NIC Ethernet Driver is a virtual Ethernet driver, similar to Macless. This DPAA Ethernet Driver has CSUM offload and zero-copy, features that cannot be achieved with Macless. This DPAA Ethernet Driver is used in complex scenarios, like IPSec offload or communication with USDPAA where memory space isolation between kernel space and underlying offload architecture (OH ports, SEC, USDPAA) space is needed. Some of these scenarios are presented in the Configuration chapter.

This type of DPAA Ethernet Driver needs two OH ports with VSP capabilities enabled in order to achieve zero-copy.

Offload NIC Ethernet Driver will be referred in this documentation as "oNIC".



7.6.1.2.3.4.1 Configuration

The main configuration options are offered, like in any other embedded Linux Ethernet driver, by the device tree configuration and the common interface offered by the Linux kernel (ifconfig, ethtool, etc.). All configuration capabilities are presented in this chapter.

7.6.1.2.3.4.1.1 Device tree configuration

oNIC Ethernet Driver has *fsl,dpa-ethernet-generic* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies an oNIC interface in a B4860QDS device tree looks like:

```
ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-generic", "fsl,dpa-ethernet-generic";
    fsl,qman-frame-queues-tx = <4000 8x8>;
    fsl,qman-frame-queues-rx = <4008 8x8>;
    fsl,oh-ports = <&oh1 &oh2>;
    fsl,disable_buff_dealloc;
    local-mac-address = [00 11 22 33 44 55];
};
```

The properties that the oNIC device tree node can have are:

- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues that forwards the frame buffers generated by oNIC to the Tx O/H port; it is recommended to have NR_CPUS Tx frame queues;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues called PCD queues that fetch the frames from the Rx O/H port to oNIC; besides these queues, oNIC uses the default and the error queue of the Rx O/H port; the PCD queues can be used in PCD schemes applied on the Rx O/H port.
- *fsl,oh-ports* – links to O/H port device tree representation. The first value is a reference to the Rx O/H port used on ingress path, the second value is a reference to the Tx O/H port used on egress path. The device tree representation for the O/H port is described below;
- *fsl,disable_buff_dealloc* - in some scenarios the TX OH port cannot enable VSP because of the OH port limitation (O/H port cannot have VSP capability and IP Fragmentation enabled in the same time), therefore another hardware module along the TX path should release the buffers generated by oNIC into the draining buffer pool. With this flag, oNIC will configure the TX OH port to NOT release the buffers into the draining buffer pool. If other hardware modules do not release the buffers into the draining pool the device will leak available memory. This is an optional attribute.
- *local-mac-address* - this is a virtual L2 address used to identify the driver in the Linux kernel network stack.

Note 1: Like Macless DPAA Ethernet Driver, oNIC does not have a MAC device to control, therefore the “fsl,fman-mac” property is missing from the device tree specification.

oNIC does not have a buffer pool specified in its device tree representation. It will use the default buffer pools specified in the O/H port device tree node. The O/H port device tree representation looks like:

```
oh2: dpa-fman0-oh@2 {
     compatible = "fsl,dpa-oh";
     fsl,bman-buffer-pools = <&bp1 &bp2 &bp3 &bp4>;
     fsl,qman-frame-queues-oh = <110 1 111 1>;
     fsl,fman-oh-port = <&fman0_oh2>;
};
```

The properties of the O/H port device tree node can have are:

- *fsl,bman-buffer-pools* - the default buffer pools managed by oNIC; up to four static buffer pools can be specified. If this O/H port has VSPs, these buffer pools will be used in the default VSP. This property is parsed by oNIC that will initialize the default buffer pools. No other entities should try to initialize these buffer pools;
- *fsl,qman-frame-queues-oh* - the default egress queues. The default queues are represented by one default frame queue and one error frame queue. This property will be parsed by oNIC that will initialize both queues and use them in case PCD frame queues are not defined;
- *fsl,fman-oh-port* - reference to FMan port device tree representation; this field is mandatory for Offline Port Driver, but it is not used by oNIC;

To activate VSP capability on a port, the user will have to configure the chosen node in the device tree. One valid entry looks like:

```
fman0_oh2-extd-args {
    cell-index = <0x1>;
    compatible = "fsl,fman-port-op-extended-args";
    vsp-window = <0x8 0x0>;
};
```

The most important property is:

- *vsp-window* - the number of VSPs that this port can have and the default VSP (starting VSP index). In the above example 8 VSPs can be added to the O/H port (with ids 0, 1, ..., 8).

The device tree node of the O/H port use statically defined buffer pools. oNIC will parse the device tree representation of the default buffer pools of the Rx O/H port and will seed them. The device tree representation of the buffer pool is:

```
bp7: buffer-pool@7 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <7>;
    fsl,bpool-ethernet-cfg = <0 256 0 192 0 0>;
    fsl,bpool-thresholds = <0x400 0xc00 0x0 0x0>;
};
```

oNIC parses *fsl,bpid* and *fsl,bpool-ethernet-cfg* properties to initialize its internal buffer pools structures. As in other DPAA Ethernet Drivers, *fsl,bpool-ethernet-cfg* property has the following meaning:

```
fsl,bpool-ethernet-cfg = <count size base_address>;
```

- *count* - represents the number of buffers from the buffer pool;
- *size* - buffer size;
- *base_address* - this value is ignored by oNIC.

Note 2: oNIC will seed its configured buffer pools when the Ethernet interface is raised up. Because of this the *fsl,bpool-ethernet-seeds* property is ignored.

Note 3: All the buffers are allocated in the kernel memory space, therefore the *base_address* value is ignored.

The example above declares a buffer pool with buffer pool ID 7, and describes a pool with 256 192-byte buffers, occupying the memory in the kernel space. The *base_address* with <0 0> value is ignored. It should be noted that the size of those parameters should be set by the root node's *#address-cells* and *#size-cells* properties.

7.6.1.2.3.4.1.2 Configuration available through Linux interfaces

Because oNIC does not manage a MAC device, some *ethtool* and *ifconfig* options will not be available. All Layer 3 configurations, like setting an IP address, are generally available.

7.6.1.2.3.4.2 Features

This chapter describes the features of the oNIC Ethernet Driver, their configuration options, fine-tuning and known limitations.

Hardware Checksum

The OH ports are able to validate and compute L3 and/or L4 checksums for certain protocols (mainly TCP/IP and UDP/IP). Because oNIC uses OH ports as communication points with the lower architecture, CSUM validation and computation are supported.

Zero Copy

oNIC, representing kernel space, works in memory "isolation". This capability is achieved through OH ports with VSP capability enabled, which are able to copy the frames from one memory space to another. oNIC code and the underlying offload architecture code are not aware of the existence of another memory space, therefore simplifying the communication procedures.

Scatter/Gather

The current implementation supports DPAA Scatter/Gather only on the TX path.

7.6.1.2.4 Offline Parsing Port Driver

Offline Parsing/Host Command Port, also known as *O/H Port* or simply *O/H*, is an FMan hardware module that is capable of multiplexing and de-multiplexing network traffic inside DPAA. Its behavior and programming model is similar to an “online” FMan port, supporting Parse-Classify-Distribute (PCD) function and buffer copy from one buffer pool to another. It is useful in complex DPAA scenarios, where plenty of hardware offload is desired, like IPSec or TCP fragmentation and reassembly.

Similar to most DPAA hardware modules, a Linux kernel driver for O/H Ports is needed for initialization and configuration. In this document, the Offline Parsing Port Driver will be referred to as *Offline Port Driver*.

7.6.1.2.4.1 Configuration

Like other DPAA Ethernet Drivers, the Offline Port Driver is a standard Linux platform device driver, whose configuration is available through device tree. Also, the support for Offline Port Driver is enabled through a kernel Kconfig option.

7.6.1.2.4.1.1 Kconfig Option

The Offline Port Driver is enabled in the Kbuild Linux configuration by default. It can be toggled via the following menuconfig option:

```
Device Drivers
    ---> Network device support
        ---> Ethernet driver support
            ---> Freescale devices
                ---> DPAA Ethernet
                    ---> Offline Ports support
```

7.6.1.2.4.1.2 Device Tree Configuration

The Offline Port Driver has *fsl,dpa-oh* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies an Offline Port Driver in a B4860QDS device tree is depicted below:

```
dpa-fman0-oh@2 {
    compatible = "fsl,dpa-oh";
    fsl,bman-buffer-pools = <&bp10>;
    fsl,qman-frame-queues-oh = <0x6e 0x1 0x6f 0x1>;
    fsl,qman-frame-queues-tx = <0x70 0x5>;
    fsl,qman-frame-queues-ingress = <base_id1 count1 ... base_idn countn>;
    fsl,qman-frame-queues-egress = <base_id1 count1 ... base_idn countn>;
    fsl,qman-channel-ids-egress = <channel_id1 ... channel_idn>;
    fsl,fman-oh-port = <&fman0_oh2>;
};
```

This device tree node resides inside *fsl,dpaa* device tree node, near DPAA Ethernet Driver device tree node specification.

Historically the offline port driver did not initialize the frame queues that entered and exited the OH port and relied on software components (Ethernet driver, USDPAA, other kernel modules) to initialize those queues. The offline port driver added capabilities to initialize ingress queues (queues that enter the Offline Port) simplifying the work for the Ethernet driver, but maintaining the same complexity in USDPAA and/or other kernel modules. Full capability for initializing both ingress and egress queues has been added, eliminating the dependency on USDPAA and/or other kernel modules. With it, complex offload architectures that have OH ports can be largely initialized by the offline port driver.

The additional device tree attributes, qman-frame-queues-ingress, qman-frame-queues-egress and qman-channel-ids-egress are optional and do not interfere with existing code. Therefore, backward compatibility is maintained.

Following are the properties that could be used within Offline Port Driver's device tree node:

- *fsl,bman-buffer-pools* - a list of buffer pools used by this O/H Port. The O/H Port will use these statically defined buffer pools in case it will do IP Fragmentation or buffer copy from one buffer pool to another. For IP Fragmentation, one incoming frame is divided into multiple frames. In order to copy data from a buffer pool to another, VSP (Virtual Storage Profile) properties must be configured on this port and the destination buffer pool is initialized and managed by the destination software entity. Therefore, these buffer pools are not initialized or managed by the Offline Port Driver. In some scenarios, this can be done by one of DPAA Ethernet Drivers. If the O/H Port does not use IP Fragmentation or VSP capabilities, this attribute will not be used;

- *fsl,qman-frame-queues-oh* - a list of base/count pairs of frame queues through which the frames are transmitted from the O/H Port to the next hardware or software module. These queues are not initialized, but are solely used as ID references for O/H Port initialization. This is necessary because the initialization of an FMan Port (online or offline) requires a TX error and a TX default frame queues. These frame queues must be initialized by other software module. This device tree property is mandatory;
- *fsl,qman-frame-queues-tx* - deprecated
- *fsl,qman-frame-queues-ingress* - a base/count pair of frame queues to be initialized by the Offline Port Driver that will fetch the frames from the hardware or software entity to the O/H Port.
- *fsl,qman-frame-queues-egress* - a list of base/count pairs of frame queues through which the frames are transmitted from the O/H Port to the next hardware or software module. These queues are connected only to DC portals therefore the property below (qman-channel-ids-egress) is mandatory. Frame queues connected to SW portals should be created from another software entity. Also, if a frame queue uses different initialization parameters it should be created by another software entity.
- *fsl,qman-channel-ids-egress* - a list of channel ids used together with "fsl,qman-frame-queues-egress" option to configure the egress frame queues.
- *fsl,fman-oh-port* - this is the reference to the O/H Port device tree node and has the same meaning as *fsl,fman-mac* from DPAA Ethernet Driver device tree specification. This attribute is mandatory.

Note 1: It should be noted that the Offline Port Driver does not initialize frame queues specified through *fsl,qman-frame-queues-oh* attribute. When the port is probed, the O/H Port is enabled and configured with the values from the device tree, but no action is taken on the declared frame queues.

Note 2: The statically defined buffer pool should be declared in a similar way as for the DPAA Ethernet Drivers.

The O/H Ports can be used in multiple scenarios as nodes that fetch traffic from multiple sources or as intermediate nodes that multiplexes incoming traffic to multiple destinations.

7.6.1.2.4.2 Features

The Offline Port Driver is an initialization driver for the O/H Ports. This chapter presents only the Offline Port Driver initialization capabilities, not the hardware features offered by the O/H Ports.

TX queues initialization

Before SDK 1.5, no frame queue referenced by the Offline Port Driver's device tree node was initialized. It was other driver's job to initialize the frame queues used by the O/H Port. As of SDK 1.5, *fsl,qman-frame-queues-tx* and *fsl,qman-channel-id* attributes were introduced. The frame queues referenced by the above mentioned *fsl,qman-frame-queues-tx* attribute are managed by the Offline Port Driver, decreasing thus the complexity of other drivers.

7.6.1.2.5 Link Management

The FMan has two types of ethernet controllers: 1G and 10G. The two types of ethernet conform to different standards (802.3 Clause 22 and Clause 45, respectively) for link management, so there are two corresponding types of MDIO controller for those standards. Each ethernet controller has its own MDIO device, however only one of each type are pinned out on currently shipping parts (10/2011). On P4080, for example, only the MDIO on FM1's first 1G MAC is pinned out, as well as the MDIO on FM1's 10G MAC. This is true even if FM1's first 1G MAC or 10G MAC is disabled.

On the 1G MACs, the MDIO controllers serve a second purpose -- configuring the SERDES link for SGMII between the MAC and the external PHY. This is done via management commands to an on-chip "TBI" PHY. This PHY is configured to sit at the address written to the TBIPA register in the MAC, and all MDIO transactions from that MAC to that address will be intercepted by the TBI PHY. This means that the setting of this register on the first 1G MAC is very relevant to system architects, as the address must not conflict with the address of an external PHY, or that PHY will not be reachable by MDIO management commands.

7.6.1.2.5.1 Device Tree

The MDIO nodes are described in the device tree, and look like this:

```
          mdio0: mdio@e1120 {
                    #address-cells = <1>;
                    #size-cells = <0>;
                    compatible = "fsl,fman-mdio";
                    reg = <0xe1120 0xee0>;
                    interrupts = <100 1 0 0>;
                    ethernet-phy@0 {
                              reg = <0x0>;
                    };
               tbi0: tbi-phy@8 {
                        reg = <0x8>;
                        device_type = "tbi-phy";
               };
      };
```

This is one of the 1G MDIO nodes for a P4080 (Not the DS, but more on that later). It specifies that there is a PHY at address 0, and the TBI PHY will sit at address 8. Note that it is the first MAC, which is the only one whose pins are connected outside the chip. The other 7 MACs would have a similar node, but with only a TBI PHY in each. This is how one would set up PHYs under a standard MDIO bus topology; the first MAC connects to all of the PHYs (up to 31 of them), and also has its assigned internal TBI PHY.

In order to associate an ethernet controller with a PHY, one uses the phy-handle property

7.6.1.2.5.2 Bootargs

```
fsl_fm_max_frm
```

Configure this to set the L2 Maximum Frame Size. This influences the Frame Manager FIFO size resources.

```
fsl_fm_rx_extra_headroom
```

Configure this to tell the Frame Manager to reserve some extra space at the beginning of a data buffer on the receive path, before Internal Context fields are copied. This is in addition to the private data area already reserved for driver internal use. The option does not affect in any way the layout of transmitted buffers. The default value (64bytes) offers best performance for the case when forwarded frames are being encapsulated (e.g. IPSec). For plain forwarding or termination cases, a value of zero is recommended for optimum performance.

The current size of the buffers in USDPAA dts files is 1728bytes (odd_cache_line * cache_line_size) which is calculated according to the default value (64bytes) of the dpa_extra_headroom. If the dpa_extra_headroom is set to 0 then the buffer size will be 1600bytes (also multiple of odd cache line). For any extra headroom values ranging between 1 and 128bytes the buffer size is 1728bytes. For values ranging between 129bytes and 256bytes, the buffer size is 1856bytes and so on.

7.6.1.2.5.3 Muxed MDIO

The Development Systems for the QorIQ product lines do not have a standard MDIO topology. In order to support the variety of configurations that are possible under the QorIQ product line, the PHYs often reside on riser cards, many of which use the same addresses; this means that it is not possible to keep all of the PHYs on the same bus. To work around this problem, the MDIO buses have been muxed so that, at any one time, only a subset of the PHYs will be connected to the MDIO buses. In order to send an MDIO transaction to a PHY, it is therefore necessary to first modify the MUX. This is done via a set of

"virtual" MDIO interfaces. Each MUX setting is considered a separate bus, each with its own PHYs. That means that the P4080DS's MDIO node looks like this:

```
   mdio0: mdio@e1120 {
          gpios = <&gpio0 0 0
                   &gpio0 1 0>;
          tbi0: tbi-phy@8 {
                  reg = <0x8>;
                  device_type = "tbi-phy";
          };
          p4080mdio0: p4080ds-mdio0 {
                  #address-cells = <1>;
                  #size-cells = <0>;
                  compatible = "fsl,p4080ds-mdio";
                  fsl,mdio-handle = <&mdio0>;
                  fsl,muxval = <0>;
                  phyrgmii: ethernet-phy@0 {
                          reg = <0x0>;
                  };
          };
   };
```

The bus now has a subnode, "p4080ds-mdio0", which represents the bus topology when the MUX is set to 0. That bus connects to the RGMII PHY at address 0. Other buses connect to the SGMII PHYs in the 3 riser card slots. It's important to note that it is not at all required to use a muxed MDIO bus on QorIQ products; this is only a solution to the problem of trying to access PHYs which can be in multiple places. The other important thing to note is that U-Boot modifies the device tree. In the example of the P4080DS, U-Boot uses the RCW to determine which interfaces are enabled, and which PHYs those interfaces are connected to. Therefore the connections in the dts file may not be the final arrangements of the PHYs.

7.6.1.2.6 Debugging

This chapter describes the debugging capabilities of the DPAA Ethernet driver.

7.6.1.2.6.1 Ethtool support

Various counters and statistics are exported through ethtool such as the number of interrupts per core, the number of frames per core, the number of available buffers, congestion detection, etc.

Following is an example of an ethtool output:

```
root@ls1043ardb:~# ethtool -S fm1-mac1
NIC statistics:
     interrupts [CPU 0]: 1
     interrupts [CPU 1]: 1
     interrupts [CPU 2]: 2
     interrupts [CPU 3]: 2
     interrupts [TOTAL]: 6
     rx packets [CPU 0]: 0
     rx packets [CPU 1]: 0
     rx packets [CPU 2]: 0
     rx packets [CPU 3]: 0
     rx packets [TOTAL]: 0
     tx packets [CPU 0]: 0
     tx packets [CPU 1]: 0
     tx packets [CPU 2]: 6
     tx packets [CPU 3]: 0
     tx packets [TOTAL]: 6
     tx recycled [CPU 0]: 0
```

```
     tx recycled [CPU 1]: 0
     tx recycled [CPU 2]: 0
     tx recycled [CPU 3]: 0
     tx recycled [TOTAL]: 0
     tx confirm [CPU 0]: 1
     tx confirm [CPU 1]: 1
     tx confirm [CPU 2]: 2
     tx confirm [CPU 3]: 2
     tx confirm [TOTAL]: 6
     tx S/G [CPU 0]: 0
     tx S/G [CPU 1]: 0
     tx S/G [CPU 2]: 0
     tx S/G [CPU 3]: 0
     tx S/G [TOTAL]: 0
     rx S/G [CPU 0]: 0
     rx S/G [CPU 1]: 0
     rx S/G [CPU 2]: 0
     rx S/G [CPU 3]: 0
     rx S/G [TOTAL]: 0
     tx error [CPU 0]: 0
     tx error [CPU 1]: 0
     tx error [CPU 2]: 0
     tx error [CPU 3]: 0
     tx error [TOTAL]: 0
     rx error [CPU 0]: 0
     rx error [CPU 1]: 0
     rx error [CPU 2]: 0
     rx error [CPU 3]: 0
     rx error [TOTAL]: 0
     bp count [CPU 0]: 128
     bp count [CPU 1]: 128
     bp count [CPU 2]: 128
     bp count [CPU 3]: 128
     bp count [TOTAL]: 512
     rx dma error: 0
     rx frame physical error: 0
     rx frame size error: 0
     rx header error: 0
     rx csum error: 0
     qman cg_tdrop: 0
     qman wred: 0
     qman error cond: 0
     qman early window: 0
     qman late window: 0
     qman fq tdrop: 0
     qman fq retired: 0
     qman orp disabled: 0
     congestion time (ms): 0
     entered congestion: 0
     congested (0/1): 0
```

7.6.1.2.6.2 Read/Write of FMan Registers

Most of the FMan configuration registers are mapped into the system memory space. Efficient debugging and testing can be done by making read/write operations on the registers through specialized tools. For example, the number of pause frames received on a particular MAC device can be computed summing the base relative address of every component:

```
0xffe000000 (the address of the SoC) +
   0x400000 (FMan 1) +
```

```
    0xe8000 (MAC 5) +
      0x234 (RX pause frames) =
-------------------------------------
0xffe4e8234
```

A memory print of the 0xffe4e8234 address will display the number of pause frames received by the fifth MAC device from the first FMan on a P4080DS platform.

The entire memory map for all mapped registers of the DPAA hardware components ca be found in each platform's Reference Manual.

7.6.1.2.6.3 Sysfs support

To enable Sysfs in the Linux kernel one must set the CONFIG_SYSFS option in Kconfig. The DPAA Ethernet Driver exports a series of information in Sysfs such as the buffer pool IDs, the frame queue IDs used by the interface and the MAC registers as shown in the following examples:

```
root@p4080ds:~# cat /sys/devices/fsl,dpaa.16/ethernet.18/net/fm2-gb0/bpids
32
root@p4080ds:~# cat /sys/devices/fsl,dpaa.16/ethernet.17/net/fm1-gb1/fqids
Rx error: 262
Rx default: 263
Rx PCD: 14464 - 14591
Tx confirmation (mq): 264 - 271
Tx error: 272
Tx default confirmation: 273
Tx: 274 - 281
```

7.6.1.2.7 Adding support for DPAA Ethernet in Topaz Hypervisor

We start from the assumption that hv.dts file for the platform has already been created starting from an existing platform. The practice is to place the device tree under the following folder hierarchy: hv-cfg/"platform_name"/"RCW_name" It means that the RCW gives the number and types of ports that will be added in hv.dts. We will consider T1040RDB hv.dts as a reference throughout this guide.

Following are the steps to be followed in order to add DPAA ports (private, shared-mac and macless) in hv.dts:

1. Add bman-portal and qman-portal nodes under "part 1" node. E.g.:

```
                bman-portal@0 {
                    device = "/bman-portals/bman-portal@0";
                };

                qman-portal@0 {
                    device = "/qman-portals@ff6000000/qman-portal@0";

                    stash-mem {
                        liodn-index = <1>;
                        dma-window = <&dw_linux1>;
                        operation-mapping = <0>;
                        stash-dest = <3>;
                    };

                    stash-dqrr {
                        liodn-index = <0>;
                        dma-window = <&dw_dqrr_qportal0>;
```

```
                operation-mapping = <0>;
                stash-dest = <3>;
            };
        };
```

Make sure dma-window "dw_dqrr_qportal0" exists under "dma-windows" node:

```
        // DMA window for stash_dqrr for qman-portal0
        dw_dqrr_qportal0: window3 {
            compatible = "dma-window";
            guest-addr = <0xf 0xf6000000>;
            size = <0 0x4000>;
        };
```

All bman-portal and qman-portal nodes must be added except at least one which should be added to the second partition. For T1040RDB, bman-portal@1c000 and qman-portal@1c000 were added in the second partition. The list of available bman-portal and qman-portal nodes for T1040RDB can be retrieved from the following dtsi file included by the platform device tree: arch/powerpc/boot/dts/fsl/t1040si-post.dtsi bman-portals are located under bportals node and qman-portals under qportals node.

2. For the current RCW that is used, add the appropriate dpaa ethernet node under "part 1" parent node. For T1040RDB the following ports are added:

```
        // FMAN0 RGMII -- ethernet 3 assigned to this partition
        dpa-ethernet@3 {
            device = "/fsl,dpaa/ethernet@3";
        };


        // FMAN0 RGMII -- ethernet 4 assigned to this partition
        dpa-ethernet@4 {
            device = "/fsl,dpaa/ethernet@4";
        };
```

The full list of available ethernet ports can be retrieved from the platform device tree: arch/powerpc/boot/dts/t1040rdb.dts

3. bman, qman, fman0 and fman1 (in case there are two FMANs) nodes must be also added under "part 1" parent node. See T1040RDB example.
4. fman0 and fman1 (if applicable) must be added under "portal-devices" node. See T1040RDB example.
5. Under "node-update" node make sure you define at least two buffer pools for macless and shared-mac interfaces. While here, make sure gpma1 is defined and guest-addr reserves the inter-partition memory area. Also check that dw_linux1 and dw_linux2 both have sub-window nodes with the same memory area region matching the inter-patition area. See T1040RDB example.
6. Make sure "fsl,dpaa" node exists under "node-update" node and check or create a macless and a shared-mac interface. The compatible string for the "fsl,dpaa" node must be: compatible = "fsl,t1040-dpaa", "fsl,dpaa"; replace t1040 with the name of the platform you want to add. The compatible string for the macless interface must be: compatible = "fsl,t1040-dpa-ethernet", "fsl,dpa-ethernet-macless"; The compatible string for the macless interface must be: compatible = "fsl,t1040-dpa-ethernet", "fsl,dpa-ethernet-shared"; Make sure the macless node includes the local-mac-address identifier. See T1040RDB example.

7. Make sure "fsl,dpaa" node exists under "node-update-phandle" node. Here the buffer pools are associated with macless and shared-mac interfaces. See T1040RDB example.
8. Under "part 1" node check or add the following nodes: bman-bpids, qman-fqids@0, qman-fqids@1, qman-pools, qman-cgrids The values used by these nodes can be retrieved from arch/powerpc/boot/dts/fsl/qoriq-dpaa-res3.dtsi For T1040RDB two partitions were created therefore the resources, bpids, fqids, etc were splitted among the two partitions. In case more partitions are created the resources must be splitted accordingly. See T1040RDB example.
9. Repeat steps 3-8 for "part 2" node in case of 2 partitions scenario. The same steps must be followed in case more than 2 partitions are added. Exception for step 6: Two macless nodes must be added under "fsl,dpaa" node. The first node is the "pair" of the macless interface in the first partition and the second is the interface connected to the shared-mac port (the port shared with the first partition). See T1040RDB example.

7.6.1.2.8 MACsec

Introduction

This chapter provides information about the MACsec kernel module and user space API that can be used to configure and control the MACsec hardware block inside the Frame Manager. The Quick Start Guide lists the steps that need to be taken in order to enabled MACsec after integrating the API into the hostapd and wpa_supplicant applications. The API Reference section details the user space MACsec API and gives examples on how to use it.

Intended Audience

This chapter is intended for software developers and architects who want to develop software applications that implement the 802.1X and 802.1AE (MACsec) protocols.

7.6.1.2.8.1 MACsec User Space API Reference

The following API is implemented in the hostapd and wpa_supplicant user space applications and can be reached by including the src/drivers/dpaa_utils_macsec.h header.

MACsec API data structures

1. enable_macsec_t

Member's description:

- if_name - char* that holds the name of the interface on which MACsec is to be enabled
- if_name_length - size_t that holds the length of the if_name field, including the terminating null character
- config_unknown_sci_treatment - boolean that selects between the default value and the one in unknown_sci_treatment
- unknown_sci_treatment - enum that selects how frames received with unknown SCI are treated (default is DISCARD_BOTH - discarded on both the controlled and uncontrolled ports)
- config_invalid_tag_treatment - boolean that selects between the default value and the one in deliver_uncontrolled
- deliver_uncontrolled - boolean that selects if frames received with an invalid SecTAG or a zero PN value should be delivered on the uncontrolled port (default is FALSE)
- config_kay_frame_treatment - boolean that selects between the default value and the one in discard_uncontrolled
- discard_uncontrolled - boolean that selects if frames received with the Encryption (E) bit set and the Changed Text (C) bit clear (reserved for use by the KaY) should be discarded on the uncontrolled port (default is FALSE)
- config_untag_treatment - boolean that selects between the default value and the one in untag_treatment
- untag_treatment - enum that selects how frames received without the MAC SecTAG are treated (default is UNTAG_DELIVER_UNCTRL_DISCARD_CTRL)
- config_pn_exhaustion_threshold - boolean that selects between the default value and the one in pn_threshold
- pn_threshold - u32 that sets the TX PN exhaustion threshold (default is 0xffffffff)

- config_keys_unreadable - boolean that selects if the RX and TX keys and hash values should be unreadable (default is FALSE)
- config_sectag_without_sci - boolean that selects if the maximum frame length should not consider the SCI's size in the SecTAG (default is FALSE)
- config_exception - boolean that selects between the default values and the ones in enable_exception and exception
- enable_exception - boolean that selects if the exception mask set by the exception field should be enabled or not
- exception - enum that defines the mask of the exception that is to be enabled or disabled (default is SINGLE_BIT_ECC | MULTI_BIT_ECC - both masks are enabled)

2. enable_secy_t

Member's description:

- macsec_id - int value returned by the dpa_macsec_enable() call that identifies one MACsec instance per interface
- sci - 64bit value made from the MAC address of the interface on which MACsec has been enabled, concatenated with a Port Identifier
- config_insertion_mode - boolean that selects between the default value and the one in sci_insertion_mode
- sci_insertion_mode - enum that selects if the SCI will be included in the SecTag or not (default is SCI_INSERTION_MODE_EXPLICIT_SECTAG - the SCI is always included)
- config_protect_frames - boolean that selects between the default value and the one in protect_frames
- protect_frames - boolean that selects if the packet will be encapsulated with MACsec header (default is TRUE)
- config_replay_window - boolean that selects between the default values and the ones in replay_window and replay_protect
- replay_protect - boolean that selects if the replay protection algorithm is enabled or not (default is FALSE)
- replay_window - unsigned integer that represents the replay window dimension (default is 0)
- config_validation_mode - boolean that selects between the default value and the one in validate_frames
- validate_frames - enum that selects the behaviour of MACsec from frames validation point of view (default is VALID_FRAME_BEHAVIOR_STRICT - strictly filter out invalid frames)
- config_confidentiality - boolean that selects between the default values and the ones in confidentiality_enable and confidentiality_offset
- confidentiality_enable - boolean that selects if the packages will be encrypted or not (default is FALSE)
- confidentiality_offset - unsigned integer that represents the offset from which the data will be encrypted (default is 0)
- config_point_to_point - boolean that selects if MACsec must be configured only for point-to-point communication (default is FALSE)
- config_exception - boolean that selects between the default values and the ones in enable_exception and exception
- enable_exception - boolean that selects if the exception mask set by the exception field should be enabled or not
- exception - enum that defines the mask of the exception that is to be enabled or disabled (default is FRAME_DISCARDED)
- config_event - boolean that selects between the default values and the ones in enable_event and event
- enable_event - boolean that selects if the event mask set by the event field should be enabled or not
- event - enum that defines the mask of the event that is to be enabled or disabled (default is NEXT_PN)

MACsec API functions

1. dpa_macsec_enable

```
int dpa_macsec_enable(enable_macsec_t en_macsec)
```

Description:

- used to configure and enable MACsec on a certain interface

Parameters:

- en_macsec - MACsec data structure including the name of the interface to enable MACsec on and several configuration options

Return:

- returns the macsec_id that will be used in further calls of the API's functions
- returns a negative value in case something went wrong

2. dpa_macsec_secy_en

```
int dpa_macsec_secy_en(enable_secy_t enable_secy)
```

Description:

- used for configuring and creating SecY; also creates a corresponding TxSc (secure channel for TX)
- sets the default values for cipher suite, SCI insertion mode, protect frames, replay protection, replay window dimension, confidentiality, confidentiality offset and point-to-point connection, as IEEE802.1AE-2006 states

Parameters:

- enable_secy - SecY data structure

Return:

- returns the sc_id allocated by MACsec (a value from 15 to 0, or simply 0, if the setup is point-to-point) that uniquely identifies the SecY
- returns a negative value in case something went wrong

3. dpa_macsec_secy_create_tx_sa

```
int dpa_macsec_secy_create_tx_sa(int macsec_id, u8 an, u8 *sak, u32 sak_len)
```

Description:

- used for creating a TxSa (secure association for TX)

Parameters:

- macsec_id - MACsec interface identifier
- an - association number
- sak - secure association key (the key used for encryption)
- sak_len - length of the key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

4. dpa_macsec_secy_activate_tx_sa

```
int dpa_macsec_secy_activate_tx_sa(int macsec_id, u8 an)
```

Description:

- used for activating the secure association for TX channel

Parameters:

- macsec_id - MACsec interface identifier
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

5. dpa_macsec_secy_create_rx_sc

```
int dpa_macsec_secy_create_rx_sc(int macsec_id, u64 sci)
```

Description:

- creates a RxSc (secure channel for RX)

Parameters:

- macsec_id - MACsec interface identifier
- sci - a 64 bit value made of the MAC address of the destination(48 bits) and a port number(16 bits)

Return:

- returns the rx_sc_id allocated by MACsec (a value from 15 to 0, or simply 0, if the setup is point-to-point) that uniquely identifies the peer to whom we are discussing MACsec
- returns a negative value in case something went wrong

6. dpa_macsec_secy_create_rx_sa

```
int dpa_macsec_secy_create_rx_sa(int macsec_id, u32 rx_sc_id, u8 an, u32 lpn, u8 *sak, u32 sak_len)
```

Description:

- creates RxSa (secure association for RX)

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- lpn - lowest packet number, value used in anti-replay algorithm
- sak - secure association key (the key used for decryption)
- sak_len - length of the key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

7. dpa_macsec_secy_activate_rx_sa

```
int dpa_macsec_secy_activate_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for activating the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier

- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

8. dpa_macsec_secy_disable_rx_sa

```
int dpa_macsec_secy_disable_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for disabling the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

9. dpa_macsec_secy_delete_rx_sa

```
int dpa_macsec_secy_delete_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for deleting the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

10. dpa_macsec_secy_delete_rx_sc

```
int dpa_macsec_secy_delete_rx_sc(int macsec_id, u32 rx_sc_id)
```

Description:

- used for deleting the secure channel for RX

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call

Return:

- returns 0 on success or a negative value in case something went wrong

11. dpa_macsec_secy_delete_tx_sa

```
int dpa_macsec_secy_delete_tx_sa(int macsec_id, u8 an)
```

Description:

- used for deleting the secure association for TX channel

Parameters:

- macsec_id - MACsec interface identifier
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

12. dpa_macsec_secy_disable

```
int dpa_macsec_secy_disable(int macsec_id)
```

Description:

- used for deleting the SecY and disabling the associated TxSc

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

13. dpa_macsec_disable

```
int dpa_macsec_disable(int macsec_id)
```

Description:

- used for disabling MACsec

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

14. dpa_macsec_disable_all

```
int dpa_macsec_disable_all(int macsec_id)
```

Description:

- used as a single call for disabling all that was enabled for MACsec to work
- is the equivalent for calling dpa_macsec_secy_disable_rx_sa, dpa_macsec_secy_delete_rx_sa, dpa_macsec_secy_delete_rx_sc, dpa_macsec_secy_delete_tx_sa, dpa_macsec_secy_disable and dpa_macsec_disable one by one

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

15. dpa_macsec_secy_get_txsc_phys_id

```
int dpa_macsec_secy_get_txsc_phys_id(int macsec_id)
```

Description:

- obtain the ID associated with the TX SC by the Fman

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns the tx_sc_id corresponding to the SecY from Fman
- returns a negative value in case something went wrong

16. dpa_macsec_secy_modify_txsa_key

```
int dpa_macsec_secy_modify_txsa_key(int macsec_id, u8 an, u8 *sak, u32 sak_len)
```

Description:

- used to change the encryption key

Parameters:

- macsec_id - MACsec interface identifier
- an - association number
- sak - new secure association key (the key used for encryption)
- sak_len - length of the new key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

17. dpa_macsec_secy_modify_rxsa_key

```
int dpa_macsec_secy_modify_rxsa_key(int macsec_id, u32 rx_sc_id, u8 an, u8 *sak, u32 sak_len)
```

Description:

- used to change the decryption key

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- sak - new secure association key (the key used for decryption)
- sak_len - length of the new key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

18. dpa_macsec_secy_get_tx_sa_active_an

```
int dpa_macsec_secy_get_tx_sa_active_an(int macsec_id)
```

Description:

- used to get the active association number for this TxSc

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns the association number on success or a negative value in case something went wrong

19. dpa_macsec_secy_get_rxsc_phys_id

```
int dpa_macsec_secy_get_rxsc_phys_id(int macsec_id, u32 rx_sc_id)
```

Description:

- obtain the ID associated with the RX SC by the Fman

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call

Return:

- returns the rx_sc_id corresponding to the peer's SecY from Fman
- returns a negative value in case something went wrong

20. dpa_macsec_secy_rx_sa_update_npn

```
int dpa_macsec_secy_rx_sa_update_npn(int macsec_id, u32 rx_sc_id, u8 an, u32 pn)
```

Description:

- used to set the next packet number that MACsec should handle on RX

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- pn - packet number

Return:

- returns 0 on success or a negative value in case something went wrong

21. dpa_macsec_secy_rx_sa_update_lpn

```
int dpa_macsec_secy_rx_sa_update_lpn(int macsec_id, u32 rx_sc_id, u8 an, u32 pn)
```

Description:

- used to set the lowest packet number that MACsec should handle on RX

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- pn - packet number

Return:

- returns 0 on success or a negative value in case something went wrong

22. dpa_macsec_get_revision

```
int dpa_macsec_get_revision(int macsec_id)
```

Description:

- used to find the current revision of MACsec

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns the revision corresponding to the macsec_id
- returns a negative value in case something went wrong

23. dpa_macsec_set_exception

```
int dpa_macsec_set_exception(int macsec_id, bool enable_ex, macsec_exception ex)
```

Description:

- used to enable/disable a trigger for a certain exception

Parameters:

- macsec_id - MACsec interface identifier
- enable_ex - bool that will select if the exception given by the third argument will be enabled or not
- ex - the exception for wich a trigger will be activated/deactivated

Return:

- returns 0 on success or a negative value in case something went wrong

MACsec API usage examples

1. Basic usage

```
        enable_macsec_t en_macsec;
        enable_secy_t enable_secy;
        int i, rx_sc_id;
        u8 an = 0;
        u32 lpn = 0;
        u32 sa_key_len = 32;
        u16 port_src = 1;
        u16 port_dst = 1;
        u64 sci_src; //MAC + port - 64b
        u64 sci_dest; //MAC + port - 64b

        u8 *sa_key = malloc(sa_key_len * sizeof(u8));
        for (i = 0; i < sa_key_len; i ++)
            sa_key[i] = 0x12;

        sci_src = (mac_src << 16) | port_src;
        sci_dest = (mac_dst << 16) | port_dest;

        en_macsec.if_name = ifname;
        en_macsec.if_name_length = strlen(ifname) + 1;

        en_macsec.config_unknown_sci_treatment = FALSE;
        en_macsec.config_invalid_tag_treatment = FALSE;
```

```
en_macsec.config_kay_frame_treatment = FALSE;
en_macsec.config_untag_treatment = FALSE;
en_macsec.config_pn_exhaustion_threshold = FALSE;
en_macsec.config_keys_unreadable = FALSE;
en_macsec.config_sectag_without_sci = FALSE;
en_macsec.config_exception = FALSE;
en_macsec.enable_exception = FALSE;

macsec_id = dpa_macsec_enable(en_macsec);

enable_secy.macsec_id = macsec_id;
enable_secy.sci = sci_src;

enable_secy.config_insertion_mode = FALSE;
enable_secy.config_protect_frames = FALSE;
enable_secy.config_replay_window = FALSE;
enable_secy.config_validation_mode = FALSE;
enable_secy.config_confidentiality = FALSE;
enable_secy.config_point_to_point = FALSE;
enable_secy.config_exception = FALSE;
enable_secy.config_event = FALSE;

dpa_macsec_secy_en(enable_secy);
dpa_macsec_secy_create_tx_sa(macsec_id, an, sa_key, sa_key_len);
dpa_macsec_secy_activate_tx_sa(macsec_id, an);

rx_sc_id = dpa_macsec_secy_create_rx_sc(macsec_id, sci_dest);
dpa_macsec_secy_create_rx_sa(macsec_id, rx_sc_id, an, lpn, sa_key, sa_key_len);
dpa_macsec_secy_activate_rx_sa(macsec_id, rx_sc_id, an);

/* ... */

free(sa_key);

dpa_macsec_disable_all(macsec_id);

/* Or:
* dpa_macsec_secy_disable_rx_sa(macsec_id, rx_sc_id, an);
* dpa_macsec_secy_delete_rx_sa(macsec_id, rx_sc_id, an);
* dpa_macsec_secy_delete_rx_sc(macsec_id, rx_sc_id);
* dpa_macsec_secy_delete_tx_sa(macsec_id, an);
* dpa_macsec_secy_disable(macsec_id);
* dpa_macsec_disable(macsec_id);
*/
```

2. Enable confidentiality

Enable the confidentiality flag and set the confidentiality offset in the enable_secy_t structure.

```
enable_secy.macsec_id = macsec_id;
enable_secy.sci = sci_src;

enable_secy.config_insertion_mode = FALSE;
enable_secy.config_protect_frames = FALSE;
enable_secy.config_replay_window = FALSE;
enable_secy.config_validation_mode = FALSE;

enable_secy.config_confidentiality = TRUE;
```

```
        enable_secy.confidentiality_enable = TRUE;
        enable_secy.confidentiality_offset = 0;

        enable_secy.config_point_to_point = FALSE;
        enable_secy.config_exception = FALSE;
        enable_secy.config_event = FALSE;
```

3. Modify the TxSa and the RxSa keys

Generate a new set of keys and call the appropriate API functions.

```
        for(i = 0; i < sa_key_len; ++i) {
            txsa_key[i] = 0x23;
        }

        dpa_macsec_secy_modify_txsa_key(macsec_id, an, txsa_key, sa_key_len);

        for(i = 0; i < sa_key_len; ++i) {
            rxsa_key[i] = 0x57;
        }

        dpa_macsec_secy_modify_rxsa_key(macsec_id, rx_sc_id, an, rxsa_key, sa_key_len);
```

7.6.1.2.8.2 MACsec Quick Start Guide

After integrating the MACsec API with hostapd and wpa_supplicant, as exemplified in the MACsec User Space API Reference on page 1051 chapter, follow these steps to run the applications.

Software Installation and Build

- Compile MACsec as a module:

```
        Device Drivers
            +-> Network device support (NETDEVICES [=y])
                +-> Ethernet driver support (ETHERNET [=y])
                    +-> Freescale devices (NET_VENDOR_FREESCALE [=y])
                        +-> DPAA Ethernet (FSL_SDK_DPAA_ETH [=y])
                            +-> DPAA Macsec [=m]
```

- Increase the FMan maximum frame size to 1554 bytes. This is needed in order to accommodate the MACsec header without decreasing the MTU:

```
        Device Drivers
            +-> Network device support (NETDEVICES [=y])
                +-> Ethernet driver support (ETHERNET [=y])
                    +-> Freescale devices (NET_VENDOR_FREESCALE [=y])
                        +-> Frame Manager Support
                            +-> Freescale Frame Manager (datapath) support - SDK driver
(FSL_SDK_FMAN [=y])
                                +-> Maximum L2 frame size [=1554]
```

- Set the appropriate FMan version (needed by B-series and T-series platforms):

```
        Device Drivers
            +-> Network device support (NETDEVICES [=y])
                +-> Ethernet driver support (ETHERNET [=y])
                    +-> Freescale devices (NET_VENDOR_FREESCALE [=y])
                        +-> Frame Manager Support
                            +-> Freescale Frame Manager (datapath) support - SDK driver
(FSL_SDK_FMAN [=y])
                                +-> FMAN_V3L like T1040, T1042, T1020, T1022 [=y]
```

- Build the hostapd and wpa_supplicant

  Add the recipes to your Yocto build by adding the following lines to your `conf/local.conf` file:

```
        IMAGE_INSTALL_append = " hostapd"
        IMAGE_INSTALL_append = " wpa-supplicant"
```

  In order to integrate the MACsec API with the hostapd application, apply the `0001-Add-DPAA-MACsec-API.patch` patch from the recipe directory (`meta-freescale/recipes-connectivity/hostapd/hostapd-2.4`) on top of the hostapd version 2.4 repository. Build independently after committing the necessary changes.

Execution setup

- Generate a set of OpenSSL keys and certificates for the 802.1X authentication.
- Create a configuration file for the HostAP.

  A separate configuration file is needed for each interface on which the hostapd will run. hostapd.conf example:

```
        ctrl_interface=/var/run/hostapd
        ctrl_interface_group=0
        use_pae_group_addr=1

        # Replace the following with the appropriate interface name
        interface=fm1-gb3

        driver=wired
        logger_syslog=-1
        logger_syslog_level=2
        logger_stdout=-1
        logger_stdout_level=2
        dump_file=/tmp/hostapd.dump

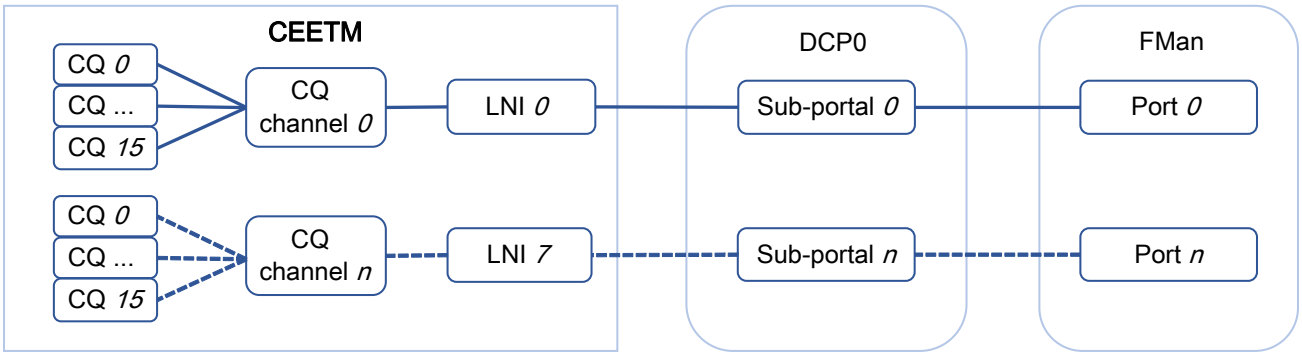
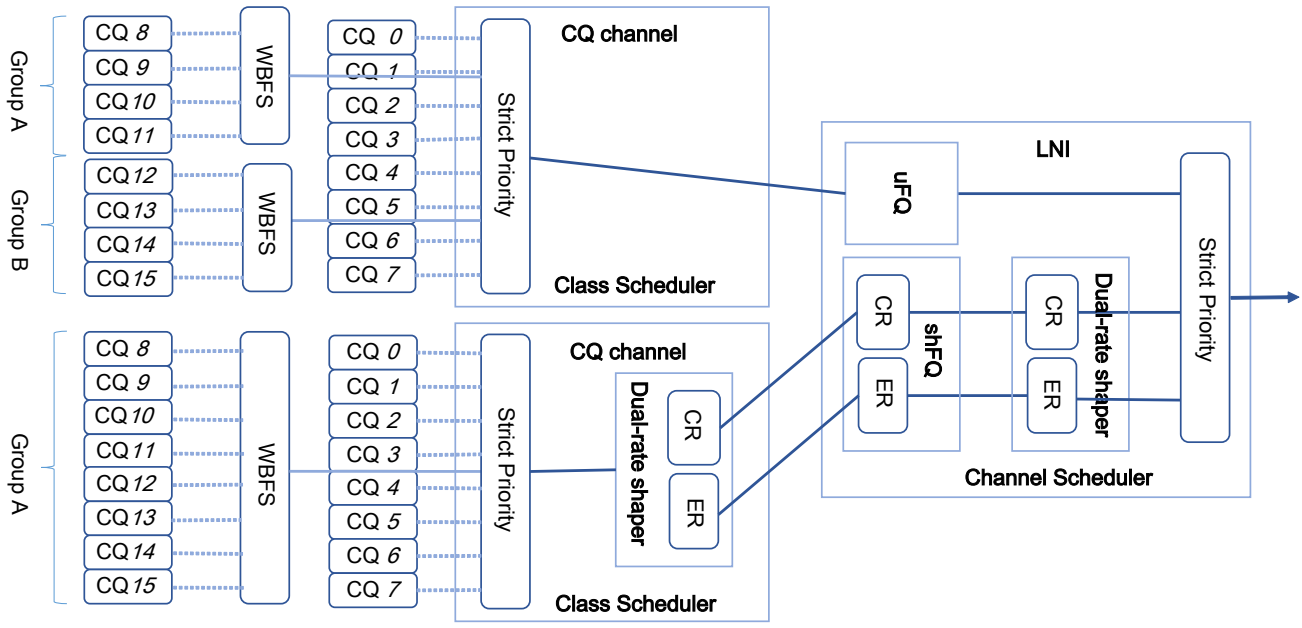
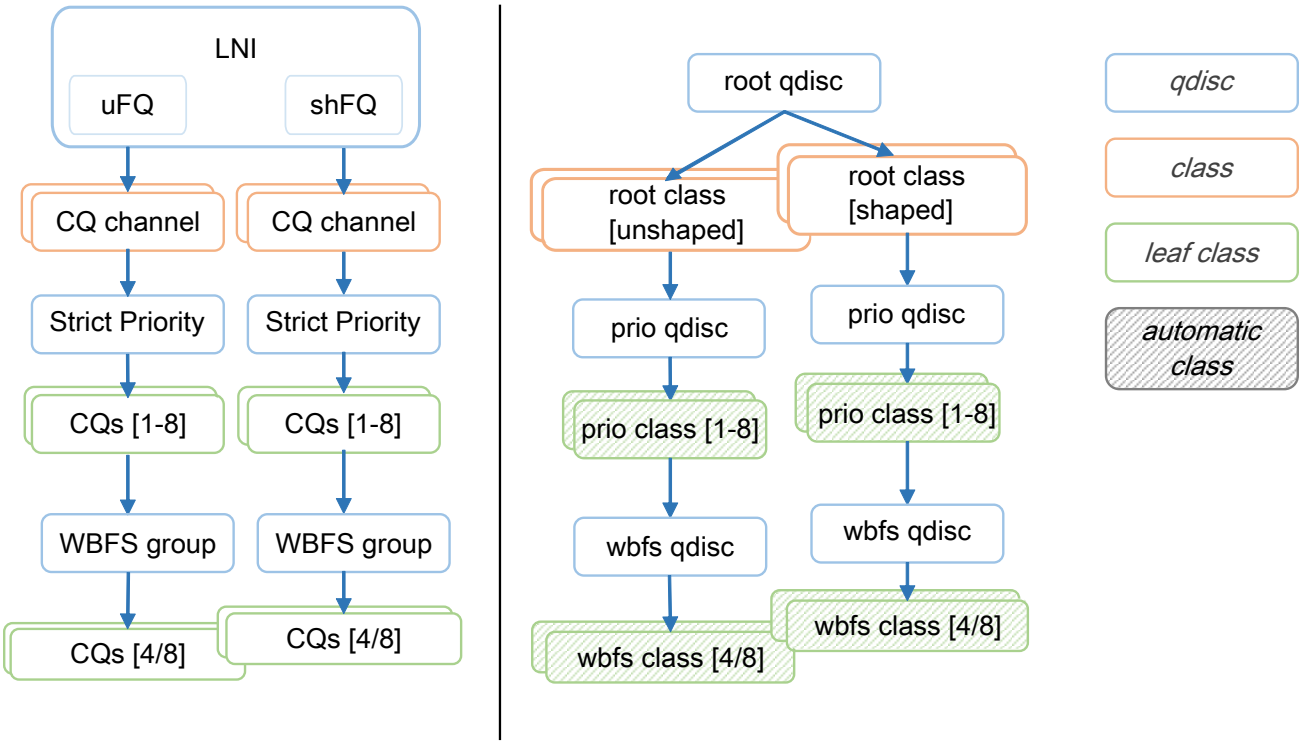
        eapol_version=2
        ieee8021x=1
        eap_server=1

        # Replace the following with the path to the eap_user_file
        eap_user_file=/etc/hostapd.eap_user

        # Replace the following with your CA certificate path
        ca_cert=/etc/server.crt
        server_cert=/etc/server.crt
        private_key=/etc/server.key
```

- Create an eap_user_file for the HostAP.

  `hostapd.eap_user` example:

```
# Phase 1 users
* PEAP
# Replace the following with your desired credentials
"test" MSCHAPV2,MD5,PEAP "password" [2]
```

- Create a configuration file for the wpa_supplicant.

  `wpa_supplicant.conf` example:

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
ap_scan=0
fast_reauth=1
eapol_version=2

network={
    ssid=""
    key_mgmt=IEEE8021X
    scan_ssid=0
    eap=PEAP
    priority=100
    phase2="auth=MSCHAPV2"

    # Replace the following with the credentials from the HostAP's eap_user_file
    identity="test"
    password="password"

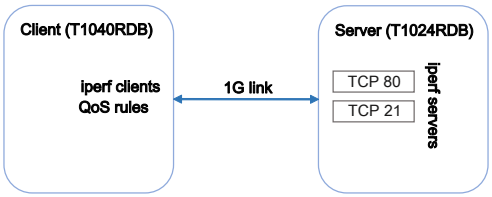
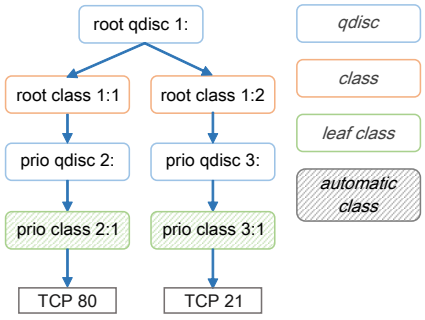
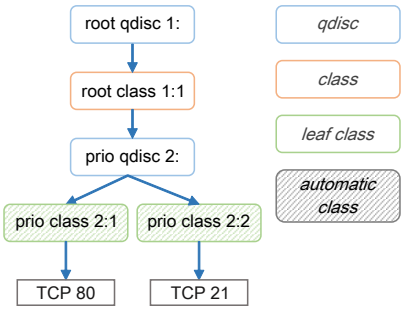
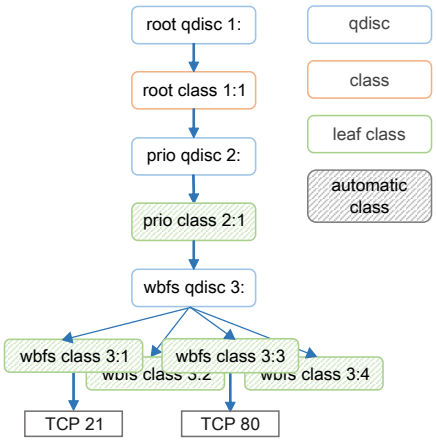
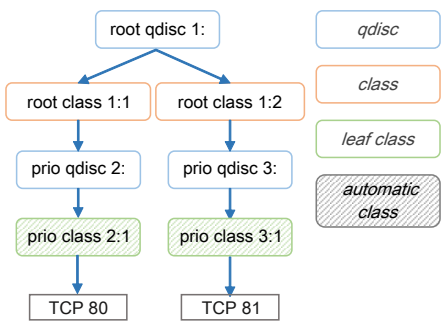
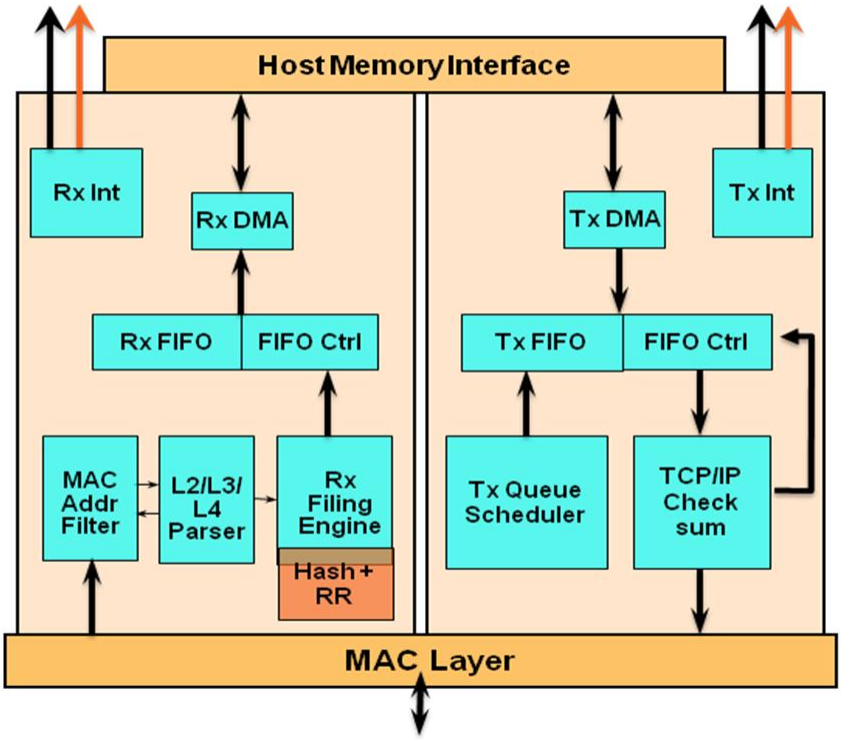
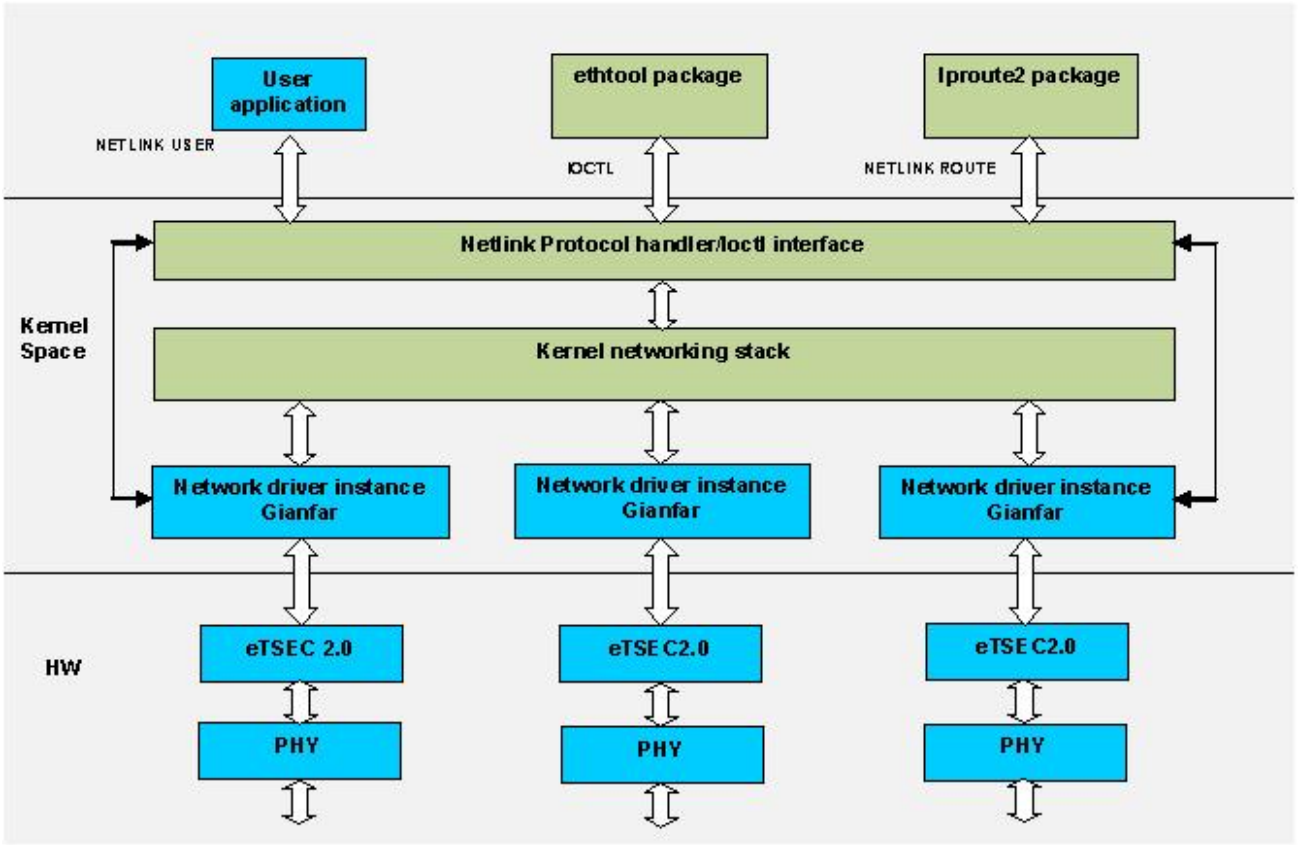
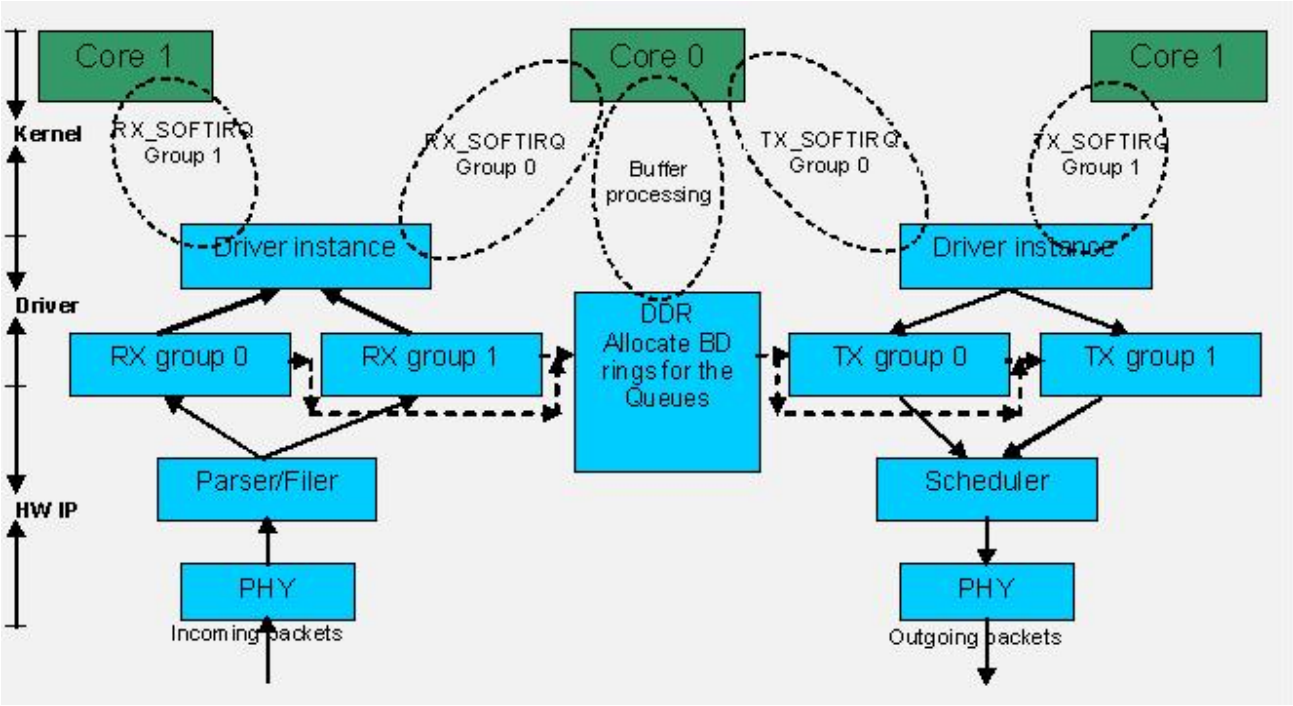
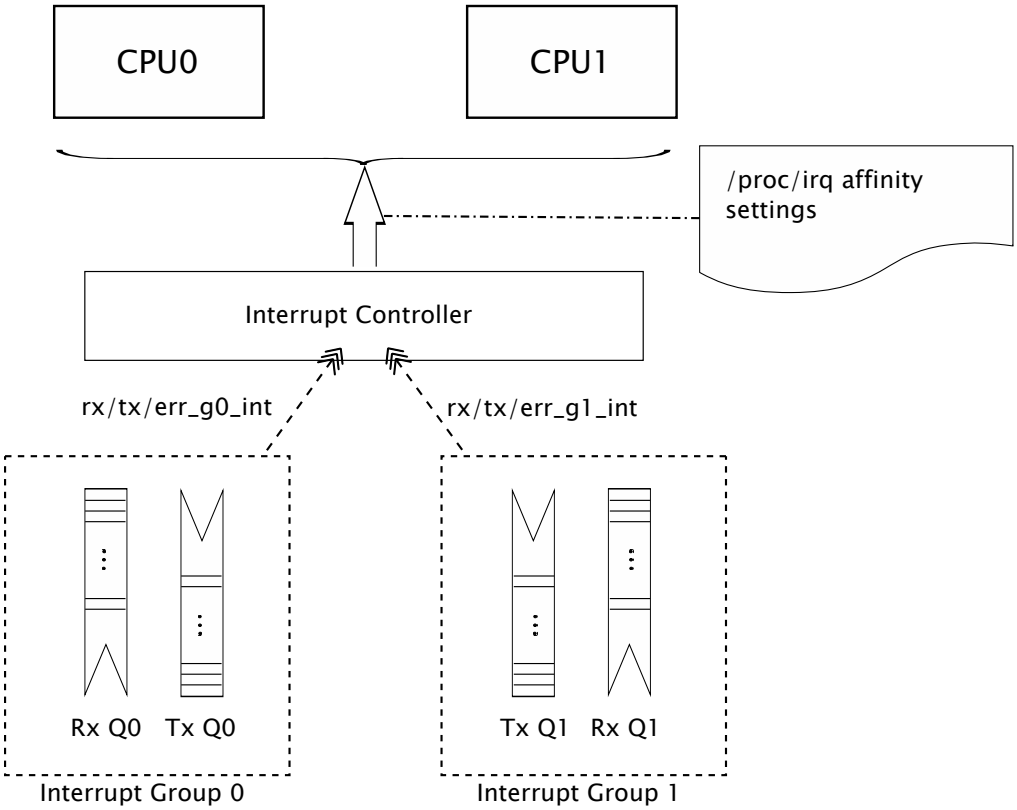
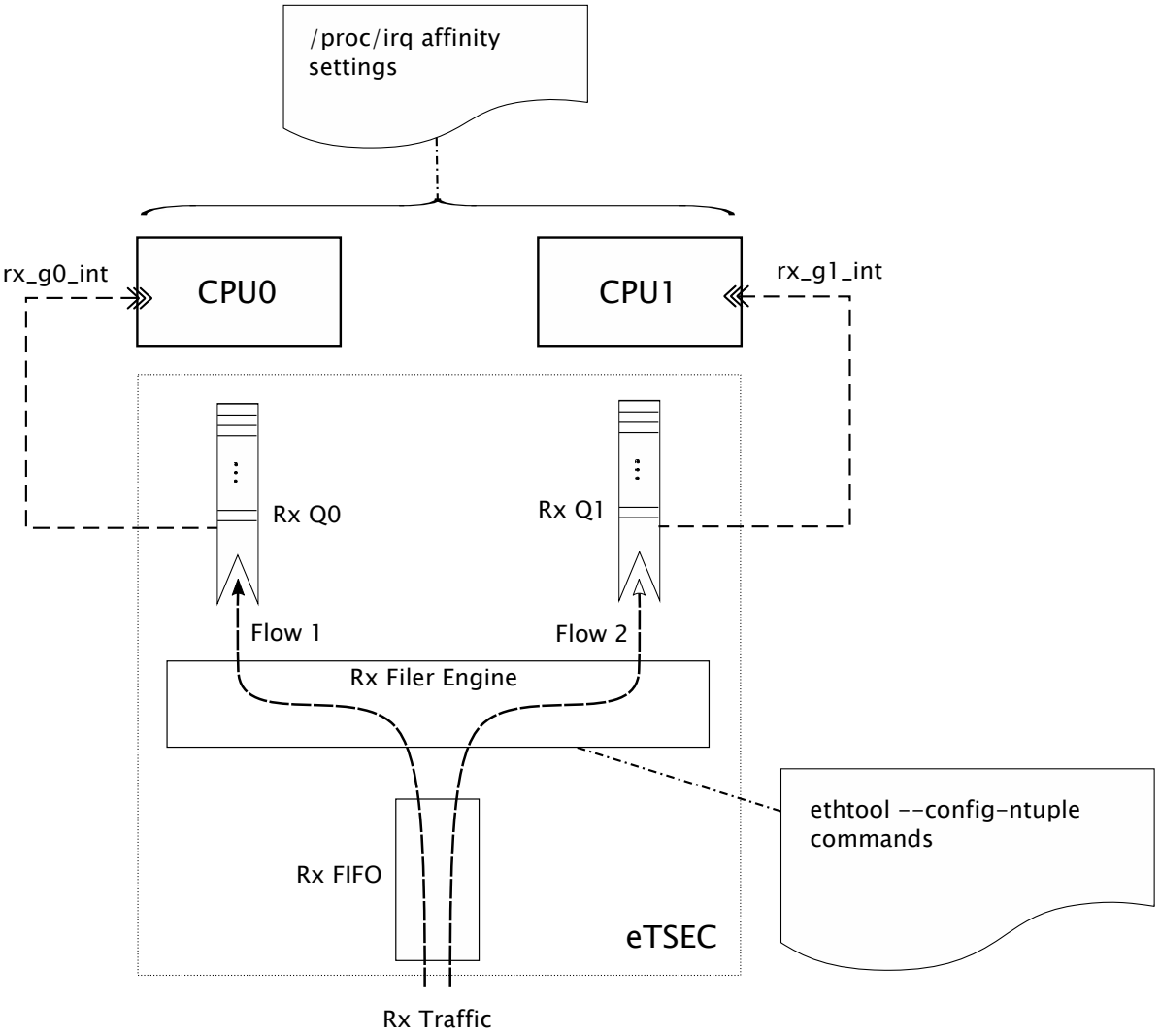
    # Replace the following with your CA certificate path
    ca_cert="/etc/server.crt"
}
```

- Boot the board according to the platform's documentation.
- Insert the `fsl-dpa-macsec` module.

  Specify in a comma separated list the interfaces on which MACsec will be enabled:

```
modprobe fsl-dpa-macsec ifs="fm1-gb3,fm1-gb4"
```

- Start the HostAP.

```
# Replace with the path to the configuration file
/usr/sbin/hostapd /etc/hostapd.conf &
```

- Start the wpa_supplicant.

```
# Replace with the path to the configuration file and the appropriate interface name
```

```
wpa_supplicant -Dwired -ifm1-gb3 -c/etc/wpa_supplicant.conf &
```

- Inspect MACsec traffic statistics with `ethtool`.

```
ethtool -S fm1-gb3
```

Limitations and known issues

- Only one SecY with one corresponding TX SC can be enabled on each interface.
- Once MACsec is enabled, all traffic passes through the SecY.
- MACsec is not compatible with ASF.
- MACsec statistics for RX path are not supported in `ethtool`.
- MACsec is not functional on ports connected to the internal L2switch on T1040 platforms. Consult your platform's specifications to determine which ports have MACsec support.

7.6.1.2.9 Changes from previous versions

- **SDK2.0**
  - *Debugging with ethtool*

    Removed the DPAA Ethernet counters from debugfs and exported them through ethtool statistics.
  - *CEETM*

    Integrated the CEETM qdisc with the DPAA Ethernet driver.
- **SDK1.8**
  - *MACsec*

    Added MACsec kernel driver and a set of User-space Linux APIs to allow developers to configure the MACsec hardware block.
  - *DPAA Ethernet as loadable modules*

    Added loadable module support for the DPAA Ethernet driver.
  - *Adding support for DPAA Ethernet in Topaz Hypervisor*

    This is a documentation section enumerating the steps to follow in order to enable DPAA Ethernet in Topaz Hypervisor.
- **SDK1.7**
  - *PCD configuration files*

    Included in the SDK package PCD configuration and policy files (.xml files) for each platform supported by the SDK. The configuration files include all the ports enabled by the RCWs associated with each platform.
- **SDK1.6**
  - *oNIC device driver - DPAA hardware offloading aware Ethernet net device*

    Added oNIC netdevice, based on OH FMAN ports in order to offer advanced DPAA offloading capabilities like: IPSec offload, zero-copy frames between USDPAA and kernel stack, OH CSUM offload.
  - *DPAA OH port driver update*

    Previous versions of the offline port driver did not initialize the frame queues that entered and exited the OH port and relied on other software components (Ethernet driver, USDPAA, or other kernel modules) to initialize those queues. This update adds full capability for initializing both ingress and egress queues, eliminating the dependency on USDPAA and/

or other kernel modules. With it, complex offload architectures that have OH ports can be largely initialized by the offline port driver.

- *Priority Flow Control (PFC) - 802.1Qbb*

  Experimental feature added on T4240/T2080 for traffic priority classes and MAC support for 802.1Qbb. PFC provides the ability to issue and respond to Pause Frames on a priority-flow basis and prevents a single flow from consuming the entire port's bandwidth. This is distinct from standard flow control which turns on or off the Ethernet port, stopping all flows.

- *Sleep/DeepSleep support and Wake on LAN (WoL) support*

  Implement the suspend/resume features that allow stopping the Ethernet port before system enters the sleep state and resuming the port to normal state when the system is waken up. Updated the Ethernet driver to work together with the Wakeup-on-LAN options of ethtool utility. More information about this can be found at Documentation/power/devices.txt in the kernel source tree.

- **SDK1.5**
  - *Single driver for termination and forwarding*

    There is only one Ethernet driver codebase now, based on the “optimized for termination” version which implements the support for S/G frames. But the driver has undergone an optimization process such that its “IP forwarding” performance is similar now to that of the removed “optimized for forwarding” driver. In addition, the “termination” performance has been improved.

    All source code related to the “forwarding” driver is removed, along with the following Kconfig options: FSL_DPAA_ETH_OPTIMIZE_FOR_IPFWD, FSL_DPAA_ETH_OPTIMIZE_FOR_TERM and CONFIG_FSL_DPAA_ETH_SG_SUPPORT. The “termination” driver support for S/G frames requires a different socket buffer layout. The **ASF** and **IEEE1588** modules has been adapted to the new buffer layout, so the existing feature set has been preserved.

  - *Buffer recycling algorithm updates*

    The most important part of the private driver performance improvement strategy consist in data buffer fragments recycling and skb structure recycling. These techniques, formerly deployed in “forwarding ” driver, has been adapted for the S/G support and the new skb layout.

  - *CPU hotplug support*

    In order to support CPU Hotplug, the DPAA Ethernet driver has been adapted to a new QMAN API and implemented a new NAPI logic which maps the portals to each available CPU. When a CPU goes offline the portal interrupts are seamlessly migrated and processed by the boot-CPU (CPU #0)..

  - *Control MAC multicast group using socket API on MAC-less netdevice.*

    The MAC-less interface is now able to configure a MAC device by using the Proxy DPAA Ethernet Driver . The proxy interface offers a simple API to MAC-less interface for enablement and disablement of the MAC device and for adding and removing mac unicast and multicast addresses. This MAC-less feature is optional and can be activated by setting "proxy" attribute in the device tree node of the MAC-less interfaces.

  - *ASF is the default SDK configuration*

    For specific use cases, ASF can provide superior performance than the upstream IP stack, bypassing the normal Ethernet driver and stack processing. Normal stack features may become unavailable in ASF configuration. But, if one needs to compile out the ASF support, the variable KERNEL_DELTA_DEFCONFIG must be set to empty.

- **SDK1.4**
  - Added pause frame control support through ethtool
  - Added netpoll support
  - Moved "QDisc bypass for performance reasons" option to ASF
  - Added Linux standard API for hardware timestamping (IEEE1588)

- Moved kernel config options for DPAA Ethernet driver from "Device Drivers -> Network device support -> Ethernet (10000 Mbit) -> NXP Data Path Frame Manager Ethernet" to "Device Drivers->Network device support->Ethernet driver support->NXP devices -> DPAA Ethernet "

- **SDK1.3**
  - The bootarg which controls the maximum frame size (previously "fsl_fman_phy_max_frm") has been renamed as "fsl_fm_max_frm" and is now available in the menuconfig via: Device Drivers --> Frame Manager support --> NXP Frame Manager (datapath) support --> Maximum L2 frame size (CONFIG_FSL_FM_MAX_FRAME_SIZE)
  - The bootarg which controls the extra headroom ("fsl_fm_rx_extra_headroom") has been moved in the Kconfig and is now available via: Device Drivers --> Frame Manager support --> NXP Frame Manager (datapath) support --> Add extra headroom at beginning of data buffers (CONFIG_FSL_FM_RX_EXTRA_HEADROOM)
  - Scatter/Gather support is activated by the driver's "Optimize for termination" compile-time choice. It is no longer explicitly accessible via the Kconfig.

7.6.1.2.9.1 Known Issues

- The MTU currently defaults to a maximum of 1522. If you want a higher MTU, it is necessary to pass fsl_fm_max_frm=N on the kernel bootargs, where "N" is the desired maximum MTU + 22.
- Scatter Gather frames and Jumbo frames are not supported on LS1043A due to the FMan A010022 errata. We recommend disabling S/G through ethtool.

```
ethtool -K <interface> tx-scatter-gather off
```

7.6.1.2.9.2 Good Questions

1. What channel are the FQs assigned to?

   A: Each interface uses by default one pool channel across all Software Portals and also the dedicated channels of each CPU. Note that any of these channels may be shared with other DPAA Eth net devices, and even with other DPAA drivers such as SEC. The *default* and *error* FQs are assigned to the pool channel. The TX queues are assigned to the (direct connect) channel linked to the TX port associated with the interface. Any other statically-defined queues will be assigned in a round-robin fashion to the core-affine portals.

2. What work queue are the FQs assigned to?

   A:

   - Tx Confirmation FQs go to WQ1
   - Rx Error and Tx Error FQs go to WQ2
   - Rx Default, Tx and PCD FQs go to WQ3

3. How do I use the core-affined queues?

   The anticipated way of using the core-affined queues is to use one of the default FMC policy files:

   ```
   /etc/fmc/config/private/common/policy_ipv4.xml
   ```

   ```
   /etc/fmc/config/private/common/policy_ipv6.xml
   ```

   Default FMC configuration files are provided for each reference board:

   ```
   /etc/fmc/config/private/<name of reference board>/<RCW directory>/<name of configuration file>
   ```

   Here are two examples showing FMC commands using the default configuration and policy files:

   ```
   (1)    fmc -c /etc/fmc/config/private/t2080rdb/RRFFXX_P_66_15/config.xml  -p /etc/fmc/config/
   private/t2080rdb/RRFFXX_P_66_15/policy_ipv4.xml -a
   ```

Note that `/etc/fmc/config/private/t2080rdb/RRFFXX_P_66_15/policy_ipv4.xml` is a soft link to `/etc/fmc/config/private/common/policy_ipv4.xml`.

```
(2)    fmc -c /etc/fmc/config/private/t4240rdb/SSFFPPH_27_55_1_9/config_20g.xml  -p /etc/fmc/config/private/common/policy_ipv4.xml  -a
```

Note that `/etc/fmc/config/private/t4240rdb/RRFFXX_P_66_15/policy_ipv4.xml` is a soft link to `/etc/fmc/config/private/common/policy_ipv4.xml`.

If you create a configuration file instead of using one of the default configuration files, be sure to use the appropriate policies found in the default policy files:

`/etc/fmc/config/private/common/policy_ipv4.xml`

`/etc/fmc/config/private/common/policy_ipv6.xml`.

7.6.1.3 Quality of Service

7.6.1.3.1 Features

DPAA platforms can offload QoS functions such as policing, shaping, scheduling and prioritization to dedicated hardware blocks.

Traffic policing is achieved on ingress through the FMan. A two rate three color marker algorithm can be configured through the fmc tool.

Traffic scheduling, shaping, and prioritization is executed on the egress path in the QMan. Multiple algorithms, such as dual rate shaping and strict prioritization, are implemented and can be configured through queuing disciplines.

7.6.1.3.2 Policing

The FMan's Policer sub block implements a two rate, three color marker (trTCM) traffic policing algorithm. The algorithm has two configurable flavors: RFC2698 and RFC4115.

The fmc tool, described in detail in Frame Manager Configuration Tool User's Guide, is used to enable the Policer and set up its parameters.

For more information regarding the FMan Policer and how it can be configured, see the FMan Policer and the Policer Section chapters.

7.6.1.3.3 Scheduling and Shaping

7.6.1.3.3.1 Description

Specific DPAA 1.x platforms offer scheduling, shaping and prioritization capabilities through CEETM (Customer Edge Egress Traffic Management). The CEETM hardware block is a member of the QMan on the NXP QorIQ T-series and B-series platforms. Its purpose is to enhance the performances of DPAA platforms by moving the egress QoS logic from software to hardware.

This chapter briefly describes the CEETM block and its capabilities. Furthermore, it presents how it can be configured through the Linux traffic control tool (`tc`) by using a custom queuing discipline.

7.6.1.3.3.1.1 The CEETM architecture

CEETM is a sub block of the QMan and is an alternative to the regular *frame queue - work queue - channel* scheduling mode. For more information regarding this workflow, or on DCPs and sub-portals, please refer to the QMan Overview chapter.

A CEETM block, pictured in Figure 163. CEETM block on page 1068, is available for each FMan and it is intended to be used by FMan sub-portals linked to Ethernet interfaces.



Figure 163. CEETM block

CEETM uses 8 Logical Network Interfaces (LNIs) that can be mapped to the FMan's DCP sub-portals. Depending on the platform used, there are 8 or 32 class queue channels (or CQ channels) that can be mapped to the LNIs. Multiple CQ channels can be mapped to the same LNI.

Each CQ channel contains 16 class queues. 8 CQs are independent while the other 8 can be grouped into 1 class group or 2 class groups of 4 queues each. The first group is called *group A* and the second is called *group B*.

7.6.1.3.3.1.2 Features

CEETM implements the following algorithms:

- Strict Priority scheduling
- Weighted Bandwidth Fair Scheduling (WBFS)
- dual-rate shaping with committed and excess rates (CR/ER)
- shaped and unshaped Fair Queueing scheduling (shFQ, uFQ)

These algorithms are used together in specific combinations based on the CEETM's architecture described previously and pictured in Figure 164. CEETM architecture on page 1068 .



Figure 164. CEETM architecture

All the CQs connected to a CQ channel pass through a Strict Priority scheduler. The lower the CQ's ID, the higher the CQ's priority (e.g. CQ#3 has a higher priority than CQ#4, thus, as long as there are frames queued to CQ#3, CQ#4 will not be dequeued).

The priority of the CQ groups is configurable. All frames coming from the grouped CQs pass through the WBFS algorithm. Each CQ belonging to a group is assigned a weight portion of the bandwidth available to the group. The weight is a value from 1 to 248 in pseudo logarithmic steps of 1.5%. A list of available weights can be found in the platform's QorIQ DPAA Reference Manual.

The CQ channels can be shaped or unshaped. For CQs leading to a shaped channel, all frames will pass through a dual-rate shaper before entering the LNI. The independent CQs, as well as the class groups, can be configured to lead their frames through the CR shaper, the ER shaper, or both.

Each LNI aggregates frames from the CQ channels linked to it. All the unshaped frames from the unshaped CQ channels mapped to the LNI pass through the uFQ algorithm. The CR/ER frames from the shaped CQ channels pass through the shFQ algorithm and through another dual-rate shaper. Lastly, all frames pass through the LNI's Strict Priority module that schedules the unshaped frame (with high priority), the CR frames (with medium priority) and the ER frames (with low priority).

The shFQ algorithm schedules a channel for transmitting if the channel's shaper is time eligible (the shaper has a positive number of tokens in its bucket). When a channel finished its tokens, it is added to a waiting queue where it must wait for any other time eligible channels ahead of it finish transmitting.

The uFQ algorithm is similar to the shFQ. In the uFQ algorithm, all channels are time eligible. After finishing to transmit all their available data, they are added to the back of the time eligible waiting queue where their bucket is instantly refilled. The token bucket limit of the unshaped channels is configurable.

For more information regarding the CEETM's capabilities and detailed descriptions of the mentioned algorithms, take a look at your platform's QorIQ DPAA Reference Manual.

7.6.1.3.3.1.3 Integration with queuing disciplines

The CEETM block can be configured through the *ceetm* queuing discipline. A comparison between the hardware block and the traffic control's terminology is drawn in Figure 165. Comparison between CEETM and tc terminology on page 1069.



Figure 165. Comparison between CEETM and `tc` terminology

A LNI can be mapped to a FMan port by adding a *root* `ceetm` qdisc to a network interface. The LNI shaper's CR and ER are configured by setting a *rate*, and optional *ceil* and *overhead*, on the qdisc.

A CQ channel can be linked to a LNI by creating a `ceetm` *root* class mapped to the *root* qdisc. For an unshaped channel, the uFQ's token bucket limit (*tbl*) needs to be configured. For a shaped channel, the *rate*, and optional *ceil*, set the CR and ER.

Note: Shaped CQ channels can be linked to the LNI only if the LNI's shaper is enabled.

A channel's independent CQs are configured when a *prio* qdisc is linked to a *root* class. Between 1 and 8 *prio* classes are generated, each class corresponding to a CQ linked to the channel's Strict Priority scheduler. The *qcount* parameter indicates the number of child classes. If the channel is shaped, all generated classes participate by default in both CR and ER shaping. In order to disable one or the other, the CQ's corresponding *prio* class's *cr* and *er* parameters can be changed.

Note: CQs linked to a shaped CQ channel can not have both CR and ER shaping disabled.

In order to configure the CQ groups, a *wbfs* qdisc is linked to one of the *prio* classes. Either 4 or 8 *wbfs* classes are generated, depending on the number of CQs in the group indicated by the *qcount* parameter. The group is placed right after its parent in the channel's Strict Priority list (e.g. if the *wbfs* qdisc is linked to the *prio* class #2, the priority list becomes: class #1, class #2, group, class #3, class #4, etc). The CQ weights are configured through the *qweight* parameter and can be changed for each CQ individually. For groups linked to shaped CQ channels, the CR and ER shaping are enabled by the *cr* and *er* parameters.

Note: Groups linked to a shaped CQ channel can not have both CR and ER shaping disabled.

For more details on the `ceetm` qdisc's parameters and configuration, see the Usage on page 1071 chapter.

7.6.1.3.3.2 User guide

7.6.1.3.3.2.1 Supported platforms

The CEETM block is present and configurable through the `ceetm` qdisc on all T-series, B-series and LS1043A/LS1046A platforms.

7.6.1.3.3.2.2 Getting started

1. Enable the *ceetm* qdisc support in the kernel, along with any classifiers or other features that might be needed.

```
-> Device Drivers
    -> Network device support (NETDEVICES [=y])
        -> Ethernet driver support (ETHERNET [=y])
            -> Freescale devices (NET_VENDOR_FREESCALE [=y])
                -> DPAA Ethernet (FSL_SDK_DPAA_ETH [=y])
                    -> DPAA CEETM QoS (FSL_DPAA_CEETM [=y])

->  Networking support (NET [=y])
    -> Networking options
        -> QoS and/or fair queueing (NET_SCHED [=y])
            -> Universal 32bit comparisons w/ hashing (u32) (NET_CLS_U32 [=y])
```

2. Add the `ceetm` recipe to your Yocto build by adding the following line to your `conf/local.conf` file.

```
IMAGE_INSTALL_append =  "  ceetm"
```

7.6.1.3.3.2.3 Limitations

- CEETM is supported on DPAA Private Ethernet interfaces only.
- CEETM isn't supported on top of Linux bonding interfaces.

7.6.1.3.3.2.4 Usage

You can see the `ceetm` qdisc's help message by running the following command:

```
~# tc qdisc add ceetm help
Usage:
... qdisc add ... ceetm type root [rate R [ceil C] [overhead O]]
... class add ... ceetm type root (tbl T | rate R [ceil C])
... qdisc add ... ceetm type prio qcount Q
... qdisc add ... ceetm type wbfs qcount Q qweight W1 ... Wn [cr CR] [er ER]

Update configurations:
... qdisc change ... ceetm type root [rate R [ceil C] [overhead O]]
... class change ... ceetm type root (tbl T | rate R [ceil C])
... class change ... ceetm type prio [cr CR] [er ER]
... qdisc change ... ceetm type wbfs [cr CR] [er ER]
... class change ... ceetm type wbfs qweight W

Qdisc types:
root - configure a LNI linked to a FMan port
prio - configure a channel's Priority Scheduler with up to eight classes
wbfs - configure a Weighted Bandwidth Fair Scheduler with four or eight classes

Class types:
root - configure a shaped or unshaped channel
prio - configure an independent class queue

Options:
R - the CR of the LNI's or channel's dual-rate shaper (required for shaping scenarios)
C - the ER of the LNI's or channel's dual-rate shaper (optional for shaping scenarios, defaults
to 0)
O - per-packet size overhead used in rate computations (required for shaping scenarios,
recommended value is 24 i.e. 12 bytes IFG + 8 bytes Preamble + 4 bytes FCS)
T - the token bucket limit of an unshaped channel used as fair queuing weight (required for
unshaped channels)
CR/ER - boolean marking if the class group or prio class queue contributes to CR/ER shaping (1)
or not (0) (optional, at least one needs to be enabled for shaping scenarios, both default to 1
for prio class queues)
Q - the number of class queues connected to the channel (from 1 to 8) or in a class group (either
4 or 8)
W - the weights of each class in the class group measured in a log scale with values from 1 to
248 (when adding a wbfs qdisc, either four or eight, depending on the size of the class group;
when updating a wbfs class, only one)
```

Filters need to be added on each qdisc layer in order to allow packets to reach the leaf classes. Likewise, all filters need to be removed from each qdisc layer when no longer used.

7.6.1.3.3.3 Examples

7.6.1.3.3.3.1 Rate limit two streams

7.6.1.3.3.3.1.1 Setup

In the following example a platform with CEETM support (T1040RDB - Client) is connected to another board (T1024RDB - Server) through a 1G link. The described setup is pictured in Figure 166. Rate example setup on page 1072.



Figure 166. Rate example setup

The `iperf` clients run on the Client while the `iperf` servers run on the Server. The Server listens on 2 TCP ports (21 and 80).

```
root@t1024rdb:~# iperf -s -p 21 &
root@t1024rdb:~# iperf -s -p 80 &
```

PCDs are applied on both platforms in advance.

```
root@t1024rdb:~# fmc -c /etc/fmc/config/private/t1024rdb/RRX_PPP_95/config.xml -p /etc/fmc/config/
private/t1024rdb/RRX_PPP_95/policy_ipv4.xml -a
root@t1040rdb:~# fmc -c /etc/fmc/config/private/t1040rdb/RR_P_66/config.xml -p /etc/fmc/config/
private/t1040rdb/RR_P_66/policy_ipv4.xml -a
```

In order to keep this example minimal, ARP frames aren't filtered and classified. Thus, MAC addresses need to be exchanged and saved in advance as well.

```
root@t1040rdb:~# arp -s <server IP address> <server HW address>
root@t1024rdb:~# arp -s <client IP address> <client HW address>
```

After adding the qdiscs, the Client runs the `iperf` clients.

```
root@t1040rdb:~# iperf -c <server IP address> -p 21 &
root@t1040rdb:~# iperf -c <server IP address> -p 80 &
```

7.6.1.3.3.3.1.2 Execution

This example's corresponding qdisc and class hierarchy is pictured in Figure 167. Rate example class hierarchy on page 1072.



Figure 167. Rate example class hierarchy

Add a ceetm qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~#  tc qdisc add dev fm1-gb0 root handle 1: ceetm type root rate 1000mbit overhead
24
```

Add a shaped channel to the LNI and configure its dual-rate shaper with a CR of 150mbps.

```
root@t1040rdb:~#  tc class add dev fm1-gb0 parent 1: classid 1:1 ceetm type root rate 150mbit
```

Add another shaped channel to the LNI and configure its dual-rate shaper with a CR of 850mbps.

```
root@t1040rdb:~#  tc class add dev fm1-gb0 parent 1: classid 1:2 ceetm type root rate 850mbit
```

Configure one of the first channel's priority classes (marked by default as both CR and ER eligible).

```
root@t1040rdb:~#  tc qdisc add dev fm1-gb0 parent 1:1 handle 2: ceetm type prio qcount 1
```

Configure one of the second channel's priority classes (marked by default as both CR and ER eligible).

```
root@t1040rdb:~#  tc qdisc add dev fm1-gb0 parent 1:2 handle 3: ceetm type prio qcount 1
```

Add filters that will classify all packets with the destination port equal to 80 and lead them through the priority class of the first channel.

```
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 1:1
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 2:1
```

Add filters that will classify all packets with the destination port equal to 21 and lead them through the priority class of the second channel.

```
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 1:2
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 3: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 3:1
```

7.6.1.3.3.3.2 Prioritization of two streams

7.6.1.3.3.3.2.1 Setup

The same setup is used as for the rate limit example.

7.6.1.3.3.3.2.2 Execution

This example's corresponding qdisc and class hierarchy is pictured in Figure 168. Prioritization example class hierarchy on page 1073.



Figure 168. Prioritization example class hierarchy

Add a ceetm qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~#  tc qdisc add dev fm1-gb0 root handle 1: ceetm type root rate 1000mbit overhead
24
```

Add a shaped channel to the LNI and configure its dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~#  tc class add dev fm1-gb0 parent 1: classid 1:1 ceetm type root rate 1000mbit
```

Configure two of the channel's priority classes (marked by default as both CR and ER eligible).

```
root@t1040rdb:~#  tc qdisc add dev fm1-gb0 parent 1:1 handle 2: ceetm type prio qcount 2
```

Add filters that will classify all packets with the destination port equal to 80 and lead them through the highest priority class of the channel.

```
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 1:1
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 2:1
```

Add filters that will classify all packets with the destination port equal to 21 and lead them through the second (lowest) priority class of the channel.

```
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 8000
0xffff flowid 1:1
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 8000
0xffff flowid 2:2
```

7.6.1.3.3.3.3 Assigning weights to two streams

7.6.1.3.3.3.3.1 Setup

The same setup is used as for the rate limit example.

7.6.1.3.3.3.3.2 Execution

This example's corresponding qdisc and class hierarchy is pictured in Figure 169. WBFS example class hierarchy on page 1074.



Figure 169. WBFS example class hierarchy

Add a ceetm qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~#  tc qdisc add dev fm1-gb0 root handle 1: ceetm type root rate 1000mbit overhead
24
```

Add a shaped channel to the LNI and configure its dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~#  tc class add dev fm1-gb0 parent 1: classid 1:1 ceetm type root rate 1000mbit
```

Configure one of the channel's priority classes (marked by default as both CR and ER eligible).

```
root@t1040rdb:~#  tc qdisc add dev fm1-gb0 parent 1:1 handle 2: ceetm type prio qcount 1
```

Configure a class group of four classes, place it after the 2:1 class in the priority list, and assign different weights to each class (10, 50, 120 and 200).

```
root@t1040rdb:~#  tc qdisc add dev fm1-gb0 parent 2:1 handle 3: ceetm type wbfs qcount 4 qweight
10 50 120 200 cr 1 er 1
```

Add filters that will classify all packets with the destination port equal to 21 and lead them through the class with the highest weight of the group.

```
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 1:1
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 2:1
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 3: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 3:1
```

Add filters that will classify all packets with the destination port equal to 80 and lead them through another classes of the group.

```
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 1:1
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 2:1
root@t1040rdb:~#  tc filter add dev fm1-gb0 parent 3: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 3:3
```

7.6.1.3.3.3.4 Unshaped Fair Queuing of two streams

7.6.1.3.3.3.4.1 Setup

In the following example a platform with CEETM support (T1024RDB - Main) is connected to two other boards: a T1024RDB (Client) through a 10G link and a T1040RDB (Server) through a 1G link. The described setup is pictured in Figure 170. Unshaped Fair Queuing example setup on page 1075.

Figure 170. Unshaped Fair Queuing example setup

The `iperf` clients run on the Client while the `iperf` servers run on the Server. The Server listens on two TCP ports (80 and 81).

```
root@t1040rdb:~# iperf -s -p 80 &
root@t1040rdb:~# iperf -s -p 81 &
```

PCDs are applied on all platforms in advance.

```
root@t1024rdb:~# fmc -c /etc/fmc/config/private/t1024rdb/RRX_PPP_95/config.xml -p /etc/fmc/config/
private/t1024rdb/RRX_PPP_95/policy_ipv4.xml -a
root@t1040rdb:~# fmc -c /etc/fmc/config/private/t1040rdb/RR_P_66/config.xml -p /etc/fmc/config/
private/t1040rdb/RR_P_66/policy_ipv4.xml -a
```

In order to keep this example minimal, ARP frames aren't filtered and classified. Thus, MAC addresses need to be exchanged and saved in advance as well.

```
# Server:
root@t1040rdb:~# arp -s <main IP address> <main HW address>
# Main:
root@t1024rdb:~# arp -s <client IP address> <client HW address>
root@t1024rdb:~# arp -s <server IP address> <server HW address>
# Client:
root@t1024rdb:~# arp -s <main IP address> <main HW address>
```

IP forwarding is enabled on the Main board. Routes are added on the Server and Client boards as well.

```
# Main:
root@t1024rdb:~# echo 1 > /proc/sys/net/ipv4/ip_forward
# Client:
root@t1024rdb:~# route add -net <server network address> <server network mask> gw <main IP
address>
# Server:
root@t1040rdb:~# route add -net <client network address> <client network mask> gw <main IP
address>
```

After adding the qdiscs, the Client runs the `iperf` clients.

```
root@t1024rdb:~# iperf -c <server IP address> -p 80 &
root@t1024rdb:~# iperf -c <server IP address> -p 81 &
```

7.6.1.3.3.3.4.2 Execution

This example's corresponding qdisc and class hierarchy is pictured in Figure 171. Unshaped Fair Queuing example class hierarchy on page 1076.



Figure 171. Unshaped Fair Queuing example class hierarchy

Add a ceetm qdisc to the interface and don't configure the LNI's dual-rate shaper.

```
root@t1024rdb:~# tc qdisc add dev fm1-mac3 root handle 1: ceetm type root
```

Add an unshaped channel to the LNI and configure its CR's token bucket limit to 1000 bytes.

```
root@t1024rdb:~# tc class add dev fm1-mac3 parent 1: classid 1:1 ceetm type root tbl 1000
```

Add another unshaped channel to the LNI and configure its CR's token bucket limit to 500 bytes.

```
root@t1024rdb:~# tc class add dev fm1-mac3 parent 1: classid 1:2 ceetm type root tbl 500
```

Configure one of the first channel's priority classes.

```
root@t1024rdb:~# tc qdisc add dev fm1-mac3 parent 1:1 handle 2: ceetm type prio qcount 1
```

Configure one of the second channel's priority classes.

```
root@t1024rdb:~# tc qdisc add dev fm1-mac3 parent 1:2 handle 3: ceetm type prio qcount 1
```

Add filters that will classify all packets with the destination port equal to 80 and lead them through the priority class of the first channel.

```
root@t1024rdb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 1:1
root@t1024rdb:~#  tc filter add dev fm1-mac3 parent 2: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 2:1
```

Add filters that will classify all packets with the destination port equal to 81 and lead them through the priority class of the second channel.

```
root@t1024rdb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 81
0xffff flowid 1:2
root@t1024rdb:~#  tc filter add dev fm1-mac3 parent 3: prio 1 protocol ip u32 match ip dport 81
0xffff flowid 3:1
```

7.6.2 Linux Ethernet for Enhanced Three Speed Ethernet Controller Family

7.6.2.1 Linux Ethernet Driver for eTSEC

7.6.2.1.1 Introduction

7.6.2.1.1.1 Overview

Gianfar is the Linux driver that enables Ethernet support for the SoCs featuring eTSEC (Enhanced Three-Speed Ethernet Controllers). Though the driver is designed to support the latest eTSEC2.0 features present on the low-power QorIQ platforms, it also maintains backward compatibility with older IPs from the same family, like eTSEC (eTSEC 1.x) and TSEC (present on the PowerQUICC III platforms) and FEC (Fast Ethernet Controller).

7.6.2.1.1.2 Purpose

The purpose of this document is to provide a user guide and configuration details for the Gianfar driver, and a high-level view of the driver's structure, as well as to describe its major functionalities with a focus on the features provided by the eTSEC2.0 IP.

7.6.2.1.1.3 Features

This section provides an overview of the major eTSEC2.0 (“virtualized” eTSEC) features:

- o MAC Layer
- *o Interrupt grouping mechanism*
- *o Virtualized register space*
- o Rx Subsystem:
  - MAC Address Filter
  - L2/L3/L4 Parser
  - Filer Engine
  - *Hash or RR Distribution*
  - *Multiple Rx Interrupt*
- o Tx Subsystem:
  - Tx Scheduler
  - L3/L4 Offload
  - *Multiple Tx Interrupt*



Figure 172. “Virtualized” eTSEC Block Diagram

- *o Interrupt virtualization:*
  - Each ring maps to one of two separate groups for interrupt and BD management; each group associated by software with a CPU.
  - Separate address spaces per group and for MDIO.

- Interrupt coalescing controls per ring in multi-group mode, packet-count based and timer based thresholds, for both Rx and Tx.

- *o TCP/IP Offload Engine (TOE):*
  - IP v4 and IP v6 header recognition on receive
  - IP v4 header checksum verification and generation
  - TCP and UDP checksum verification and generation
  - Per-packet configurable offload
  - Recognition of VLAN, stacked-VLAN, 802.2, PPPoE session, MPLS stacks, and ESP/AH IP-Security headers
- *o Quality of service (QoS) support:*
  - Transmission from up to eight queues: priority-based queue selection or modified weighted round-robin (MWRR) queue selection with fair bandwidth allocation
  - Reception to up to eight physical queues:

    - Table-oriented queue filing strategy based on 16 header fields or flags

    - Frame rejection support for filtering applications

    - Filing based on Ethernet, IP, and TCP/UDP properties, including VLAN fields, Ether-type, IP protocol type, IP TOS or differentiated services, IP source and destination addresses, TCP/UDP port number

7.6.2.1.1.4 Notes on high level decomposition and data flow

A system level block view, from a network device perspective, may be depicted as follows:



Figure 173. Gianfar High level decomposition

The eTSEC2.0 and PHY are the hardware blocks, the kernel networking stack along with the network driver are running in the Kernel space, and finally ethtool and iproute2 are examples of user space tools used for configuring the network devices.

The eTSEC2.0 includes some additional support compared with the previous versions:

- it has support for interrupt virtualization
- on the TX side, it can distribute packets to the multiple queues based on simple hashing or round robin mechanisms

The eTSEC2.0 has support for multiple RX and TX queues. On the receive side, an incoming packet will be filed to one of the queues based on the rules programmed into the filer. By default, all the packets will be filed to queue 0. On the transmit side, either a simple hash based implementation or a round robin algorithm distributes the packets to the available number of queues. User space packages like ethtool and iproute are used to configure the network device parameters. The ethtool interface is extended to provide support for filer programming.

The kernel space module for the network driver is the most important block as it communicates with both the user space and the H/W IP to control the processing of packets. The eTSEC network device driver will be referred to as Gianfar in the rest of the document.

The Gianfar driver may be divided into sub-blocks based on the number of independent threads that Linux will run in order to completely transfer a packet from ingress to egress side. The basic functionality of any Ethernet driver is to handle the reception of packets from an ingress port (might include checksum calculation, header verification, etc), as well as the transmission of packets on the egress port (might include checksum re-calculation, header manipulation, etc). There are also the device configuration and control functionalities, and device status reporting. When the Ethernet driver is actually implementing these functionalities, it needs to interact with the core (Kernel) as well as the hardware IP (the Ethernet controller).



Figure 174. Gianfar Packet Flow

In the above Figure it can be noted that the receive side includes parsing/ filing before a packet is "put" into a buffer descriptor. The transmit side includes a H/W queue scheduler for transmission of packets.

As already mentioned, eTSEC2.0 has support for multiple hardware queues for Rx and Tx in hardware. These queues are basically divided into two groups; let's say all odd numbered queues correspond to one group and even numbered queues

correspond to other group. In a multi core environment (e.g. a dual core system), each group of queues can be programmed to be handled by one of the two cores, which will result in an increased performance.

For simplicity, we always assume that:

- All the even numbered queues are mapped to Group 0 and odd numbered queues are mapped to Group1.
- Group 0 interrupts can be assigned to be processed by Core 0 and Group 1 interrupts to be processed by Core 1 (except for error the interrupts, which are always destined to Core 0, which is the master core).

From the above figure, it can be noticed that there will be a receive, and a transmit thread running on each core, for processing the packets corresponding to the group assigned to that core. The receive thread processes the received packets - handles the RX buffer descriptor (BD) rings, and passes the received packets to the networking stack for further processing; the transmit thread schedules the packets passed down by the stack to be transmitted out of the device. There's also a transmission cleanup thread, triggered by the TX confirmation interrupts, to handle the TX BD rings and congestion.

Gianfar may be broadly decomposed into the following sub-blocks:

1. Initialization block
2. Receive block
3. Transmit block
4. Control block

So, the Receive and Transmit blocks handle processing of ingress and egress packets.

Before processing packets the driver needs to perform some initialization steps like:

- extracting the device tree parameters
- initialization of the multiple queues
- registering the driver with the kernel
- allocating buffer descriptors
- registering the interrupts (etc.)

All these functionalities are implemented by the Initialization block.

Each of these sub-modules implements various functionalities, as detailed in the coming section.

7.6.2.1.2 Functionality

7.6.2.1.2.1 Multi-Queue support

eTSEC features multiple physical queues or BD rings. The multi-queue support (MQ) in the driver is enabled by default for eTSEC2.0 IPs.

Hardware queue events are mapped to one of the two available CPUs via eTSEC *Interrupt Groups*. For eTSEC2.0, each Rx/Tx hardware queue or BD ring is mapped to one of the two available Interrupt Groups, and each group in turn has its Rx/Tx interrupt lines assigned to a given CPU. By default, the driver enables 1 Rx and 1 Tx queue per Interrupt Group.

eTSEC2.0 supports 2 Interrupt Groups, this is also known as the Multi-Group (MG) mode in Gianfar. Each group has its own Rx, Tx and Err interrupt lines which can be individually affined to any of the 2 CPUs, as a measure to balance the processing load. Also, each interrupt group has its own block of registers, most notably `ievent`, `imask`, `tstat`, and `rstat`, so queue events are handled at the interrupt group level. Having more than 1 Rx and 1 Tx queue assigned to a single interrupt group would thus incur a software processing overhead that would not be justifiable for the majority of use cases. This is why the driver enables by default only 1 set of Rx and Tx queues per Interrupt Group.

eTSEC1.x and other older eTSEC IPs support only one interrupt group (g0), meaning that they are working in Single Group (SG) mode.

The mapping Rx/Tx queues to interrupt groups is by default: Rx Q0 and Tx Q0 assigned to Group0 (`g0`), and Rx Q1 and Tx Q1 assigned to Group1 (`g1`).



Figure 175. Multi-Queue Multi-Group

NOTE

Supporting more than one Rx/Tx queue per interrupt group has been obsoleted (see above). As a result, the following device tree properties are *obsolete*: `fsl,num-rx-queues`, `fsl,num-tx-queues`, `fsl,rx-bit-map`, and `fsl,tx-bit-map`.

7.6.2.1.2.2 Receive Side Scaling support

eTSEC supports multiple Rx and Tx descriptor queues (see multi-queue support). On reception, eTSEC can send different packets to different queues to distribute processing among CPUs. This mechanism is generally known as “Receive-side Scaling” (RSS).

In Gianfar, packets are distributed by applying "n-tuple" filters configured from `ethtool -N` (`--config-ntuple` option). These filters are converted by Gianfar to eTSEC Filer H/W rules. Based on these programmable filters, each packet is assigned to one of a small number of logical flows. Packets for each flow are steered to separate receive queues, which in turn can be processed by separate CPUs.



Figure 176. eTSEC RSS support

In Gianfar, Rx flows may be classified either by hashing various protocol header fields, see `ethtool -N rx-flow-hash` option, or by specifying flow type classification rules, see `ethtool -N flow-type` option. Refer to ethtool Linux man-pages for `ethtool -N` option details. A simple usage example is shown below.

```
root@ls1021aqds:~# ethtool -N eth0 flow-type udp4 src-ip 172.16.1.4 dst-port 5000 action 0
fsl-gianfar ethernet.4 eth0: Receive Queue Filtering enabled
Added rule with ID 254
root@ls1021aqds:~# ethtool -N eth0 flow-type udp4 src-ip 172.16.1.4 dst-port 5001 action 1
Added rule with ID 253
root@ls1021aqds:~# ethtool -n eth0
2 RX rings available
Total 2 rules

Filter: 253
        Rule Type: UDP over IPv4
        Src IP addr: 172.16.1.4 mask: 0.0.0.0
        Dest IP addr: 0.0.0.0 mask: 255.255.255.255
        TOS: 0x0 mask: 0xff
        Src port: 0 mask: 0xffff
        Dest port: 5001 mask: 0x0
        Action: Direct to queue 1
```

```
Filter: 254
        Rule Type: UDP over IPv4
        Src IP addr: 172.16.1.4 mask: 0.0.0.0
        Dest IP addr: 0.0.0.0 mask: 255.255.255.255
        TOS: 0x0 mask: 0xff
        Src port: 0 mask: 0xffff
        Dest port: 5000 mask: 0x0
        Action: Direct to queue 0

root@ls1021aqds:~# iperf -s -u -p 5000 &
[1] 1017
------------------------------------------------------------
Server listening on UDP port 5000
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
root@ls1021aqds:~# iperf -s -u -p 5001 &
[2] 1020
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:          7          0       GIC 176  eth0_g0_tx
177:          6          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          0          0       GIC 179  eth0_g1_tx
180:          0          0       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er

[  3] local 172.16.1.100 port 5000 connected with 172.16.1.4 port 52163
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.003 ms    0/  893 (0%)

root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:          9          0       GIC 176  eth0_g0_tx
177:        902          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          1          0       GIC 179  eth0_g1_tx
180:          0          0       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er

[  3] local 172.16.1.100 port 5001 connected with 172.16.1.4 port 46257
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.004 ms    0/  893 (0%)

root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:         10          0       GIC 176  eth0_g0_tx
177:        902          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          1          0       GIC 179  eth0_g1_tx
180:        894          0       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er
root@ls1021aqds:~# echo 1 > /proc/irq/177/smp_affinity
root@ls1021aqds:~# echo 2 > /proc/irq/180/smp_affinity
root@ls1021aqds:~# iperf -s -u -p 1000 &
[3] 1031
------------------------------------------------------------
```

```
Server listening on UDP port 1000
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
[  3] local 172.16.1.100 port 1000 connected with 172.16.1.4 port 58669
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.002 ms    0/  893 (0%)

root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:         13          0       GIC 176  eth0_g0_tx
177:       1798          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          1          0       GIC 179  eth0_g1_tx
180:        894          0       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er

[  4] local 172.16.1.100 port 5001 connected with 172.16.1.4 port 58876
[  4]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.004 ms    0/  893 (0%)

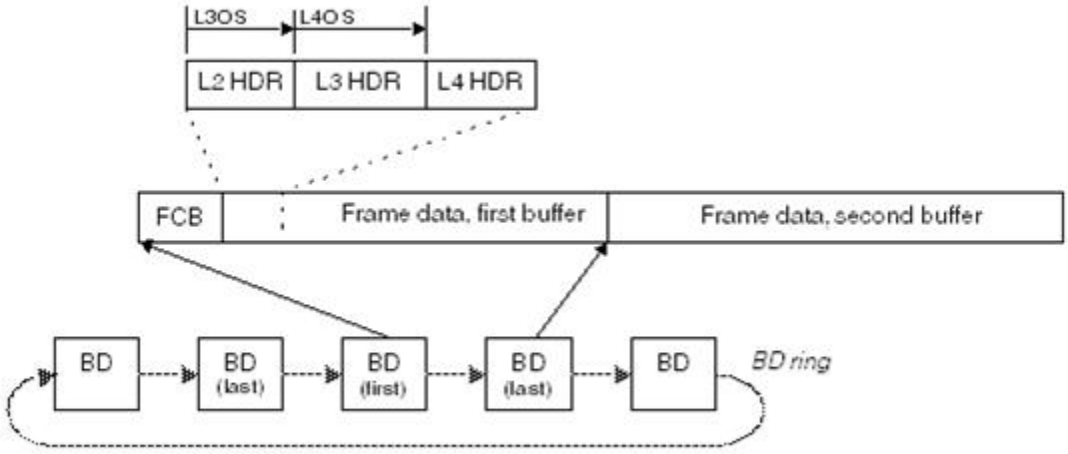
root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:         16          0       GIC 176  eth0_g0_tx
177:       1800          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          1          0       GIC 179  eth0_g1_tx
180:        894        894       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er

root@ls1021aqds:~# ethtool -n eth0
2 RX rings available
Total 2 rules

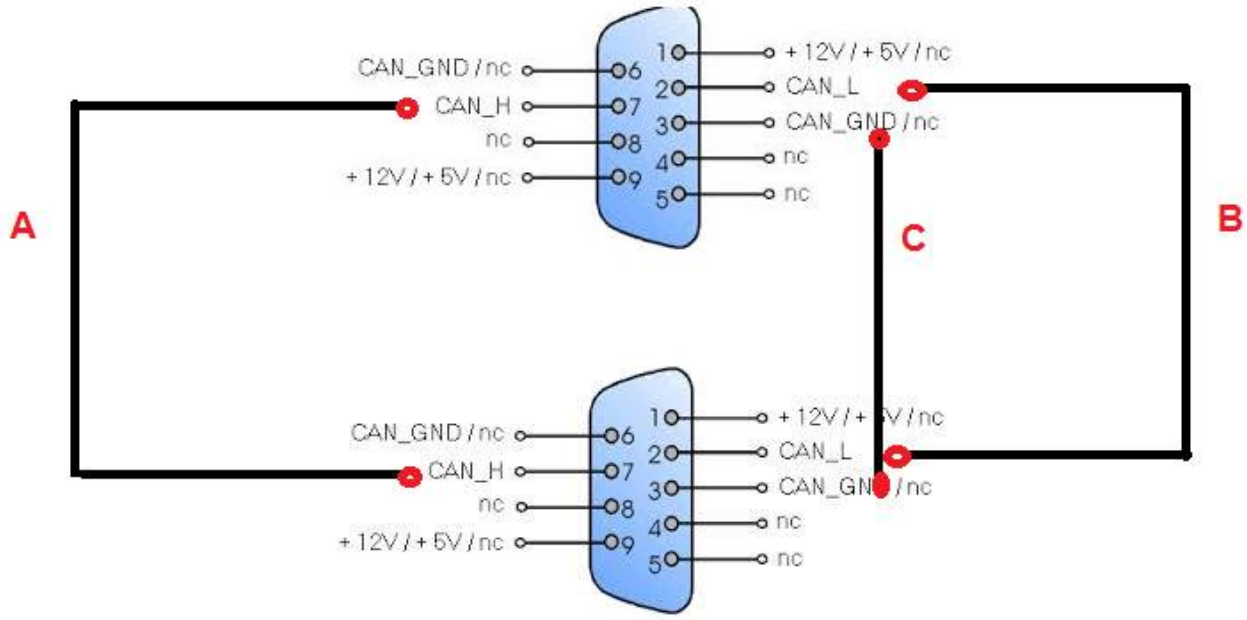
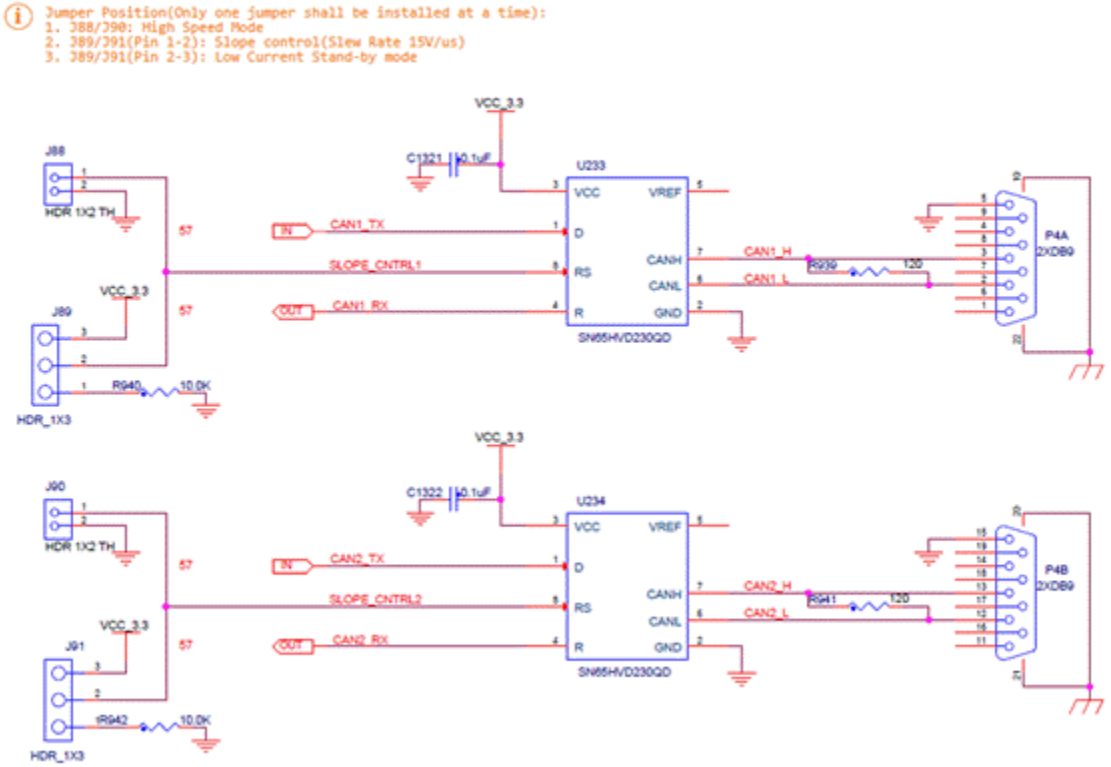
Filter: 253
        Rule Type: UDP over IPv4
        Src IP addr: 172.16.1.4 mask: 0.0.0.0
        Dest IP addr: 0.0.0.0 mask: 255.255.255.255
        TOS: 0x0 mask: 0xff
        Src port: 0 mask: 0xffff
        Dest port: 5001 mask: 0x0
        Action: Direct to queue 1

Filter: 254
        Rule Type: UDP over IPv4
        Src IP addr: 172.16.1.4 mask: 0.0.0.0
        Dest IP addr: 0.0.0.0 mask: 255.255.255.255
        TOS: 0x0 mask: 0xff
        Src port: 0 mask: 0xffff
        Dest port: 5000 mask: 0x0
        Action: Direct to queue 0

root@ls1021aqds:~# ethtool -N eth0 delete 254
root@ls1021aqds:~# ethtool -N eth0 delete 253
root@ls1021aqds:~# ethtool -n eth0
2 RX rings available
Total 0 rules

root@ls1021aqds:~#
```

7.6.2.1.2.3 NAPI support

Gianfar uses NAPI handling on both Rx and Tx paths, the Linux polling mechanism being triggered by frame receive interrupts and, respectively, frame transmit confirmation interrupts. The driver registers irqs for both Rx and Tx, and the NAPI (polling) handlers are provided to the Kernel.

On the receive path:

- When the receive interrupt gets triggered on a given CPU, a softirq for the polling function on Rx is scheduled.
- The RX_SOFTIRQ thread is raised by the Kernel, on the CPU on which it was triggered (and scheduled), and the Rx queues mapped to the corresponding interrupt group will be processed by the driver's polling function and the incoming packets are being passed to the networking stack.

Similarly on the transmit part:

- A frame transmit confirmation interrupt triggers the scheduling of a softirq under whose context the driver's polling routine for cleaning the Tx rings is invoked.
- The Tx polling routine is also associated with a given interrupt group and it will handle only the transmit queues that are affiliated to that interrupt group.

For packet forwarding, for instance, by mapping the per flow Rx and Tx queues to interrupt groups that are associated to the same CPU, makes it possible to maintain per CPU buffer pools used for reclaiming buffers on a per flow basis, improving cache locality at the same time.

7.6.2.1.2.4 Interrupt Coalescing

On a high speed network interface the rate of packet reception and transmission can be as high as the CPUs would be spending most of the time servicing these interrupts. With the interrupt coalescing feature, packets are collected and one single interrupt is generated for multiple packets to avoid flooding the system with interrupts from the Ethernet device.

eTSEC supports hardware coalescing of interrupts for both receive and transmit, using packet-count-based thresholds, complemented by timer-based thresholds. Gianfar provides basic support for setting the coalescing parameters via `ethtool -C`, for each device instance, by implementing the following "set coalesce" options:

Table 193. `ethtool -C` options:

rx-frames	packet count threshold for receive (Rx)
rx-usecs	time threshold in microseconds, for receive (Rx)
tx-frames	packet count threshold for transmit confirmation (Tx)
tx-usesc	time threshold in microseconds, for transmit confirmation (Tx)

7.6.2.1.2.5 Header Recognition and Csum Offload

Header recognition on receive (feature provided by eTSEC), combined with parsing functions and/or hashing of extracted property fields (in case of eTSEC2.0), is used to implement advanced TCP/IP offloading functionality and QoS provisions by programming queue filing strategies into hardware.

Gianfar provides an API to program eTSEC's filer hardware block with packet filtering rules.

On Rx, the TCP/IP Offload Engine (TOE):

- can parse frames:
  - at layer 2 of the stack only (Ethernet headers and switching headers)
  - layers 2 to 3 (including IPv4 or IPv6)
  - layers 2 to 4 (including TCP and UDP)
- provides protocol header recognition

- provides header verification (IPv4 header checksum verification)
- provides TCP/UDP payload checksum verification including verification of associated pseudo-header checksums

For large frames, offload of checksum verification saves a significant fraction of the CPU cycles that would otherwise be spent by the TCP/IP stack. IP packet fragmentation and re-assembly, and TCP stream establishment and tear-down are not performed in hardware.

On Tx side, TOE provides IPv4 and TCP/UDP header checksum generation. The eTSEC does not checksum transmitted packets with IPv6 routing headers or calculate TCP/UDP checksums from IP fragments. If a transmitted TCP segment requires checksum generation but IPv6 extension headers would prevent eTSEC from calculating the pseudoheader checksum, software can calculate just the pseudoheader checksum in advance and supply it to the eTSEC as part of per-frame TOE configuration.

On the Rx side, Gianfar lets the Kernel know that checksum verification is not required if valid IP headers or TCP/UDP headers were found and valid sums were verified, by setting the `CHECKSUM_UNNECESSARY` flag. On Tx side, the checksum is generated (offloaded) for TCP/UDP packets over IPv4 based on the pseudo-header checksum *(phcs)* provided by the Linux networking stack. Gianfar instructs the stack about its ability to provide partial checksumming, based on the *phcs* for TCP/UDP packets, by setting the `NETIF_F_IP_CSUM` device capability flag.

The Frame Control Blocks (FCBs) are 8-byte blocks of TOE control and/or status data that are passed between the driver and each eTSEC. A FCB always precedes the frame it applies to, and is present only when TOE functions are being used.

The first BD of each frame points to the initial data buffer and the FCB. Custom or received Ethernet preamble sequences also follow the FCB if preambles are visible.



Figure 177. Location of Frame Control Blocks for TOE Parameters

For Tx, FCBs are inserted by Gianfar and TOE acceleration may be applied on a frame-by-frame basis. In the case of RxBD rings, the FCBs are inserted by eTSEC and TOE acceleration is enabled for receive for all frames in this case.

7.6.2.1.2.6 Scatter Gather Support

Scatter-Gather I/O is a method by which a single procedure call sequentially writes data from multiple buffers to a single data stream or reads data from a data stream to multiple buffers. The buffers are given in a vector of buffers. Scatter/gather refers to the process of gathering data from, or scattering data into, the given set of buffers. The I/O can be performed synchronously or asynchronously to this procedure.

On the Tx side, Gianfar supports "gathering" big packets from multiple buffers. This ability is signaled by the driver to the Linux network stack by setting the `NETIF_F_SG` device hardware feature flag. The driver takes into account the number of fragments composing the packet that is going to be transmitted, and places each fragment into consecutive BD ring buffers before issuing the command to start sending the frame.

On the Rx side, the eTSEC controller is capable of "scattering" big packets into multiple fixed size buffers having consecutive buffer descriptors (BDs). Gianfar supports this feature by implementing paged allocation, so that jumbo frames exceeding a fixed buffer size of 2048 bytes can be automatically received into multiple such buffers. Instead of pre-allocating huge memory buffers to be able to support jumbo frame reception, the paged allocation scheme implemented in Gianfar uses multiple half-page sized buffers thus reducing memory allocation pressure. The driver is also managing a local cache of memory pages, re-using free pages from the cache for future receptions, further improving Rx allocation overhead.

7.6.2.1.3 Configuration & Control

7.6.2.1.3.1 Device Tree initialization

Gianfar complies with the device tree (DTS) based open firmware support requirements, and supports multiple Ethernet device instances. The default configuration parameters that are passed via DTS for an Ethernet device instance (node) include:

1. `compatible` and `model` fields defining the driver compatibility across multiple controller H/W IP generations:

Table 194. Gianfar compatibility

Device type (IP):	`.compatible`	`.model`
eTSEC2.0 (veTSEC)	`"fsl,etsec2"`	`"eTSEC"`
eTSEC (eTSEC1.x)	`"gianfar"`	`"eTSEC"`
TSEC	`"gianfar"`	`"TSEC"`
FEC	`"gianfar"`	`"FEC"`

2. Interrupt grouping of multiple queues, for eTSEC2.0: `queue-group` subnode, including:
   - Interrupt numbers assignment for the Rx, Tx and Error lines, `interrupts` property;
   - Interrupt group register block address and size, `reg` property;
3. Power management capability properties:
   - `fsl,magic-packet`: If present, indicates that the hardware supports waking up via magic packet;
   - `fsl,wake-on-filer`: If present, indicates that the hardware supports waking up by Filer General Purpose Interrupt (FGPI) asserted on the Rx int line. This is an advanced power management capability allowing certain packet types (user) defined by filer rules to wake up the system.
4. For older DTs, number of supported TX and RX queues: `fsl,num-rx-queues` and `fsl,num-tx-queues`; *[obsolete]*
5. Various link management properties.

Typical eTSEC2.0 device tree node (LS1021a example):

```
enet0: ethernet@2d10000 {
    compatible = "fsl,etsec2";
    device_type = "network";
    #address-cells = <2>;
    #size-cells = <2>;
    interrupt-parent = <&gic>;
    model = "eTSEC";
    fsl,magic-packet;
    fsl,wake-on-filer;

    queue-group@2d10000 {
        #address-cells = <2>;
        #size-cells = <2>;
        reg = <0x0 0x2d10000 0x0 0x1000>;
```

```
        interrupts = <GIC_SPI 144 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 145 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 146 IRQ_TYPE_LEVEL_HIGH>;
    };

    queue-group@2d14000  {
        #address-cells = <2>;
        #size-cells = <2>;
        reg = <0x0 0x2d14000 0x0 0x1000>;
        interrupts = <GIC_SPI 147 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 148 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 149 IRQ_TYPE_LEVEL_HIGH>;
    };
};
```

7.6.2.1.3.2 Ethtool support

Table 195. Non-exhaustive list of the most notable *ethtool* commands implemented by Gianfar:

Commands		Description
`ethtool -C`	`rx-usecs N rx-frames N tx-usecs N tx-frames N`	Set interrupt coalescing for a given device, packet count ('frames') and time in microsecs ('usecs') thresholds, for Rx and resp. Tx.
`ethtool -G`	`rx N  tx N`	Set RxBD ring, resp. TxBD ring sizes for a given device.
`ethtool -K`	`rxvlan on|off txvlan on|off`	Turn on/off H/W VLAN tag extraction(rx) / insertion(tx).
`ethtool -S`		Show interface statistics, Linux specific counters and various eTSEC H/W counters supporting RMON MIB group 1, group 2 (ifTable counters), group 3, group 9, RMON MIB 2, and the 802.3 Ethernet MIB statistics.
`ethtool -N`	`rx-flow-hash tcp4|udp4|tcp6|udp6 v|t|s|d|f|n`	Configure Rx network flow classification options. The classified flows may be tcp/udp over ipv4/v6, and the hashing may be performed on various header fields, according to the 3rd parameter: • s,d: src/dest IP addresses; • v: VLAN id; • t: L3 PROTO field, • f,n: source and dest TCP/UDP ports.
`ethtool -N`	`flow-type ether|ip4|tcp4|udp4|sctp4`	Inserts or updates a classification rule for the specified flow type. Most IPv4 flow types are supported: raw IPv4, TCP, UDP, SCTP, as well as L2 flow specifications (ether). For a detailed description of the command sub-options refer to ethtool Linux man-pages.

NOTE

For detailed description of `ethtool` command options refer to ethtool Linux man-pages.

7.7 FlexCAN

7.7.1 FlexCAN User Manual

Description

The FlexCAN module is a communication controller implementing the CAN protocol according to the CAN 2.0B protocol specification. NXP's LS1021A can support 4 Flexcan IP module instances if there are 2 CAN ports brought out on DB9 male connectors on the board.

Dependencies

Hardware:

One LS1021A board, Serial cable (female-to-female) having the following connection schema, so that the DB9 CAN ports on the LS1021A board can be connected properly:



Figure 178. DB9 CAN Connections

Software:

RCW (LS1021A), U-Boot (LS1021A), Kernel (LS1021A), RFS (LS1021A Yocto RFS)

RCW Configuration

- Build the **rcw_1000_misc.bin** RCW file for FlexCAN using the following GIT tree: http://sw-stash.freescale.net/projects/SDK/repos/rcw/browse
- Flash the RCW on the LS1021A board.

U-Boot Configuration

Execute the following command at the u-boot prompt:

=> setenv hwconfig “can”

=> saveenv

Compile time options

N/A

Runtime options

Env Variable	Env Description	Option Description
hwconfig	setenv hwconfig "can"	This option is used to select CAN feature on the LS1021A board.

Kernel Configure Options

Tree View

Following is an example of how to enable FlexCAN driver support.

Kernel Configure Tree View Options	Description

[*] Networking support -->
  <*> CAN bus subsystem support -->
  -- CAN bus subsystem support
  <*> Raw CAN Protocol (raw access with CAN-ID filtering)
  < > Broadcast Manager CAN Protocol (with content filtering)
  CAN Device Drivers -->
  <*> Platform CAN drivers with Netlink support
  [*] CAN bit-timing calculation
  <*> Support for NXP FlexCAN based chips
	Enable FlexCAN driver and CAN protocol stack

NOTE

The FlexCAN driver can be statically/dynamically linked with the kernel Image.

Identifier

Option	Values	Default Value	Description
CONFIG_NET	y/n	y	Enable networking support
CONFIG_CAN	y/m/n	n	Enable CAN bus support
CONFIG_CAN_RAW	y/m/n	n	Enable CAN RAW Protocol support
CONFIG_CAN_DEV	y/m/n	n	Enable platform CAN driver support
CONFIG_CAN_CALC_BITTIMING	y/n	n	Enable baud rate setting using sysfs
CONFIG_CAN_FLEXCAN	y/m/n	n	Enable NXP FlexCAN protocol

Device Tree Binding

Below is the definition of the device tree node required by this feature

Property	Description
compatible = " fsl,ls1021a-flexcan"	Should be "fsl, ls1021a-flexcan"
reg = <0x0 0x2a70000 0x0 0x1000>	Offset and length of the register set for the device
interrupts = < GIC_SPI 127 IRQ_TYPE_LEVEL_HIGH>	IRQ line for FlexCAN controller CAN0
clocks = <&platform_clk 1>	CAN module Clock Source.
clock-names = <per>	CAN Engine Clock Source. This property selects the peripheral clock. Valid values are **ipg** and **per** **ipg**: CAN engine clock source is oscillator clock. Current release doesn't support this option. **per:** The CAN engine clock source is the peripheral clock (platform clock).

Below is an example device tree node required by FlexCAN.

```
can0@2a70000{
           compatible = "fsl, ls1021a-flexcan ";
           reg = <0x0 0x2a70000 0x0 0x1000>;
           interrupts = < GIC_SPI 127 IRQ_TYPE_LEVEL_HIGH >;
           clocks = <&platform_clk 1>;
           clock-names = <per>
};
```

Source Files

The following source file is related to Flexcan feature in u-boot.

Source File	Description
arch/arm/cpu/armv7/ls102xa/fdt.c	Providing the fix-up for clock_freq in the dts file.
board/freescale/ls1021aqds/ls1021aqds.c	Provides the support for board level FlexCAN muxing when u-boot environment variable **hwconfig** is set to can.

The following source files are related to Flexcan feature in Linux kernel.

Source File	Description
drivers/net/can/flexcan.c	Flexcan driver module

Board Connections

On the LS1021A board, mount Jumper 1 as shown in figure below to ensure that HI speed mode is working and also ensure that the 120ohm resistance b/w CAN_H and CAN_L is mounted properly on the board.



Figure 179. Board Connections

User Space Application

Please select the following package for testing Flexcan. Please refer to the LS1 Yocto WiKi page (http://wiki.freescale.net/display/layrarch/Yocto+framework+for+LS1+image+build) for the detailed build procedure.

Command Name	Description	Package Name
cansend	Userspace SocketCAN test program for sending CAN frames on a CAN interface	cansend
candump	Userspace SocketCAN test program for sniffing CAN frames received on a CAN interface	candump
ip	The iproute package contains networking utilities (ip and rtmon, for example) which are designed to use the advanced networking capabilities of the Linux kernel.	iproute(2)

Verification in U-Boot

N/A

Verification in Linux

To cross check whether FlexCAN has been configured in the kernel or not, run the following command at Linux prompt:

```
root@ls1021aqds:~# cat /proc/interrupts
           CPU0       CPU1
118:       3889          0       GIC 118  serial
120:        126          0       GIC 120  2180000.i2c
```

```
125:          0          0       GIC 125  xhci-hcd:usb1
126:       1521          0       GIC 126  mmc0
129:          0          0       GIC 129  2110000.dspi
135:          2          0       GIC 135  1710000.jr
136:          0          0       GIC 136  1720000.jr
137:          0          0       GIC 137  1730000.jr
138:          0          0       GIC 138  1740000.jr
150:       5371          0       GIC 150  Freescale ftm timer
158:          1          0       GIC 158  can0
167:          0          0       GIC 167  eDMA
189:         21          0       GIC 189  eth2_g0_tx
190:          4          0       GIC 190  eth2_g0_rx
191:          0          0       GIC 191  eth2_g0_er
201:          0          0       GIC 201  ds3232
IPI0:         0          0  CPU wakeup interrupts
IPI1:         0       5359  Timer broadcast interrupts
IPI2:      3158       3718  Rescheduling interrupts
IPI3:         0          0  Function call interrupts
IPI4:         3          3  Single function call interrupts
IPI5:         0          0  CPU stop interrupts
Err:          0
```

Test Procedure

Guide lines for testing FlexCAN on the LS1021A board.

Internal LoopBack test (SoC Level loopback)

- Enable CAN interface **can0** with root permissions on the Linux prompt

```
# ip link set can0 up type can bitrate 125000 loopback on
```

- Set candump to snif packets on can0 interface:

```
# candump can0 -n 2 &
```

- Set cansend to send a packet with standard identifier on can0 interface:

```
# cansend can0 5A1#123412341234
```

Expected behavior after internal loopback testing:

- CAN0 interface which is receiving the can frame should show, that it has read the frame ID and data correctly:

```
5A1 [6] 12 34 12 34 12 34
```

- After the successful Tx or RX from the CAN controller, the Tx and Rx Interrupt should increment for CAN0 interface.

```
   root@ls1021aqds:~# cat /proc/interrupts
           CPU0       CPU1
118:       3889          0       GIC 118  serial
120:        126          0       GIC 120  2180000.i2c
125:          0          0       GIC 125  xhci-hcd:usb1
126:       1521          0       GIC 126  mmc0
129:          0          0       GIC 129  2110000.dspi
135:          2          0       GIC 135  1710000.jr
136:          0          0       GIC 136  1720000.jr
137:          0          0       GIC 137  1730000.jr
138:          0          0       GIC 138  1740000.jr
```

```
150:       5371          0       GIC 150  Freescale ftm timer
158:          2          0       GIC 158  can0
167:          0          0       GIC 167  eDMA
189:         21          0       GIC 189  eth2_g0_tx
190:          4          0       GIC 190  eth2_g0_rx
191:          0          0       GIC 191  eth2_g0_er
201:          0          0       GIC 201  ds3232
IPI0:          0          0  CPU wakeup interrupts
IPI1:          0       5359  Timer broadcast interrupts
IPI2:       3158       3718  Rescheduling interrupts
IPI3:          0          0  Function call interrupts
IPI4:          3          3  Single function call interrupts
IPI5:          0          0  CPU stop interrupts
Err:           0
```

NOTE

The underlined number represents the interrupts on successful Tx or RX. So this number should increase on successful Tx and Rx.

External LoopBack test (Single Board).

- Attach the CAN0 and CAN1 interface on the LS1021A board with serial cable (prepared as mentioned in **Dependencies** section).
- Enable the CAN0 and CAN1 interface on the board:

```
# ip link set can0 up type can bitrate 125000
# ip link set can1 up type can bitrate 125000
```

- Set candump to snif packets on can0 interface:

```
# candump can0 -n 1 &
```

- Set cansend to send a packet with standard identifier on can1 interface:

```
# cansend can1 5A1#123412341234
```

Expected behavior after external loopback testing:

- CAN0 interface which is receiving the can frame should show, that it has read the frame ID and data correctly:

```
5A1 [6] 12 34 12 34 12 34
```

- After the successful Tx or RX from the CAN controllers, the Tx Interrupt should increase for CAN1 interface and Rx Interrupt should increment for CAN0 interface.

Benchmarking

TBD

Known Bugs, Limitations, or Technical Issues

Currently we are running Flexcan with Backwards Compatibility Configuration.

Supporting Documentation

SoC Reference Manual (for eg. LS1021A RM)

7.8 Flash Memory

7.8.1 JFFS2 on NOR Flash Device Driver User Manual

Linux SDK for QorIQ Processors

Description

Flash Controller in the SoC includes a NOR flash Machine. NOR flash machine support various type of NOR flash.

NOTE

This User Manual is also valid for Integrated Flash Controller (IFC). Please refer to the "IFC NOR Flash User Manual" to enable IFC NOR.

Module Loading

The Flash device driver supports kernel built-in and module.

U-Boot Configuration

Env Variable	Env Description	Sub Option	Option Description
bootargs	Kernel command line argument passed to kernel	setenv bootargs root=/dev/mtdblockx rootfstype=jffs2 rw console=ttyS0,115200	Uses NOR jffs2 filesystem as rootfs filesystem , and x should be the NOR jffs2 partition's mtdblock number.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`  <*> Memory Technology Device (MTD) support  --->`
`  --- Memory Technology Device (MTD) support`
`  [*] Command line partition table parsing`
`  <*> Flash partition map based on of descriptbion`
`  <*> Direct char device access to MTD devices`
`  <*> Caching block device access to MTD devices`
`  RAM/ROM/Flash chip drivers --->`
`  <*> Detect flash chips by Common Flash Interface (CFI) probe`
`  <*> Support for AMD/Fujitsu flash chips`
`  Mapping Drivers for chip access --->`
`  <*> Flash device in physical memory map based on OF description`
	MTD and AMD flash support

`File systems --->`
`  Miscellaneous filesystems --->`
`  <*> Journaling Flash File System v2 (JFFS2) support`
`  (0) JFFS2 debugging verbosity`
`  [*] JFFS2 write-buffering support`
	JFFS2 FS support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_MTD	y/m/n	y	Enable Memory Technology Devices support
CONFIG_MTD_CMDLINE_PARTS	y/n	y	Enable parsing partition via kernel command line
CONFIG_MTD_BLOCK	y/m/n	y	Enables block driver for MTD device
CONFIG_MTD_OF_PARTS	y/m/n	y	Enables partition parsing of partition map from the children of the flash node
ONFIG_MTD_CFI_AMDSTD	y/m/n	y	Enables support for AMD/Fujitsu flash chips
ONFIG_MTD_PHYSMAP_OF	y/m/n	y	Enables physical memory map based on OF description
CONFIG_JFFS2_FS	y/m/n	y	Enable Journal Flash FS v2 support
CONFIG_JFFS2_FS_WRITEBUFFER	y/n	y	Enable JFFS2 write-buffering support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
/linux/drivers/mtd	Linux MTD subsystem
/linux/drivers/mtd/chips	Linux MTD NOR device support
/linux/fs/jffs2	Linux JFFS2 FS system

In U-Boot, flash the JFFS2 FS to the NOR flash

Note: the erase block size for the NOR flash device (as example) is 128KB. Generate the JFFS2 FS accordantly.

```
=> tftp 1000000 $nor_jffs2_fs
=> erase $nor_jffs_start_address $nor_jffs_end_address /* erase address and size is decided by
your partition setting */
=> cp.b 1000000 $nor_jffs_start_address $filesize /* flash JFFS2 FS to the jffs2 partition on NOR
device */

The nor_jffs_start_address and nor_jffs_end_addresscan get from (board)_Quick_Strat_Guide or
(board)_user_manual.
```

Special MTD partition DTS

Note: The mtdblock number for JFFS2 partitions on NOR device is decided by your system configuration and your partition setting.

Use mtdparts=fe8000000.nor:1m(uboot),8m(kernel),512k(dtb),11m(jffs2),-(fs) in the bootargs for NOR partitions on B4860QDS.

Boot up the board

Note: Different boards have different partition settings. The following example is B4860QDS board's partition setting.

The mtdblock3 is NOR JFFS2 partition.

```
fe8000000.nor: Found 1 x16 devices at 0x0 in 16-bit bank. Manufacturer ID 0x000001 Chip ID
0x002801
Amd/Fujitsu Extended Query Table at 0x0040
Amd/Fujitsu Extended Query version 1.5.
number of CFI chips: 1
5 cmdlinepart partitions found on MTD device fe8000000.nor
Creating 5 MTD partitions on "fe8000000.nor":
0x000000000000-0x000000100000 : "uboot"
0x000000100000-0x000000900000 : "kernel"
0x000000900000-0x000000980000 : "dtb"
0x000000980000-0x000001480000 : "jffs2"
0x000001480000-0x000008000000 : "fs"
....
....
/* log in kernel */
root@b4860qds:~# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00100000 00020000 "uboot"
mtd1: 00800000 00020000 "kernel"
mtd2: 00080000 00020000 "dtb"
mtd3: 00b00000 00020000 "jffs2"
mtd4: 06b80000 00020000 "fs"
mtd5: 00100000 00020000 "NAND U-Boot Image"
mtd6: 00100000 00020000 "NAND DTB Image"
mtd7: 00a00000 00020000 "NAND Linux Kernel Image"
mtd8: 1f400000 00020000 "NAND RFS Image"
mtd9: 00080000 00001000 "spife110000.0"

[root@b4860qds root]# ls /dev/mtdblock3 -l /* the block number should be the NOR jffs2
partition's mtdblock number*/
brw-r----- 1 root disk 31, 4 Mar  5 14:13 /dev/mtdblock3
[root@b4860ds root]# mount -t jffs2 /dev/mtdblock3 /mnt/
[root@b4860ds root]# ls /mnt/
bin  etc   lib      log  opt   root  sys  usr
dev  home  linuxrc  mnt  proc  sbin  tmp  var
[root@b4860qds root]# umount /mnt/
[root@b4860qds root]# ls /mnt/
cdrom  floppy  nfs  rwfs  src
[root@b4860qds root]#
```

Another way to test NOR JFFS2 support is to boot up kernel and mount JFFS2 filesystem

Note: the block number should be the NOR JFFS2 partition's mtdblock number

Under U-Boot prompt:

```
setenv bootargs root=/dev/mtdblock3 rootfstype=jffs2 rw console=ttyS0,115200
tftp 1000000 uImage
tftp c00000 dtb
bootm 1000000 - c00000
```

7.8.2 JFFS2 on NAND Flash Device Driver User Manual

Linux SDK for QorIQ Processors

Description

The Enhanced Local Bus Controller in the platform includes a special NAND flash Control Machine (FCM). FCM provides the hardware support for using small or large page NAND flash.

NOTE

This User Manual is written for Enhanced Local Bus Controller (eLBC). It is also valid for Integrated Flash Controller (IFC). Please refer to "IFC NAND Flash User Manual" for enabling IFC NAND.

Module Loading

The eLBC device driver supports kernel built-in and module.

U-Boot Configuration

Env Variable	Env Description	Sub Option	Option Description
bootargs	Kernel command line argument passed to kernel	setenv bootargs root=/dev/mtdblockx rootfstype=jffs2 rw console=ttyS0,115200	Uses NAND JFFS2 filesystem as rootfs filesystem, and x should be the NAND JFFS2 partition's mtdblock number.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`  <*> Memory Technology Device (MTD) support  --->`
`  --- Memory Technology Device (MTD) support`
`  [*] Command line partition table parsing`
`  <*> Direct char device access to MTD devices`
`  <*> Caching block device access to MTD devices`
`  Mapping Drivers for chip access --->`
`  <*> Flash device in physical memory map based on OF description`
`  <*> NAND Device support --->`
`  [*] NAND support for Freescale eLBC contorllers`
	MTD and eLBC NAND support

`File systems --->`
`  Miscellaneous filesystems --->`
`  <*> Journaling Flash File System v2 (JFFS2) support`
`  (0) JFFS2 debugging verbosity`
`  [*] JFFS2 write-buffering support`
	JFFS2 FS support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_MTD	y/m/n	y	Enable Memory Technology Devices support
CONFIG_MTD_CMDLINE_PARTS	y/n	y	Enable parsing partition via kernel command line
CONFIG_MTD_BLOCK	y/m/n	y	Enables block driver for MTD device
CONFIG_MTD_OF_PARTS	y/m/n	y	Enables partition parsing of partition map from the children of the flash node
CONFIG_MTD_NAND	y/m/n	y	Enables MTD NAND subsystem
CONFIG_MTD_NAND_FSL_ELBC	y/n	y	Enables NXP ELBC NAND device driver
CONFIG_JFFS2_FS	y/m/n	y	Enable Journal Flash FS v2 support
CONFIG_JFFS2_FS_WRITEBUFFER	y/n	y	Enable JFFS2 write-buffering support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
/linux/drivers/mtd	Linux MTD subsystem
/linux/drivers/mtd/nand	Linux MTD NAND device support
/linux/fs/jffs2	Linux JFFS2 FS system

In U-boot, flash the JFFS2 FS to the NAND flash

Note: the erase block size for the NAND flash device (as example) is 128KB. Generate the JFFS2 FS accordantly.

```
=> tftp 1000000 $nand_jffs2_fs
=> nand erase $nand_jffs2_start_address $nand_jffs2_size /*erase address and size is decided by
your partition setting */
=> nand write 1000000 $nand_jffs2_start_address $filesize /* flash JFFS2 FS to the jffs2-nand
partition of NAND device */
The nand_jffs2_start_address and nand_jffs2_size can get from (board)_Quick_Strat_Guide or
(board)_user_manual.
```

Special MTD partition DTS

Note: The mtdblock number for JFFS2 partitions on NAND device is decided by your system configuration and your partition setting.

Using 'mknod' command to create the corresponding mtdblock device node if it does not exist in the file system.

```
nand@2,0 {
         #address-cells = "<1>";
         #size-cells = "<1>";
         compatible = ""fsl,mpc8572-fcm-nand","fsl,elbc-fcm-nand";
         reg = <0x2 0x0 0x40000>;
         u-boot@0 {
                 reg = <0x0 0x02000000>;
                 label = "NAND - U-Boot Image";
```

```
                read-only;
        };
        jffs2@2000000 {
                reg = <0x02000000 0x10000000>;
                label = "NAND - JFFS2 Root File System";
        };
        ramdisk@12000000 {
                reg = <0x12000000 0x08000000>;
                label = "NAND - RAMDISK File System";
                read-only;
        };
        kernel@1a000000 {
                reg = <0x1a000000 0x04000000>;
                label = "NAND - Linux Kernel Image";
        };
        dtb@1e000000 {
                reg = <0x1e000000 0x01000000>;
                label = "NAND - DTB Image";
                read-only;
        };
        empty@1f000000 {
                reg = <0x1f000000 0x21000000>;
                label = "NAND - Empty";
        };
};
nand@4,0 {
        compatible = ""fsl,mpc8572-fcm-nand","fsl,elbc-fcm-nand";
        reg = <0x4 0x0 0x40000>;
};
nand@5,0 {
        compatible = ""fsl,mpc8572-fcm-nand","fsl,elbc-fcm-nand";
        reg = <0x5 0x0 0x40000>;
};
nand@6,0 {
        compatible = ""fsl,mpc8572-fcm-nand","fsl,elbc-fcm-nand";
        reg = <0x6 0x0 0x40000>;
```

Boot up the board

Note: The following example assumes that only NAND is enabled, NOR is not enabled.

Otherwise, the mtdblock number for NAND will be different, e.g. mtdblock1 will be NAND JFFS2 partition

```
....
NAND device: Manufacturer ID: 0xec, Chip ID: 0xd3 (Samsung NAND 1GiB 3,3V 8-bit)

nand_bbt: ECC error while reading bad block table

RedBoot partition parsing not available

Creating 6 MTD partitions on "ffa00000.flash":

0x00000000-0x02000000 : "NAND - U-Boot Image"

0x02000000-0x12000000 : "NAND - JFFS2 Root File System"

0x12000000-0x1a000000 : "NAND - RAMDISK File System"

0x1a000000-0x1e000000 : "NAND - Linux Kernel Image"
```

```
0x1e000000-0x1f000000 : "NAND - DTB Image"

0x1f000000-0x40000000 : "NAND - Empty"

eLBC NAND device at 0xffa00000, bank 2

NAND device: Manufacturer ID: 0xec, Chip ID: 0xd3 (Samsung NAND 1GiB 3,3V 8-bit)

nand_bbt: ECC error while reading bad block table

RedBoot partition parsing not available

eLBC NAND device at 0xffa40000, bank 4

NAND device: Manufacturer ID: 0xec, Chip ID: 0xd3 (Samsung NAND 1GiB 3,3V 8-bit)

nand_bbt: ECC error while reading bad block table

RedBoot partition parsing not available

eLBC NAND device at 0xffa80000, bank 5

NAND device: Manufacturer ID: 0xec, Chip ID: 0xd3 (Samsung NAND 1GiB 3,3V 8-bit)

nand_bbt: ECC error while reading bad block table

RedBoot partition parsing not available

eLBC NAND device at 0xffac0000, bank 6

sh-2.05b# cat /proc/mtd

dev:    size   erasesize  name

mtd0: 02000000 00020000 "NAND - U-Boot Image"

mtd1: 10000000 00020000 "NAND - JFFS2 Root File System"

mtd2: 08000000 00020000 "NAND - RAMDISK File System"

mtd3: 04000000 00020000 "NAND - Linux Kernel Image"

mtd4: 01000000 00020000 "NAND - DTB Image"

mtd5: 21000000 00020000 "NAND - Empty"

mtd6: 40000000 00020000 "ffa40000.flash"

mtd7: 40000000 00020000 "ffa80000.flash"

mtd8: 40000000 00020000 "ffac0000.flash"

...

/* log in kernel */
-sh-2.05b #
-sh-2.05b # mount -t jffs2 /dev/mtdblock2 /mnt/cdrom/
-sh-2.05b # ls /mnt/cdrom
bin   dev  home  linuxrc  opt   root  sys  usr
```

```
boot  etc  lib   mnt      proc  sbin  tmp  var
-sh-2.05b # umount /mnt/cdrom
```

Another way to test NAND JFFS2 support is to boot up kernel and mount JFFS2 filesystem

Note: the block number should be the NAND JFFS2 partition's mtdblock number

Under U-Boot prompt:

```
setenv bootargs root=/dev/mtdblock2 rootfstype=jffs2 rw console=ttyS0,115200
tftp 1000000 uImage
tftp c00000 dtb
bootm 1000000 - c00000
```

7.8.3 Integrated Flash Controller NOR Flash User Manual

Description

NXP's Integrated Flash Controller can be used to connect various types of flashes e.g. NOR/NAND on board for boot functionality as well as data storage.

U-Boot Configuration

Compile time options

Below are major u-boot configuration options related to this feature defined in platform specific config files under include/configs/ directory.

Option Identifier	Description
CONFIG_FSL_IFC	Enable IFC support
CONFIG_FLASH_CFI_DRIVER CONFIG_SYS_FLASH_CFI CONFIG_SYS_FLASH_EMPTY_INFO	Enable CFI Driver for NOR Flash devices

Source Files

The following source files are related to this feature in u-boot.

Source File	Description
./drivers/misc/fsl_ifc.c	Set up the different chip select parameters from board header file
drivers/mtd/cfi_flash.c	CFI driver support for NOR flash devices

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree View Options	Description

Device Drivers --->
  <*> Memory Technology Device (MTD) support --->
  [*] MTD partitioning support
  [*] Command line partition table parsing
  <*> Flash partition map based on OF description
  <*> Direct char device access to MTD devices
  -*- Common interface to block layer for MTD 'translation layers'
  <*> Caching block device access to MTD devices
  < > FTL (Flash Translation Layer) support
  RAM/ROM/Flash chip drivers --->
  <*> Detect flash chips by Common Flash Interface (CFI) probe
  <*> Support for Intel/Sharp flash chips
  <*> Support for AMD/Fujitsu/Spansion flash chips
  Mapping drivers for chip access --->
  <*> Flash device in physical memory map based on OF description
	These options enable CFI support for NOR Flash under MTD subsystem and Integrated Flash Controller support on Linux

File systems ---> [*] Miscellaneous filesystems ---> <*> Journalling Flash File System v2 (JFFS2) support	This option enables JFFS2 file system support for MTD Devices

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Special Configure needs to be enabled("Y") for LS1021 and LS1043. Please find in below table with default value as "N"

Option	Values	Default Value	Description
CONFIG_FSL_IFC	Y/N	Y	Integrated Flash Controller support
CONFIG_MTD	Y/N	Y	Memory Technology Device (MTD) support

Table continues on the next page...

Table continued from the previous page...

Option	Values	Default Value	Description
CONFIG_MTD_PARTITIONS	Y/N	Y	MTD partitioning support
CONFIG_MTD_CMDLINE_PARTS	Y/N	Y	Allow generic configuration of the MTD partition tables via the kernel command line.
CONFIG_MTD_OF_PARTS	Y/N	Y	This provides a partition parsing function which derives the partition map from the children of the flash nodes described in Documentation/powerpc/booting-without-of.txt
CONFIG_MTD_CHAR	Y/N	Y	Direct char device access to MTD devices
CONFIG_MTD_BLOCK	Y/N	Y	Caching block device access to MTD devices
CONFIG_MTD_CFI	Y/N	Y	Detect flash chips by Common Flash Interface (CFI) probe
CONFIG_MTD_GEN_PROBE	Y/N	Y	NA
CONFIG_MTD_MAP_BANK_WIDTH_1	Y/N	Y	Support 8-bit buswidth
CONFIG_MTD_MAP_BANK_WIDTH_2	Y/N	Y	Support 16-bit buswidth
CONFIG_MTD_MAP_BANK_WIDTH_4	Y/N	Y	Support 32-bit buswidth
CONFIG_MTD_PHYSMAP_OF	Y/N	Y	Flash device in physical memory map based on OF description
CONFIG_FTL	Y/N	N	FTL (Flash Translation Layer) support
CONFIG_MTD_CFI_INTELEXT	Y/N	Y	Support for Intel/Sharp flash chips
CONFIG_MTD_CFI_AMDSTD	Y/N	Y	Support for AMD/Fujitsu/Spansion flash chips
CONFIG_MTD_CFI_ADV_OPTIONS	Y/N	N	Enable only for LS1021 and LS1043
CONFIG_MTD_CFI_BE_BYTE_SWAP	Y/N	N	Enable only for LS1021 and LS1043

Device Tree Binding

Documentation/devicetree/bindings/powerpc/fsl/ifc.txt

Documentation/devicetree/bindings/memory-controllers/fsl/ifc.txt

Flash partitions are specified by platform device tree.

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/memory/fsl_ifc.c	Integrated Flash Controller driver to handle error interrupts
drivers/mtd/mtdpart.c	Simple MTD partitioning layer
drivers/mtd/mtdblock.c	Direct MTD block device access

Table continues on the next page...

Table continued from the previous page...

Source File	Description
drivers/mtd/mtdchar.c	Character-device access to raw MTD devices.
drivers/mtd/ofpart.c	Flash partitions described by the OF (or flattened) device tree
drivers/mtd/ftl.c	FTL (Flash Translation Layer) support
drivers/mtd/chips/cfi_probe.c	Common Flash Interface probe
drivers/mtd/chips/cfi_util.c	Common Flash Interface support
drivers/mtd/chips/cfi_cmdset_0001.c	Support for Intel/Sharp flash chips
drivers/mtd/chips/cfi_cmdset_0002.c	Support for AMD/Fujitsu/Spansion flash chips

Verification in U-Boot

Test the Read/Write/Erase functionality of NOR Flash

1. Boot the u-boot with above config options to get NOR Flash access enabled. Check this in boot log,

   ```
   FLASH: * MiB
   ```

   where * is the size of NOR Flash
2. Erase NOR Flash
3. Make test pattern on memory e.g. DDR
4. Write test pattern on NOR Flash
5. Read the test pattern from NOR Flash to memory e.g DDR
6. Compare the test pattern data to verify functionality.

Test Log :

Test log with initial u-boot log removed

```
--
--

FLASH: 32 MiB
L2:    256 KB enabled
--
--
/* u-boot prompt */
=> mw.b 1000000 0xa5 10000
=> md 1000000
01000000: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000010: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000020: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000030: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000040: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000050: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000060: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000070: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000080: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000090: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000a0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000b0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
```

```
010000c0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000d0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000e0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000f0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
=> protect off all
Un-Protect Flash Bank # 1
=> erase ee000000 ee01ffff

. done
Erased 1 sectors
=> cp.b 1000000 ee000000 10000
Copy to Flash... 9....8....7....6....5....4....3....2....1....done
=> cmp.b 1000000 ee000000 10000
Total of 65536 bytes were the same
=>
```

Verification in Linux

To cross check whether IFC NOR driver has been configured in the kernel or not, see the kernel boot log with following entries. Please note mtd partition number can be changed depending upon device tree.

```
ee000000.nor: Found 1 x16 devices at 0x0 in 16-bit bank

Amd/Fujitsu Extended Query Table at 0x0040

number of CFI chips: 1

RedBoot partition parsing not available

Creating 4 MTD partitions on "ee000000.nor":

0x000000040000-0x000000080000 : "NOR DTB Image"

ata1: Signature Update detected @ 0 msecs

0x000000080000-0x000000780000 : "NOR Linux Kernel Image"

0x000000800000-0x000001c00000 : "NOR JFFS2 Root File System"

0x000001f00000-0x000002000000 : "NOR U-Boot Image"
```

To verify NOR flash device accesses see the following test,

```
[root@ root]# cat /proc/mtd

dev:    size   erasesize  name

mtd0: 00040000 00020000 "NOR DTB Image"

mtd1: 00700000 00020000 "NOR Linux Kernel Image"

mtd2: 01400000 00020000 "NOR JFFS2 Root File System"

mtd3: 00100000 00020000 "NOR U-Boot Image"

mtd4: 00100000 00004000 "NAND U-Boot Image"
```

```
mtd5: 00100000 00004000 "NAND DTB Image"

mtd6: 00400000 00004000 "NAND Linux Kernel Image"

mtd7: 00400000 00004000 "NAND Compressed RFS Image"

mtd8: 00f00000 00004000 "NAND JFFS2 Root File System"

mtd9: 00700000 00004000 "NAND User area"

mtd10: 00080000 00010000 "SPI (RO) U-Boot Image"

mtd11: 00080000 00010000 "SPI (RO) DTB Image"

mtd12: 00400000 00010000 "SPI (RO) Linux Kernel Image"

mtd13: 00400000 00010000 "SPI (RO) Compressed RFS Image"

mtd14: 00700000 00010000 "SPI (RW) JFFS2 RFS"

[root@ root]# flash_eraseall -j /dev/mtd2

Erasing 128 Kibyte @ 1400000 -- 100% complete. Cleanmarker written at 13e0000.

[root@P1010RDB root]# mount -t jffs2 /dev/mtdblock2 /mnt/

JFFS2 notice: (1202) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0 of xdatum
(0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.

[root@ root]# cd /mnt/

[root@ mnt]# ls -l

[root@ mnt]# touch flash_file

[root@ root]# umount mnt
//ls must list local_file
[root@ root]# ls mnt
//mount again
[root@ root]# mount -t  jffs2 /dev/mtdblock2 /mnt/
JFFS2 notice: (1219) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0 of xdatum
(0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.
//use ls ; it must show the created file
[root@ root]# ls /mnt/
flash_file
//unmount
[root@ root]# umount /mnt/
```

7.8.4 Integrated Flash Controller NAND Flash User Manual

Description

NXP's Integrated Flash Controller can be used to connect various types of flashes (e.g. NOR/NAND) on board for boot functionality as well as data storage.

U-Boot Configuration

Compile time options

Below are major U-Boot configuration options related to this feature defined in platform specific config files under include/ configs/ directory.

Option Identifier	Description
CONFIG_FSL_IFC	Enable IFC support
CONFIG_NAND_FSL_IFC	Enable NAND Machine support on IFC
CONFIG_SYS_MAX_NAND_DEVICE	No of NAND Flash chips on platform
CONFIG_MTD_NAND_VERIFY_WRITE	Verify NAND flash writes
CONFIG_CMD_NAND	Enable various commands support for NAND Flash
CONFIG_SYS_NAND_BLOCK_SIZE	Block size of the NAND flash connected on Platform

Source Files

The following source files are related to this feature in u-boot.

Source File	Description
./drivers/misc/fsl_ifc.c	Set up the different chip select parameters from board header file
drivers/mtd/nand/fsl_ifc_nand.c	IFC nand flash machine driver file

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree View Options	Description

Device Drivers --->
  <*> Memory Technology Device (MTD) support --->
  [*] MTD partitioning support
  [*] Command line partition table parsing=
  <*> Flash partition map based on OF description
  <*> Direct char device access to MTD devices
  -*- Common interface to block layer for MTD 'translation layers'
  <*> Caching block device access to MTD devices
	These options enable Integrated Flash Controller NAND support to work with MTD subsystem available on Linux.
	Also UBIFS support needs to be enabled.

Kernel Configure Tree View Options	Description
<*> NAND Device Support ---> <*> NAND support for Freescale IFC controller **Enable UBIFS filesystem in linux configuration** Device Drivers ---> <*> Memory Technology Device (MTD) support ---> UBI - Unsorted block images ---> <*> Enable UBI (4096) UBI wear-leveling threshold (1) Percentage of reserved eraseblocks for bad eraseblocks handling < > MTD devices emulation driver (gluebi) *** UBI debugging options *** [ ] UBI debugging File systems ---> [*] Miscellaneous filesystems ---> <*> UBIFS file system support [*] Extended attributes support [ ] Advanced compression options [ ] Enable debugging	

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_FSL_IFC	y/n	y	Enable Integrated Flash Controller support
CONFIG_MTD_NAND_FSL_IFC	y/n	Y	Enable Integrated Flash Controller NAND Machine support
CONFIG_MTD_PARTITIONS	y/n	Y	MTD partitioning support
CONFIG_MTD_CMDLINE_PARTS	y/n	Y	Allow generic configuration of the MTD partition tables via the kernel command line.
CONFIG_MTD_OF_PARTS	y/n	Y	This provides a partition parsing function which derives the partition map from the children of the flash nodes described in Documentation/powerpc/booting-without-of.txt
CONFIG_MTD_CHAR	y/n	Y	Direct char device access to MTD devices
CONFIG_MTD_BLOCK	y/n	Y	Caching block device access to MTD devices
CONFIG_MTD_GEN_PROBE	y/n	Y	NA
CONFIG_MTD_PHYSMAP_OF	y/n	Y	Flash device in physical memory map based on OF description

Device Tree Binding

`Documentation/devicetree/bindings/memory-controllers/fsl/ifc.txt`

Flash partitions are specified by platform device tree.

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/memory/fsl_ifc.c	Integrated Flash Controller driver to handle error interrupts
drivers/mtd/nand/fsl_ifc_nand.c	Integrated Flash Controller NAND Machine driver
include/linux/fsl_ifc.h	IFC Memory Mapped Registers

Verification in U-Boot

Test the Read/Write/Erase functionality of NAND Flash

1. Boot the u-boot with above config options to get NAND Flash driver enabled. Check this in boot log,

   ```
   NAND: * MiB

   Where * is NAND flash size
   ```

2. Erase NAND Flash
3. Make test pattern on memory e.g. DDR
4. Write test pattern on NAND Flash
5. Read the test pattern from NAND Flash to memory e.g DDR
6. Compare the test pattern data to verify functionality.

Test Log :

```
...
...

Flash: 32 MiB

L2:    256 KB enabled

NAND:  32 MiB

...
...

/* U-boot prompt */
=> nand erase.chip



NAND erase.chip: device 0 whole chip

Bad block table found at page 65504, version 0x01 Bad block table found at page 65472, version
0x01

Skipping bad block at  0x01ff0000


Skipping bad block at  0x01ff4000


Skipping bad block at  0x01ff8000


Skipping bad block at  0x01ffc000




OK

=> mw.b 1000000 0xa5 100000

=> md 1000000

01000000: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

01000010: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

01000020: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

01000030: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

01000040: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

01000050: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

01000060: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

01000070: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
```

```
01000080: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

01000090: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

010000a0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

010000b0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

010000c0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

010000d0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

010000e0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

010000f0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

=> nand write 1000000 0 100000


NAND write: device 0 offset 0x0, size 0x100000

 1048576 bytes written: OK

=> nand read 2000000 0 100000


NAND read: device 0 offset 0x0, size 0x100000

 1048576 bytes read: OK

=>  cmp.b 1000000 2000000 100000

Total of 1048576 bytes were the same
```

Verification in Linux

To cross check whether IFC NAND driver has been configured in the kernel or not, check the following. Please note mtd partition numbers can be changed depending upon board device tree

```
[root@(none) root]# cat /proc/mtd

dev:   size   erasesize  name

mtd0: 00100000 00004000 "NAND U-Boot Image"

mtd1: 00100000 00004000 "NAND DTB Image"

mtd2: 00400000 00004000 "NAND Linux Kernel Image"

mtd3: 00400000 00004000 "NAND Compressed RFS Image"

mtd4: 00f00000 00004000 "NAND Root File System"

mtd5: 00700000 00004000 "NAND User area"
```

```
mtd6: 00080000 00010000 "SPI (RO) U-Boot Image"

mtd7: 00080000 00010000 "SPI (RO) DTB Image"

mtd8: 00400000 00010000 "SPI (RO) Linux Kernel Image"

mtd9: 00400000 00010000 "SPI (RO) Compressed RFS Image"

mtd10: 00700000 00010000 "SPI (RW) JFFS2 RFS"


[root@(none) root]# flash_eraseall /dev/mtd4 Erasing 16 Kibyte @ f00000 -- 100% complete.


[root@(none) root]# ubiattach /dev/ubi_ctrl -m 4

UBI: attaching mtd4 to ubi0

UBI: physical eraseblock size:   16384 bytes (16 KiB)

UBI: logical eraseblock size:    15360 bytes

UBI: smallest flash I/O unit:    512

UBI: VID header offset:          512 (aligned 512)

UBI: data offset:                1024

UBI: empty MTD device detected

UBI: create volume table (copy #1)

UBI: create volume table (copy #2)

UBI: attached mtd4 to ubi0

UBI: MTD device name:            "NAND Root File System"

UBI: MTD device size:            15 MiB

UBI: number of good PEBs:        960

UBI: number of bad PEBs:         0

UBI: max. allowed volumes:       89

UBI: wear-leveling threshold:    4096

UBI: number of internal volumes: 1

UBI: number of user volumes:     0

UBI: available PEBs:            947

UBI: total number of reserved PEBs: 13

UBI: number of PEBs reserved for bad PEB handling: 9
```

```
UBI: max/mean erase counter: 0/0

UBI: image sequence number: 0

UBI: background thread "ubi_bgt0d" started, PID 7541 UBI device number 0, total 960 LEBs
(14745600 bytes, 14.1 MiB), available 947 LEBs (14545920 bytes, 13.9 MiB), LEB size 15360 bytes
(15.0 KiB)



[root@(none) root]# ubimkvol /dev/ubi0 -N rootfs -s 14205KiB Volume ID 0, size 947 LEBs (14545920
bytes, 13.9 MiB), LEB size 15360 bytes (15.0 KiB), dynamic, name "rootfs", alignment 1



[root@(none) root]# mount -t ubifs /dev/ubi0_0 /mnt/

UBIFS: default file-system created

UBIFS: mounted UBI device 0, volume 0, name "rootfs"

UBIFS: file system size:   14361600 bytes (14025 KiB, 13 MiB, 935 LEBs)
UBIFS: journal size:       721920 bytes (705 KiB, 0 MiB, 47 LEBs)

UBIFS: media format:       w4/r0 (latest is w4/r0)

UBIFS: default compressor: lzo

UBIFS: reserved for root:  678333 bytes (662 KiB)



[root@(none) root]# cd /mnt/

[root@(none) mnt]# ls

[root@(none) mnt]# touch flash_file

[root@(none) mnt]# ls -l

total 0

-rw-r--r-- 1 root root 0 Jul  6 14:45 flash_file

[root@(none) mnt]# cd

[root@(none) root]# umount /mnt/

UBIFS: un-mount UBI device 0, volume 0



[root@(none) root]# mount -t ubifs /dev/ubi0_0 /mnt/

UBIFS: mounted UBI device 0, volume 0, name "rootfs"

UBIFS: file system size:   14361600 bytes (14025 KiB, 13 MiB, 935 LEBs)
UBIFS: journal size:       721920 bytes (705 KiB, 0 MiB, 47 LEBs)

UBIFS: media format:       w4/r0 (latest is w4/r0)
```

```
UBIFS: default compressor: lzo

UBIFS: reserved for root:  678333 bytes (662 KiB)



[root@(none) root]# ls -l /mnt/

total 0

-rw-r--r-- 1 root root 0 Jul  6 14:45 flash_file
```

Known Bugs, Limitations, or Technical Issues

Boards which have NAND Flash with 512byte page size, JFFS2 cannot be supported using H/W ECC support of IFC , as there is not enough remaining space in the OOB area.

To use JFFS2 use SOFT ECC.

7.9 IEEE 1588 Device Driver User Manual

7.9.1 IEEE 1588 Device Driver User Manual

Description

From IEEE-1588 perspective the components required are:

1. IEEE-1588 extensions to the gianfar driver or DPAA/DPAA2 driver.
2. A stack application for IEEE-1588 protocol.

Module Loading

IEEE 1588 device driver supports either kernel built-in or module

Kernel Configure Tree View Options

1. eTSEC - Using IXXAT stack

Kernel Configure Tree View Options	Description
Device Drivers ---> [*]Network device support ---> [*]Ethernet driver support ---> <*> Gianfar Ethernet [*] Gianfar 1588	Enable 1588 driver for IXXAT stack

2. eTSEC - Using PTPd stack

Kernel Configure Tree View Options	Description
Device Drivers ---> PTP clock support ---> <*> Freescale eTSEC as PTP clock	Enable 1588 driver for PTPd stack

3. DPAA - Using IXXAT stack

Kernel Configure Tree View Options	Description
Device Drivers ---> [*]Network device support ---> [*]Ethernet driver support ---> [*]Freescale devices ---> <*>IEEE 1588-compliant timestamping Optimization choices for the DPAA Ethernet driver ---> (X)Optimize for forwarding	Enable IEEE 1588-compliant timestamping

4. DPAA - Using PTPd stack

Kernel Configure Tree View Options	Description
Device Drivers ---> PTP clock support ---> <*> Freescale DPAA as PTP clock	Enable 1588 driver for PTPd stack

5. DPAA2 - Using PTPd stack

Kernel Configure Tree View Options	Description
Device Drivers ---> PTP clock support ---> <*> Freescale DPAA2 as PTP clock	Enable 1588 driver for PTPd stack

Compile-time Configuration Options

For eTSEC (IXXAT)

Option	Values	Default Value	Description
CONFIG_GIANFAR	y/n	y	Enable eTSEC driver support
CONFIG_FSL_GIANFAR_1588	y/n	n	Enables 1588 driver support

For eTSEC (PTPd)

Option	Values	Default Value	Description
CONFIG_GIANFAR	y/n	y	Enable eTSEC driver support
CONFIG_PTP_1588_CLOCK_GIANFAR	y/n	y	Enables 1588 driver support

For DPAA (IXXAT)

Option	Values	Default Value	Description
CONFIG_FSL_DPAA_1588	y/n	n	Enable IEEE 1588 support
CONFIG_FSL_SDK_DPAA_ETH	y/n	y	Enables DPAA driver support

For DPAA (PTPd)

Option	Values	Default Value	Description
CONFIG_PTP_1588_CLOCK_DPAA	y/n	n	Enable IEEE 1588 support
CONFIG_FSL_SDK_DPAA_ETH	y/n	y	Enables DPAA driver support

For DPAA2 (PTPd)

Option	Values	Default Value	Description
CONFIG_PTP_1588_CLOCK_DPAA2	y/n	n	Enable IEEE 1588 support
CONFIG_FSL_DPAA2_ETH	y/n	y	Enables DPAA2 driver support

Source Files

The driver source is maintained in the Linux kernel source tree.

1. eTSEC (for IXXAT)

Source File	Description
drivers/net/ethernet/freescale/gianfar.c	IEEE 1588 hooks in the Ethernet driver
drivers/net/ethernet/freescale/gianfar_1588.c	IEEE 1588 driver

2. eTSEC (for PTPd)

Source File	Description
drivers/net/ethernet/freescale/gianfar.c	IEEE 1588 hooks in the Ethernet driver
drivers/net/ethernet/freescale/gianfar_ptp.c	IEEE 1588 driver

3. DPAA (for IXXAT)

Source File	Description
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_1588.c	IEEE 1588 driver support
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_1588.h	IEEE 1588 driver head file
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_eth.c	DPAA Ethdriver support

4. DPAA (for PTPd)

Source File	Description
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_ptp.c	IEEE 1588 driver support
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_eth.c	DPAA Ethdriver support

5. DPAA2 (for PTPd)

Source File	Description
drivers/staging/fsl-dpaa2/rtc/rtc.c	IEEE 1588 driver support
drivers/staging/fsl-dpaa2/ethernet/dpaa2-eth.c	DPAA Ethdriver support

Device Tree Binding

1. eTSEC (for IXXAT)

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,gianfar-ptp-timer' for IEEE 1588 driver

```
Default node:
         ptp_timer: ptimer@24e00 {
                 compatible = "fsl,gianfar-ptp-timer";
                 reg = <0x24e00 0xb0>;
                 fsl,ts-to-buffer;
                 fsl,tmr-prsc = <0x2>;
                 fsl,clock-source-select = <1>;
         };

         enet0: ethernet@24000 {
                 ......
                 ptimer-handle = <&ptp_timer>;
         };
```

2. eTSEC (for PTPd)

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,etsec-ptp' for IEEE 1588 driver

```
Default node:
         ptp_clock@b0e00 {
```

```
                compatible = "fsl,etsec-ptp";
                reg = <0xb0e00 0xb0>;
                interrupts = <68 2 0 0 69 2 0 0>;
                fsl,tclk-period = <10>;
                fsl,tmr-prsc    = <2>;
                fsl,tmr-add     = <0x80000016>;
                fsl,tmr-fiper1  = <0x3b9ac9f6>;
                fsl,tmr-fiper2  = <0x00018696>;
                fsl,max-adj     = <199999999>;
         };
```

3. DPAA

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,fman-rtc'
reg	integer	Required	Register map

```
Default node:
        fman0: fman@400000 {
                #address-cells = <1>;
                #size-cells = <1>;
                cell-index = <0>;
                compatible = "fsl,p4080-fman", "fsl,fman", "simple-bus";

                enet0: ethernet@e0000 {
                        cell-index = <0>;
                        compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
                        reg = <0xe0000 0x1000>;
                        fsl,port-handles = <&fman0_rx0 &fman0_tx0>;
                        tbi-handle = <&tbi0>;
                        phy-handle = <&phy0>;
                        phy-connection-type = "sgmii";
                        ptimer-handle = <&ptp_timer0>;
                };

                ...enet1/2...

                enet3: ethernet@e6000 {
                        cell-index = <3>;
                        compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
                        reg = <0xe6000 0x1000>;
                        fsl,port-handles = <&fman0_rx3 &fman0_tx3>;
                        tbi-handle = <&tbi3>;
                        phy-handle = <&phy3>;
                        phy-connection-type = "sgmii";
                        ptimer-handle = <&ptp_timer0>;
                };

                ptp_timer0: rtc@fe000 {
                        compatible = "fsl,fman-rtc";
                        reg = <0xfe000 0x1000>;
                };
        };


        fman1: fman@500000 {
                #address-cells = <1>;
```

```
                #size-cells = <1>;
                cell-index = <1>;
                compatible = "fsl,p4080-fman", "fsl,fman", "simple-bus";

                enet5: ethernet@e0000 {
                        cell-index = <0>;
                        compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
                        reg = <0xe0000 0x1000>;
                        fsl,port-handles = <&fman1_rx0 &fman1_tx0>;
                        tbi-handle = <&tbi5>;
                        phy-handle = <&phy5>;
                        phy-connection-type = "sgmii";
                        ptimer-handle = <&ptp_timer1>;
                };

                ...enet6/7...

                enet8: ethernet@e6000 {
                        cell-index = <3>;
                        compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
                        reg = <0xe6000 0x1000>;
                        fsl,port-handles = <&fman1_rx3 &fman1_tx3>;
                        tbi-handle = <&tbi8>;
                        phy-handle = <&phy8>;
                        phy-connection-type = "sgmii";
                        ptimer-handle = <&ptp_timer1>;
                };

                ptp_timer1: rtc@fe000 {
                        compatible = "fsl,fman-rtc";
                        reg = <0xfe000 0x1000>;
                };
        };
```

4. DPAA2

   NA.

Verification in Linux and test procedure

Connect two boards through crossover, ex, eth1 to eth1, and connect the other ethernet port on each board to switch or PC. One board runs as master, and the other as slave.

- **Using the IXXAT stack image**

  1. Enable IEEE-1588 support in the gianfar/DPAA driver, rebuild and reload it. You should get the following message during system boot:

     ```
     ...
     IEEE1588: ptp 1588 is initialized.
     ...
     ```

     - On the master side:

       ```
       # ifconfig eth1 192.168.1.100 allmulti up

       # ./ptp -i 0:eth1 -do        //run stack application
       ```

Note: On a DPAA platform, use fm1-gb1 instead of eth1

- On the slave side :

```
# ifconfig eth1 192.168.1.200 allmulti up

#./ptp -i 0:eth1 -do        //run stack application
```

Note: On a DPAA platform, use the desired interface(e.g. fm1-gb1) instead of eth1

2. You should start getting synchronization messages on the slave side.

NOTE

A detailed configuration of the IEEE-1588 stack application is described in a Quick Start Guide provided with the stack application.

- **Using the PTPd stack image**

1. Enable IEEE-1588 support in the gianfar/DPAA/DPAA2 driver, rebuild and reload it. You should get the following message during system boot:

```
...
pps pps0: new PPS source ptp0
...
```

- On the master side:

```
# ifconfig eth0 up

# ifconfig eth0 192.168.1.100

# ./ptpd -i eth0 -MV        //run stack application
```

Note: On a DPAA/DPAA2 platform, use the desired interface(e.g. fm1-gb1) instead of eth0. Also make sure the ptp device name through kernel boot log, and the stack uses default /dev/ptp0. If there is more than 1 ptp clock, please use '-o' arguement to clarify dpaa ptp clock. For example, './ptpd -i eth0 -MV -o /dev/ptp1'.

- On the slave side :

```
# ifconfig eth0 up

# ifconfig eth0 192.168.1.200

# ./ptpd -i eth0 -sV --servo:kp=0.32 --servo:ki=0.05        //run stack application
```

Note: On a DPAA/DPAA2 platform, use the desired interface(e.g. fm1-gb1) instead of eth0. Also make sure the ptp device name through kernel boot log, and the stack uses default /dev/ptp0. If there is more than 1 ptp clock, please use '-o' arguement to clarify dpaa ptp clock. For example, './ptpd -i eth0 -sV -o /dev/ptp1 --servo:kp=0.32 --servo:ki=0.05'.

2. You should start getting synchronization messages on the slave side.

Known Bugs, Limitations, or Technical Issues

- For DPAA, the PTPd stack limits to use only one ptp timer, so only the interfaces on the second FMAN (such as fm2-gb2) are available for PTPd if the platform has two FMANs.

- For eTSEC, the slave PPS signal (available on `TSEC_1588_PULSE_OUT1` pin) is not phase aligned with master PPS signal. This is a known limitation.
- For eTSEC, when running the IEEE-1588 function, the user needs to disable `CONFIG_RX_TX_BUFF_XCHG`. In the default SDK kernel image, the ASF is enabled, so `CONFIG_RX_TX_BUFF_XCHG` is enabled too. You should disable `CONFIG_RX_TX_BUFF_XCHG` and rebuild kernel for 1588 function.
- VLAN feature was disabled in the default kernel configuration. When using 1588 over VLAN in your application or solution, VLAN needs to be enabled in the Linux kernel.
- Packet loss issue could be observed on LS1021ATWR; ethernet interfaces are connected in back-to-back way. The root cause is that the PHY supports IEEE 802.11az EEE mode by default. The low speed traffic will make it go into low power mode. It affects 1588 synchronization performance greatly. Use the workaround below to disable the feature:

```
# ifconfig eth0 up
# ethtool --set-eee eth0 advertise 0
# ifconfig eth0 down
# ifconfig eth0 up
```

7.10 Low Power UART User Guide

7.10.1 Low Power UART User Guide

Description

Low Power Universal asynchronous receiver/transmitter (LPUART) is a high speed and low power uart. Refer to below table for the NXP soc can support LPUART.

SOC	Num of LPUART module
LS1021A	6
LS1043A	6

U-boot ConfigurationCompile time options

Below are major u-boot configuration options related to this feature defined in platform specific config files under include/ configs/ directory.

Option Identifier	Description
CONFIG_LPUART	Enable lpuart support
CONFIG_FSL_LPUART	Enable NXP lpuart support
CONFIG_LPUART_32B_REG	Select 32-bit lpuart register mode

Choosing predefined u-boot board configs:

Please make the defconfig include 'lpuart', like: ls1021atwr_nor_lpuart_defconfig. That's will supoort lpuart.

Runtime options

Env Variable	Env Description	Sub option	Option Description
bootargs	Kernel command line argument passed to kernel	console=ttyLP0,1152000	select LPUART0 as the system console

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree View Options	Description
Device Drivers ---> Character devices --->\n Serial drivers --->\n <*> Freescale lpuart serial port support [*] Console on Freescale lpuart serial port	LPUART driver and enable console support

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_SERIAL_FSL_LPUART	y/m/n	n	LPUART Driver

Device Tree Binding

Below is an example device tree node required by this feature. Note that it may has differences among platforms.

```
lpuart0: serial@2950000 {
                        compatible = "fsl,vf610-lpuart";
                        reg = <0x0 0x2950000 0x0 0x1000>;
                        interrupts = <GIC_SPI 80 IRQ_TYPE_LEVEL_HIGH>;
                        clocks = <&sysclk>;
                        clock-names = "ipg";
                        fsl,lpuart32;
                        status = "okay";
```

Source Files

The following source file are related the this feature in u-boot.

Source File	Description
drivers/serial/serial_lpuart.c	The LPUART driver file

The following source file are related the this feature in Linux kernel.

Source File	Description
drivers/tty/serial/fsl_lpuart.c	The LPUART driver file

Verification in U-Boot

1. Boot up U-Boot from bank0, and update rcw and u-boot for lpuart support to bank4, first copy the rcw and U-Boot binary to the tftp directory.
2. Please refer to the platform depoly document to update the rcw and uboot.
3. After all is updated, run u-boot command to switch to alt bank, then will bring up the new U-Boot to the lpuart console.

```
CPU:   Freescale LayerScape LS1020E, Version: 1.0, (0x87081010)
Clock Configuration:
       CPU0(ARMV7):1000 MHz,
       Bus:300  MHz, DDR:800  MHz (1600 MT/s data rate),
Reset Configuration Word (RCW):
       00000000: 0608000a 00000000 00000000 00000000
       00000010: 60000000 00407900 e0025a00 21046000
       00000020: 00000000 00000000 00000000 08038000
       00000030: 00000000 001b7200 00000000 00000000
I2C:   ready
Board: LS1021ATWR
CPLD:  V2.0
PCBA:  V1.0
VBank: 0
DRAM:  1 GiB
Using SERDES1 Protocol: 48 (0x30)
Flash: 0 Bytes
MMC:   FSL_SDHC: 0
EEPROM: NXID v16777216
PCIe1: Root Complex no link, regs @ 0x3400000
PCIe2: disabled
In:    serial
Out:   serial
Err:   serial
SATA link 0 timeout.
AHCI 0001.0300 1 slots 1 ports ? Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc
Found 0 device(s).
SCSI:  Net:   eTSEC1 is in sgmii mode.
eTSEC2 is in sgmii mode.
eTSEC1, eTSEC2 [PRIME], eTSEC3
=>
```

Verification in Linux

1. After uboot startup, set the command line parameter to pass to the linux kernel including console=ttyLP0,115200 in boootargs. For deploy the ramdisk as rootfs, the bootargs can be set as: "set bootargs root=/dev/ram0 rw console=ttyLP0,115200"

```
=> set bootargs root=/dev/ram0 rw console=ttyLP0,115200

=> dhcp 81000000 <tftpboot dir>/zImage.ls1021a;tftp 88000000 <tftpboot dir>/
initrd.ls1.uboot;tftp 8f000000 <tftpboot dir>/ls1021atwr.dtb;bootz 81000000 88000000 8f000000
```

```
[...]

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Booting Linux on physical CPU 0xf00
Linux version 3.12.0+ (xxx@rock) (gcc version 4.8.3 20131202 (prerelease) (crosstool-NG
linaro-1.13.1-4.8-2013.12 - LinaroGCC 2013.11) ) #664 SMP Tue Jun 24 15:30:45 CST 2014
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=30c73c7d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine: Freescale Layerscape LS1021A, model: LS1021A TWR Board
Memory policy: ECC disabled, Data cache writealloc
PERCPU: Embedded 7 pages/cpu @8901c000 s7936 r8192 d12544 u32768
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 520720
Kernel command line: root=/dev/ram rw console=ttyLP0,115200
PID hash table entries: 4096 (order: 2, 16384 bytes)

[...]

ls1021atwr login: root
root@ls1021atwr:~#
```

2. After the kernel boot up to the console, You can type any shell command in the LPUART TERMINAL.

7.11 PCI Express Interface Controller

7.11.1 PCIE - QorIQ PCIE System (Linux BSP)

Description PCI and PCI Express Controller are integrated with the MPC85xx cpu.

Module Loading

The MPC85xx PCIE host bridge support code is compiled into the kernel. It is not available as a module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Bus support --->`
`[*] PCI support`
`[*] Message Signaled Interrupts (MSI and MSI-X)`
	Enable PCI host bridge and message support

`Bus support --->`
`PCI host controller drivers  --->`
`[*] Freescale Layerscape PCIe controller`
	Enable NXP Layerscape PCIe controller

`Device Drivers --->`
`[*]Network device support --->`
`[*]Ethernet device support --->`
`[*] Intel devices --->`
	Intel PRO/1000 PCI-Express support

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description
<*> Intel (R) PRO/1000 PCI-Express Gigabit Ethernet support	
Device Drivers ---> <*> Serial ATA and Parallel ATA drivers (libata) ---> <*> Silicon Image 3124/3132 SATA support	Enable support for Silicon Image 3124/3132 Serial ATA.

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_PCI	y/n	y	Enable PCI host bridge
CONFIG_PCI_MSI	y/n	y	Message support
CONFIG_PCI_LAYERSCAPE	y/n	y	Enable PCI for Layerscape
CONFIG_E1000E	y/m/n	y	Enable Intel Pro/1000 driver
CONFIG_SATA_SIL	y/m/n	y	Silicon Image SATA support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
arch/powerpc/sysdev/fsl_pci.c	The MPC85XX platform PCIE host bridge support source
drivers/pci/host/pcie-layerscape.c	The Layerscape platform PCIE host bridge support source
drivers/net/ethernet/intel/e1000e/	Intel Pro/1000 driver source code
drivers/ata/sata_sil.c	Silicon Image source code

SATA Card Test Procedure

```
the user can use command
fdisk, mke2fs mount to operate the ide disk.
After kernel boots up, please follow the log to operate:


[root@pX0XX /root]# fdisk -l

Disk /dev/sda: 85.8 GB, 85899345920 bytes
255 heads, 63 sectors/track, 10443 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/sda doesn't contain a valid partition table
```

```
[root@pX0XX /root]# fdisk /dev/sda
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.


The number of cylinders for this disk is set to 10443.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-10443, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-10443, default 10443): 100

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table
sd 0:0:0:0: [sda] 167772160 512-byte hardware sectors (85899 MB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Asking for cache data failed
sd 0:0:0:0: [sda] Assuming drive cache: write through
 sda: sda1


[root@pX0XX /root]# mke2fs /dev/sda1
mke2fs 1.34 (25-Jul-2003)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
100576 inodes, 200804 blocks
10040 blocks (5.00%) reserved for the super user
First data block=0
7 block groups
32768 blocks per group, 32768 fragments per group
14368 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 31 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.


[root@pX0XX /root]# mkdir sda1_test
[root@pX0XX /root]# mount /dev/sda1 sda1_test/
```

```
[root@pX0XX /root]# cp /bin/tar sda1_test/
[root@pX0XX /root]#
```

Ethernet Card Test Procedure

- plug Intel Pro/1000e network card into standard PCI-E slot on a board. After linux bootup, ifconfig ethx ip address and netmask, then do ping testing.

  Tips: x ethernet interface number, an example is as the following for Intel e1000 network card is eth0.

  For example:

  After kernel boot up, bring up with the pci Ethernet card

  ifconfig ethx 192.168.20.100

  ip address should not be conficted with other Ethernet port.

  In Linux window, run ping 192.168.20.101

Known Bugs, Limitations, or Technical Issues

- LSI-SAS card cannot be used on the second PCIe controller when system enables more than one PCIe controller. Use code modification below to workaround this issue:

```
--- a/arch/powerpc/sysdev/fsl_pci.c
+++ b/arch/powerpc/sysdev/fsl_pci.c
@@ -511,7 +511,7 @@ int __init fsl_add_bridge(struct platform_device *pdev, int is_primary)
         printk(KERN_WARNING "Can't get bus-range for %s, assume"
             " bus 0\n", dev->full_name);

-    pci_add_flags(PCI_REASSIGN_ALL_BUS);
+    pci_add_flags(PCI_ENABLE_PROC_DOMAINS);
     hose = pcibios_alloc_controller(dev);
     if (!hose)
         return -ENOMEM;
@@ -846,7 +846,7 @@ int __init mpc83xx_add_bridge(struct device_node *dev)
                " bus 0\n", dev->full_name);
     }

-    pci_add_flags(PCI_REASSIGN_ALL_BUS);
+    pci_add_flags(PCI_ENABLE_PROC_DOMAINS);
     hose = pcibios_alloc_controller(dev);
     if (!hose)
         return -ENOMEM;
```

7.11.2 PCIe Linux Driver

Module Loading

The MPC85xx/Layerscape PCIE host bridge support code is compiled into the kernel. It is not available as a module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Bus support --->`
`[*] PCI support`
`[*] Message Signaled Interrupts (MSI and MSI-X)`
	Enable PCI host bridge and message support

`Bus support --->`
`PCI host controller drivers --->`
`[*] Freescale Layerscape PCIe controller`
	Enable NXP Layerscape PCIe controller

`Device Drivers --->`
`[*]Network device support --->`
`[*]Ethernet device support --->`
`[*] Intel devices --->`
`<*> Intel (R) PRO/1000 PCI-Express Gigabit Ethernet support`
	Intel PRO/1000 PCI-Express support

`Device Drivers --->`
`<*> Serial ATA and Parallel ATA drivers (libata) --->`
`<*> Silicon Image 3124/3132 SATA support`
	Enable support for Silicon Image 3124/3132 Serial ATA.

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_PCI	y/n	y	Enable PCI host bridge
CONFIG_PCI_MSI	y/n	y	Message support
CONFIG_PCI_LAYERSCAPE	y/n	y	Enable PCI for Layerscape
CONFIG_E1000E	y/m/n	y	Enable Intel Pro/1000 driver
CONFIG_SATA_SIL	y/m/n	y	Silicon Image SATA support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
arch/powerpc/sysdev/fsl_pci.c	The MPC85XX platform PCIE host bridge support source
drivers/pci/host/pci-layerscape.c	The Layerscape platform PCIE host bridge support source
drivers/net/ethernet/intel/e1000e/	Intel Pro/1000 driver source code
drivers/ata/sata_sil.c	Silicon Image source code

SATA Card Test Procedure

```
the user can use command
fdisk, mke2fs mount to operate the ide disk.
After kernel boots up, please follow the log to operate:


[root@pX0XX /root]# fdisk -l

Disk /dev/sda: 85.8 GB, 85899345920 bytes
255 heads, 63 sectors/track, 10443 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/sda doesn't contain a valid partition table


[root@pX0XX /root]# fdisk /dev/sda
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.


The number of cylinders for this disk is set to 10443.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-10443, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-10443, default 10443): 100

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table
sd 0:0:0:0: [sda] 167772160 512-byte hardware sectors (85899 MB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Asking for cache data failed
sd 0:0:0:0: [sda] Assuming drive cache: write through
 sda: sda1


[root@pX0XX /root]# mke2fs /dev/sda1
mke2fs 1.34 (25-Jul-2003)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
100576 inodes, 200804 blocks
10040 blocks (5.00%) reserved for the super user
First data block=0
```

```
7 block groups
32768 blocks per group, 32768 fragments per group
14368 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 31 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.


[root@pX0XX /root]# mkdir sda1_test
[root@pX0XX /root]# mount /dev/sda1 sda1_test/
[root@pX0XX /root]# cp /bin/tar sda1_test/
[root@pX0XX /root]#
```

Ethernet Card Test Procedure

- plug Intel Pro/1000e network card into standard PCI-E slot on a board. After linux bootup, ifconfig ethx ip address and netmask, then do ping testing.

  Tips: x ethernet interface number, an example is as the following for Intel e1000 network card is eth0.

  For example:

  After kernel boot up, bring up with the pci Ethernet card

  ifconfig ethx 192.168.20.100

  ip address should not be conficted with other Ethernet port.

  In Linux window, run ping 192.168.20.101

Known Bugs, Limitations, or Technical Issues

- LSI-SAS card cannot be used on the second PCIe controller when system enables more than one PCIe controller. Use code modification below to workaround this issue:

```
--- a/arch/powerpc/sysdev/fsl_pci.c
+++ b/arch/powerpc/sysdev/fsl_pci.c
@@ -511,7 +511,7 @@ int __init fsl_add_bridge(struct platform_device *pdev, int is_primary)
         printk(KERN_WARNING "Can't get bus-range for %s, assume"
             " bus 0\n", dev->full_name);

-    pci_add_flags(PCI_REASSIGN_ALL_BUS);
+    pci_add_flags(PCI_ENABLE_PROC_DOMAINS);
     hose = pcibios_alloc_controller(dev);
     if (!hose)
         return -ENOMEM;
@@ -846,7 +846,7 @@ int __init mpc83xx_add_bridge(struct device_node *dev)
                " bus 0\n", dev->full_name);
     }

-    pci_add_flags(PCI_REASSIGN_ALL_BUS);
+    pci_add_flags(PCI_ENABLE_PROC_DOMAINS);
     hose = pcibios_alloc_controller(dev);
     if (!hose)
         return -ENOMEM;
```

7.11.3 EDAC Driver User Manual

Description

The EDAC kernel module's goal is to detect and report errors that occur within the computer system running under Linux.

Note

Currently the EDAC wasn't supported on P5040/P5020 64bit.

Module Loading

Linux EDAC Driver supports kernel built-in or module.

Kernel Configure Options

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
Device Drivers ---> <*> EDAC (Error Detection And Correction) reporting ---> <*> Main Memory EDAC (Error Detection And Correction) reporting <*> Freescale MPC83xx / MPC85xx	Enables EDAC support for 85xx platform

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_EDAC_MM_EDAC	y/n	y/n	Enables EDAC core support
CONFIG_EDAC_MPC85XX	y/n	y/n	Enables EDAC NXP 85xx support

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,qoriq-memory-controller', 'fsl,p4080-pcie'
reg	integer	Required	Register map

Default node:

```
ddr1: memory-controller@8000 {
    compatible = "fsl,qoriq-memory-controller-v4.4", "fsl,qoriq-memory-controller";
    reg = <0x8000 0x1000>;
    interrupts = <16 2 1 23>;
};

/* controller at 0x200000 */
pci0 {
    compatible = "fsl,p4080-pcie";
```

```
    device_type = "pci";
    #size-cells = <2>;
    #address-cells = <3>;
    bus-range = <0x0 0xff>;
    clock-frequency = <33333333>;
    interrupts = <16 2 1 15>;
 };
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/edac/edac_core.c	Enables EDAC core support
drivers/edac/mpc85xx_edac.c	Enables EDAC NXP 85xx support

Kernel boot message:

```
 ........
 ........
EDAC MC: Ver: 2.1.0
Freescale(R) MPC85xx EDAC driver, (C) 2006 Montavista Software
EDAC MC0: Giving out device to 'MPC85xx_edac' 'mpc85xx_mc_err': DEV mpc85xx_mc_err
MPC85xx_edac acquired irq 16 for MC
MPC85xx_edac MC err registered
EDAC MC1: Giving out device to 'MPC85xx_edac' 'mpc85xx_mc_err': DEV mpc85xx_mc_err
MPC85xx_edac acquired irq 16 for MC
MPC85xx_edac MC err registered
EDAC PCI0: Giving out device to module 'MPC85xx_edac' controller 'mpc85xx_pci_err': DEV
'ffe200000.pcie' (INTERRUPT)
MPC85xx_edac acquired irq 16 for PCI Err
MPC85xx_edac PCI err registered
EDAC PCI1: Giving out device to module 'MPC85xx_edac' controller 'mpc85xx_pci_err': DEV
'ffe201000.pcie' (INTERRUPT)
MPC85xx_edac acquired irq 16 for PCI Err
MPC85xx_edac PCI err registered
EDAC PCI2: Giving out device to module 'MPC85xx_edac' controller 'mpc85xx_pci_err': DEV
'ffe202000.pcie' (INTERRUPT)
MPC85xx_edac acquired irq 16 for PCI Err
MPC85xx_edac PCI err registered
Testing edac driver is start.
PCIE error(s) detected
PCIE ERR_DR register: 0x00020000
PCIE ERR_CAP_STAT register: 0x80000001
PCIE ERR_CAP_R0 register: 0x00000800
PCIE ERR_CAP_R1 register: 0x00000000
PCIE ERR_CAP_R2 register: 0x00000000
PCIE ERR_CAP_R3 register: 0x00000000
 ........
 ........
 ........
p4080 login: root
Password:
```

```
[root@p4080 root]#
```

Test Procedure:

```
[root@p4080 root]#
[root@p4080 root]# cat /proc/interrupts |grep EDAC
 16:          1          0          0          0          0          0          0          0
OpenPIC Level [EDAC] MC err, [EDAC] MC err, [EDAC] PCI err, [EDAC] PCI err, [EDAC] PCI err
[root@p4080 root]#
[root@p4080 root]#
```

Now, see that whether the total number of interrupt 16 of EDAC is zero or less than twenty. If it is that, EDAC driver is OK.

7.11.4 PCIe Advanced Error Reporting User Manual

Description
How to test the PCI Express Advanced Error Reporting (AER) function.

Testing the PCIe AER error recovery code in actual environment is quite difficult because it is hard to trigger real hardware errors. So we use a software tool based error injection to fake various kinds of PCIe errors.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Bus options  --->`
`    [*] PCI Express support`
`    [*]   Root Port Advanced Error Reporting support`
`    <*>     PCIe AER error injector support`
	enable PCI-Express AER and AER-INJECTOR in kernel

Kernel compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_PCIEAER	y/n	y	Enable AER
CONFIG_PCIEAER_INJECT	y/n	n	Enables AER INJECT

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/pci/pcie/aer/*.c	AER driver support

- **Prepare aer-inject test tool**

```
1, Download aer-inject test utility.
```

```
2, Write a test config file
e.g. $ vi aer-cfg
     AER
     DOMAIN 0001
     BUS 1
     DEV 0
     FN 0
     COR_STATUS BAD_TLP
     HEADER_LOG 0 1 2 3

NOTE:
error type can be ["COR_STATUS", "UNCOR_STATUS"]

Corrected error can be:
["BAD_TLP", "RCVR", "BAD_DLLP", "REP_ROLL", "REP_TIMER"]

Uncorrected non-fatal error can be:
["POISON_TLP", "COMP_TIME", "COMP_ABORT", "UNX_COMP", "ECRC", "UNSUP"]

Uncorrected fatal error can be:
 ["TRAIN", "DLP", "FCP", "RX_OVER", "MALF_TLP"]
```

- **Test Steps**

```
1, insert a pcie device in PCI slot of board, ensure the pcie device has AER capability, e.g.
e1000e PCIe NIC network card.

2, In u-boot prompt, add "pcie_ports=native" in bootargs command-line.
=> setenv othbootargs pcie_ports=native

3, boot the kernel and filesystem.
=> tftp 1000000 uImage;tftp 2000000 board.dtb; tftp 3000000 rootfs.ext2.gz.uboot; bootm 1000000
3000000 2000000

4, check AER device and config
# zcat /proc/config.gz|grep -i CONFIG_PCIEAER_INJECT
CONFIG_PCIEAER_INJECT=y

# cat /proc/cmdline
root=/dev/ram rw console=ttyS0,115200 pcie_ports=native
check "pcie_ports=native" has been set.

# ls /dev/aer_inject
Check if the aer injector device is created.

# lspci
00:00.0 Class 0604: 1957:0410
01:00.0 Class 0200: 8086:10d3
e.g. here device "01:00.0" is the PCIe NIC e1000 network card in the test scenario.

5, Download aer-inject and aer-cfg from host to test-board
$ scp aer-inject aer-cfg root@test-board-ip:~

6, ensure the pcie device domain-number/bus-number/device-number/function-number in aer-cfg is
accordant to those in the output of lspci

7, Run aer-inject, corresponding error information will be reported as below and AER will recover
PCIE device according to the type of errors.
# ./aer-inject aer-cfg
```

```
example of error report as below:
pcieport 0000:00:00.0: AER: Corrected error received: id=0100
e1000e 0000:01:00.0: PCIe Bus Error: severity=Corrected, type=Data Link Layer, id=0100(Receiver
ID)
e1000e 0000:01:00.0:   device [8086:10d3] error status/mask=00000040/00002000
e1000e 0000:01:00.0:    [ 6] Bad TLP
root@p1010rdb:~#

8, The pcie device(e1000e PCIE NIC) should still work after AER error recovery.
# ping 192.168.1.1 -c 2 -s 64
PING 192.168.1.1 (192.168.1.1): 64 data bytes
72 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=0.272 ms
72 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.210 ms
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.210/0.241/0.272/0.031 ms
```

Note:

On some legacy platforms with legacy PCI conroller(e.g. some non-DPAA platforms), hardware doesn't support Fatal error type for AER, just support Non-Fatal error.

Generally, DPAA platforms with new PCIE controller can support both Fatal error and Non-Fatal error.

7.11.5 PCI-e Remove and Rescan User Manual

Description
Describes how to remove and rescan a PCI-e device under runtime Linux system.

U-boot Configuration
Use the default configurations.

Kernel Configure Options
Use the default configurations, make sure the configure option is set while doing "make menuconfig" for kernel.

Kernel Configure Tree View Options	Description
Device Drivers ---> [*] Network device support---> [*] Ethernet (1000 Mbit) ---> [*] Intel(R) PRO/1000 PCI-Express Gigabit Ethernet support	This option enables kernel support for Intel PCI-e e1000e network card

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_E1000E	y/n	y	Intel PCI-e e1000e network card driver

Device Tree Binding

Use the default dtb file.

Verification in Linux

Make sure the PCI-e controller which you add the PCI-e e1000e network card to works as RC mode. Use the kernel, dtb and ramdisk rootfs to boot the board.

```
1. Suppose the PCI-e device under /sys/bus/pci/devices/0001\:03\:00.0 is the Intel PCI-e e1000e
network card, recognized as eth0. The
/sys/bus/pci/devices/0001\:02\:00.0 is the bus of network card. Configure an ip and ping another
host which is in the same subnet, make sure the network card works well.

    # ls /sys/bus/pci/devices/0001\:03\:00.0/net
    eth0
    # ifconfig eth0 10.193.20.100
    # ping -I eth0 10.193.20.31

2. Remove the PCI-e network card from system.
    # echo 1 > /sys/bus/pci/devices/0001\:03\:00.0/remove
    e1000e 0001:03:00.0 eth0: removed PHC

3. Check whether the PCI-e network card still exist in system. All should fail.
    # ifconfig eth0
    # ls /sys/bus/pci/devices/0001\:03\:00.0

4. Rescan it from the bus.
    # echo 1 > /sys/bus/pci/devices/0001\:02\:00.0/rescan

5. Check whether the device is rescanned and works well.
    # ls /sys/bus/pci/devices/0001\:03\:00.0
    # ifconfig eth0 10.193.20.100
    # ping -I eth0 10.193.20.31

6. All the commands of step 5 should success.
```

Known Bugs, Limitations, or Technical Issues

The support of PCI-e device remove rescan on powerpc platform is first added in NXP LInux SDK 1.4(kernel version: 3.8.4). If it fail, the PCI-e device will be rescaned, but the driver of the device will fail to loaded.

7.11.6 PCIE EP

Description

SKMM (Secure Key Management Module) is Linux user space implementation for accelerating the encryption/decryption operations on the NXP processors with SEC engine. SKMM consists of two parts of software, the host kernel driver and the application on EP side. The host driver interfaces with OpenSSL or sysfs to pass the encryption/decryption operations to EP side by abstract request and the EP application parses the abstract request to do the encryption/decryption operations, pass the results to host side. The key point here is the private key used is only kept in EP side and is invisible to host side. Also the private key(s) is stored in NOR flash using blob mechanism for protecting the key across system power cycles.

SKMM has two sub use cases:

- - SKMM with PCIe data path. (For now, only SKMM with PCIe data path is supported.)

  For SKMM with PCIe data path, the two boards are connected through the PCIe interface, the requests are sent to EP through PCIe interface.

- - SKMM with Ethernet data path.

  Requests are sent to the EP through the Ethernet port.

Platforms Supported

- P4080DS
- T4240QDS

Compiling SKMM code

Refer to SDK Documentation provided with this release for installing and using Yocto for Image compilation and building.

How to configure and build Host images for PowerPC

1. Change directory to Yocto, execute commands to configure kernel

```
> source ./poky/fsl-setup-poky -m <board-type> -j 12 -t 12

#> bitbake -c menuconfig linux-qoriq-sdk

To P4080DS and T4240QDS:
  Location:
    -> Device Drivers
      -> DMA Engine support
        ->[*] NXP Elo and Elo Plus DMA support (enable the option)
          [ ] Network: TCP receive copy offload (disable the option)
```

2. Execute Commands to build uImage with DMA enabled

```
#> bitbake linux-qoriq-sdk
```

3. Execute Command to generate RFS with kernel modules for host and SKMM application for EP

```
#> bitbake fsl-image-core
```

How to configure and build Host images for X86:

NOTE

When using X86 as Host of SKMM, need a Linux distribution to be installed to X86 PC, suggested using Ubuntu 12.04 64bit.

1. Change directory to Yocto, then extract the SKMM Host source code as following:

```
#> source  ./poky/fsl-setup-poky -m <board-type> -j 12 -t 12
#> bitbake -c patch skmmhost
```

2. The source code will be in `tmp/work/<board -type>-fsl_networking-linux/skmmhost/git-r0/git/`. Copy the source directory to your X86 Host machine for building.
3. Change the directory to the top of the SKMM Host's source code, execute the command:

```
#> make ARCH=x86_64 KERNEL_DIR=/lib/modules/$(uname -r)/build
```

4. The driver modules will be built in the current directory, named "fsl_crypto.ko" and "rsa_test.ko".

How to configure and build EP kernel:

1. Change the directory to Yocto, then execute commands to configure the kernel:

```
#> source ./poky/fsl-setup-poky -m <board-type> -j 12 -t 12
#> bitbake -c menuconfig linux-qoriq-sdk
```

   - For P4080ds: Location:

```
-> Cryptographic API
  ->[*] Hardware crypto devices
    -> < >   NXP CAAM-Multicore driver backend (disable the option)
-> Device Drivers
  -> <*> Userspace I/O drivers
    -> <*>   NXP SEC support(disable the option)
```

   - For T4240qds: Location:

```
-> Cryptographic API
  ->[*] Hardware crypto devices
    -> < >   NXP CAAM-Multicore driver backend (disable the option)
-> Device Drivers
  -> <*> Userspace I/O drivers
    -> <*>   NXP SEC support(disable the option)
  -> <*> VFIO Non-Privileged userspace driver framework(enable the option)
    -> <*>   VFIO support for Fresscale PCI Endpoint devices(enable the option)
```

2. Execute command to build the kernel image

```
#> bitbake linux-qoriq-sdk
```

3. Execute command to generate RFS with SKMM application for EP

```
#> bitbake fsl-image-core
```

Configure physical connection

EP

- For P4080ds, route the PCIe cable between slot #1 and the Host.
- For T4240qds, route the PCIe cable between slot #5 and the Host.

Host

- For P4080ds, route the PCIe cable between slot #3 and the EP
- For X86, route the PCIe cable between X86 PCIe slot with the x4 to x1 connector, and the EP
- For T4240qds, route the PCIe cable between slot #7 and the EP

How to operate EP

Store all the images for PowerPC machine on the tftp server. Configure the board IP and tftp server IP on the u-boot environment using following commands, the third step is to reserve memory for SKMM, it couldn't be ignored.

For P4080ds as EP

1.

```
#> setenv ipaddr <board ip >
```

```
#> setenv serverip <tftp server ip >
#> setenv bootargs "$bootargs usdpaa_mem=256m"
#> save
```

2. Program RCW with `R_PPPNN_0x5/rcw_ep_1500mhz.bin` to flash.
3. Execute following command at u-boot prompt to boot the board

```
#> tftp 1000000 uImage-<bsp>.bin
#> tftp 2000000 fsl-image-core-<bsp>.ext2.gz.uboot
#> tftp c00000 uImage-<bsp>.dtb
#> bootm 1000000 2000000 c00000
```

4. Once the Linux Image boots, enter username=root and password=root to logon.
5. If the SKMM application is being run for the first time, update private key into Nor flash MTD4.

```
root@p4080:flash_eraseall /dev/mtd4
root@p4080:skmm_$(host) /dev/mtd4 update-key ~/.skmm/RSA_priv3
```

NOTE

skmm_$(host) is the application for different Host. For x86 its name is "skmm_x86_64", for PowerPC its is "skmm_powerpc", please check the application used is correct to Host.

6. Run SKMM application, then EP will wait for request offloaded

```
root@p4080:skmm_$(host) /dev/mtd4
```

For T4240qds as EP

1.

```
#> setenv ipaddr <board ip >
#> setenv serverip <tftp server ip >
#> setenv bootargs "$bootargs usdpaa_mem=256m"
#> save
```

2. Program RCW with “RR_XXSSPRPH_1_28_6_12/rcw_1_28_6_12_pexep_1666MHz.bin”

```
#> tftp 1000000 uImage-<bsp>.bin
#> tftp 2000000 fsl-image-core-<bsp>.ext2.gz.uboot
#> tftp c00000 uImage-<bsp>.dtb
#> fdt addr $fdtaddr;fdt mknod /localbus/nor partition@7000000;
#> fdt set /localbus/nor/partition@7000000 reg <0x07000000 0x00100000>;
#> fdt set /localbus/nor/partition@7000000 label "blob";
#> bootm $loadaddr - $fdtaddr;
```

3. Once the Linux Image boots, enter username=root and password=root to logon.
4. If the SKMM application is being run for the first time, update private key into Nor flash MTD0

```
root@t4240:flash_eraseall /dev/mtd0
root@t4240:skmm_$(host) /dev/mtd0 update-key ~/.skmm/RSA_priv3
```

NOTE

skmm_$(host) is the application for different Host. For x86 its name is "skmm_x86_64", for PowerPC its is "skmm_powerpc", please check the application used is correct to Host.

5. Run SKMM application, then EP will wait for request offloaded

```
root@t4240:skmm_$(host) /dev/mtd0
```

How to operate Host

1. boot up PowerPC Host:
2. Configure the board IP and tftp server IP on the u-boot environment using following commands:

```
#> setenv ipaddr <board ip >
#> setenv serverip <tftp server ip >
#> save
```

3. Execute following command at u-boot prompt to boot the board

```
   #> tftp 1000000 uImage-<bsp>.bin
#> tftp 2000000 fsl-image-core-<bsp>.ext2.gz.uboot
#> tftp c00000 uImage-<bsp>.dtb
#> bootm 1000000 2000000 c00000
```

4. Once the Linux Image boots, enter username=root and password=root to logon.
5. Insmod the module to start the test process

```
Root #>:insmod /lib/modules/$(uname -r)/extra/fsl_crypto.ko dev_config_file=/etc/skmm/
skmm_crypto.cfg
```

6. boot up X86 Host
7. There is no specific setup for X86, change directory to c293_skmm_host copied from Yocto, and make sure the modules has been generated.
8. Insert the module to start the test process
   - For PowerPC Host:

```
Root #>:insmod /lib/modules/$(uname -r)/extra/fsl_crypto.ko dev_config_file=/etc/skmm/
skmm_crypto.cfg
```

   - For X86 Host:

```
root #>:insmod fsl_crypto.ko dev_config_file=crypto.cfg
```

How to test

When you complete one of RSA public key test or private key test, you have to reboot both HOST and EP, and reload the fsl_crypto.ko module above, then move on another function test step.

```
For PowerPC as Host
  RSA public key:
  Root #>:insmod /lib/modules/$(uname -r)/extra/rsa_test.ko op=rsa_pub
  RSA private key:
  Root #>:insmod /lib/modules/$(uname -r)/extra/rsa_test.ko op=rsa_priv3
```

```
For x86 as Host
  RSA_public key:
  root #>:insmod rsa_test.ko op=rsa_pub
  RSA private key:
  root #>:insmod rsa_test.ko op=rsa_priv3
```

Because PowerPC haven't supported PCIe hotplug yet, removing fsl_crypto.ko module will cause a Host kernel call trace

Funtion test result

If there was nothing “ERROR” log printed, it means test result was correct, but if the console prints the words like “!!!!! Not matching byte got [xx] original [%0x] at index [xx]”(note: xx is a number of 0 to127), it means test failed.

Performance test

```
RSA pubilc key:
Root #>:echo "RSA_PUB_OP_1K" >/sys/fsl_crypto/fsl_crypto_1/test-i/test_name
RSA private key:
Root #>:echo "RSA_PRV_OP_1K" >/sys/fsl_crypto/fsl_crypto_1/test-i/test_name
```

After console prints the start time and end time, it means the test process is finished, then please use the formula bellow to calculate the performance ops/s (The number of crypto operations in a second):

```
Ops/s = Host cpu Frequency / ((end time – start time) / 5000)
```

For example, to P4080ds, Cpu Frequency is 1.5GHz(1.5 x 109). 5000 is the number of crypto operations (it is the default setting for performance test), it has been hard code, so it couldn’t be modified.

7.12 QUICC Engine HDLC/TDM

7.12.1 QUICC Engine HDLC/TDM User Manual

Linux SDK for QorIQ Processors

Description

HDLC, standing for High-level Data Link Control, is one of the most common protocols of the Layer 2 (Data Link Layer) of the seven-layer OSI model. HDLC uses a zero insertion/deletion process (commonly known as bit stuffing) to ensure that the bit pattern of the delimiter flag does not occur in the fields between flags. The HDLC frame is synchronous and therefore relies on the physical layer for a method of clocking and of synchronizing the transmitter/receiver.

The HDLC/TDM driver is implemented by UCC and TSA(HDLC is upper layer protocol of TDM). It enables UCC1/3 to work in hdlc protocol, connected with X-TDM-DS26522 card to support T1/E1 function. It can work in normal or loopback mode both for tdm controller and phy. connect X-TDM-DS26522 card to TDM Riser slot, it can transmit data and receive data.

U-Boot Configuration

Compile time options

Below are major u-boot configuration options related to this feature defined in platform specific config files under include/ configs/ directory.

Option Identifier	Description

Choosing predefined u-boot modes:

make T1040RDB_config

before doing the actually build

Runtime options

Env Variable	Env Description	Sub option	Option Description
hwconfig	Hardware configuration for u-boot	qe-hdlc	QUICC Engine TDM enabled in DTB
bootargs	Kernel command line argument passed to kernel		

Kernel Configure Options

Tree View

T1040RDB and X-TDM-DS26522 card:

Kernel Configure Tree View Options	Description
Device Drivers ---> SOC (System On Chip) specific Drivers ---> [*] Freescale QUICC Engine (QE) Support [*] Network device support ---> [*] Wan interfaces support ---> <*> Generic HDLC layer <*> Raw HDLC support <*> Freescale QUICC Engine HDLC support <*> TDM Network Drivers ---> <*> SLIC MAXIM DS26522 CARD SUPPORT	Enable the QE TDM driver and X-TDM-DS26522 card driver.

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_QUICC_ENGINE	y/n	n	QUICC Engine enabled
CONFIG_FSL_UCC_TDM	y/n	n	QUICC Engine TDM lib
CONFIG_SLIC_MAXIM	y/m/n	n	Enable x-tdm-ds26522 card support
FSL_UCC_HDLC	y/m/n	n	QUICC Engine driver driver

Device Tree Binding

Below is the definition of the device tree node required by this feature

Property	Type	Status	Description
qe	qe	enable	QUICC Engine node
ucc	hdlc	enable	QE UCC HDLC node.
si	si	si	QE TSA node

Below is an example device tree node required by this feature. Note that it may have differences among platforms.

T1040RDB and X-TDM-DS26522 card:

```
ucc_hdlc: ucc@2000 {
                        compatible = "fsl,ucc_hdlc";
                        rx-clock-name = "clk8";
                        tx-clock-name = "clk9";
                        fsl,rx-sync-clock = "rsync_pin";
                        fsl,tx-sync-clock = "tsync_pin";
                        fsl,tx-timeslot = <0xfffffffe>;
                        fsl,rx-timeslot = <0xfffffffe>;
                        fsl,tdm-framer-type = "e1";
                        fsl,tdm-mode = "normal";
                        fsl,tdm-id = <0>;
                        fsl,siram-entry-id = <0>;
                        fsl,tdm-interface;
                };
                spi@110000 {
                        slic@3 {
                                compatible = "maxim,ds26522";
                                reg = <3>;
                                spi-max-frequency = <2000000>; /* input clock */
                        };
                };
```

Source Files

The following source file are related the this feature in Linux.

T1040RDB and X-TDM-DS26522 card:

Source File	Description
drivers/soc/fsl/qe/qe_tdm.c	QE UCC TDM lib
include/soc/fsl/qe/qe_tdm.h	QE UCC TDM lib head file.
drivers/net/tdm/slic_ds26522.c	X-TDM-DS26522 card driver.
drivers/net/wan/fsl_ucc_hdlc.*	QE HDLC driver
arch/powerpc/boot/dts/fsl/t104xrdb.dtsi	Define the device tree nodes for T1040RDB hdlc
arch/powerpc/boot/dts/fsl/t1040rdb.dts	Define the device tree nodes for T1040RDB ds26522

User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name

Verification in U-boot

N/A

Verification in Linux

1. After u-boot startup,set "qe-hdlc" parameter in hwconfig.
2. After bootup kernel, Kernel boot log for hdlc:

```
hdlc: HDLC support module revision 1.22
```

3. QE HDLC T1/E1 test
   a. Make X-TDM-DS26522 card connected to T1040RDB board Slot.
   b. To test tdm external ports, please plugin tdm t1/e1 loopback cable in the related port.

      The following is HDLC port mapping with X-TDM-DS26522 card:

```
HDLC Port                 X-TDM-DS26522 Port
Port A                         CH1;
Port B                         CH2;
```

   c. HDLC test using E1.

      Use the default dts to test E1 function. Test module can receive ucc_num as parameter. This number should be 1/3 related to the tdm port.

```
root@t1040rdb:~# mount /dev/mmcblk0 /mnt && cp /mnt/sethdlc ./sethdlc && ./sethdlc hdlc0
hdlc && \
 192.168.0.1 hdlc0 && \
ping 192.168.0.2> ifconfig hdlc0 192.168.0.1 up && \
> route add -net 192.168.0.0 netmask 255.255.255.0 gw 192.168.0.1 hdlc0 && \
> ping 192.168.0.2
hdlc0: Carrier detected
PING 192.168.0.2 (192.168.0.2): 56 data bytes
Tx data skb->len:86

Transmitted data:
ff
44
45
00
00
54
cf
71
40
00
40
01
e9
e3
c0
a8
irq ucce:20000
TxBD: 1c00
Received data length:88
while entry times:0
Received data:
ff
44
45
```

```
00
00
54
cf
71
40
00
40
01
e9
e3
c0
a8
skb->protocol:800
irq ucce:80000
```

7.13 Real Time Clock (off-chip)

7.13.1 RTC Driver User Manual (Linux BSP)

Linux SDK for QorIQ Processors

Description
Provides the RTC function.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
Device Drivers-> Real Time Clock-->[*] Set system time from RTC on startup and resume (new) (rtc0) RTC used to set the system time (new) <[*] /sys/class/rtc/rtcN (sysfs) <[*] /proc/driver/rtc (procfs for rtc0) <[*] /dev/rtcN (character devices)	Enable RTC driver

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_RTC_LIB	y/m/n	y	Enable RTC lib
CONFIG_RTC_CLASS	y/m/n	y	Enable generic RTC class support
CONFIG_RTC_HCTOSYS	y/n	y	Set the system time from RTC when startup and resume
CONFIG_RTC_HCTOSYS_DEVICE		"rtc0"	RTC used to set the system time
CONFIG_RTC_INTF_SYSFS	y/m/n	y	Enable RTC to use sysfs

Table continues on the next page...

Table continued from the previous page...

Option	Values	Default Value	Description
CONFIG_RTC_INTF_PROC	y/m/n	y	Use RTC through the proc interface
CONFIG_RTC_INTF_DEV	y/m/n	y	Enable RTC to use /dev interface

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/rtc/	Linux RTC driver

Device Tree Binding

Preferred node name: rtc

Property	Type	Status	Description
compatible	string	Required	Should be "dallas,ds3232"

Default node:

```
    i2c@3000 {
                #address-cells = <1>;
                #size-cells = <0>;
                compatible = "fsl-i2c";
                reg = <0x3000 0x100>;
                interrupts = <43 2>;
                interrupt-parent = <&mpic>;
                dfsrr;
                rtc@68 {
                        compatible = "dallas,ds3232";
                        reg = <0x68>;
                };
              };
```

Verification in Linux

Here is the rtc booting log

```
      ...
      rtc-ds3232 1-0068: rtc core: registered ds3232 as rtc0
      MC object device driver dpaa2_rtc registered
      rtc-ds3232 0-0068: setting system clock to 2000-01-01 00:00:51 UTC (946684851)
      ...

NOTE: Please refer to the related DTS file to enable the RTC driver before building.
      For example, LS2080AQDS board, should enable the below option:
      <*> Dallas/Maxim DS3232
```

Change the RTC time in Linux Kernel

```
        ~ # ls /dev/rtc -l
        lrwxrwxrwx    1 root     root            4 Jan 11 17:55 /dev/rtc -> rtc0
        ~ # date
        Sat Jan  1 00:01:38 UTC 2000
        ~ # hwclock
        Sat Jan  1 00:01:41 2000  0.000000 seconds
        ~ # date 011115502011
        Tue Jan 11 15:50:00 UTC 2011
        ~ # hwclock -w
        ~ # hwclock
        Tue Jan 11 15:50:36 2011  0.000000 seconds
        ~ # date 011115502010
        Mon Jan 11 15:50:00 UTC 2010
        ~ # hwclock -s
        ~ # date
        Tue Jan 11 15:50:49 UTC 2011
        ~ #

        NOTE: Before using the rtc driver, make sure the /dev/rtc node in your file system is
correct. Otherwise, you need to make correct node for /dev/rtc
```

7.14 SATA Controller

7.14.1 NXP Native SATA Driver User Manual

Description
The driver supports NXP native SATA controller.

Module Loading

SATA driver supports either kernel built-in or module.

Kernel Configure Tree View Options	Description
Device Drivers---> <*> Serial ATA and Parallel ATA drivers ---> --- Serial ATA and Parallel ATA drivers <*> AHCI SATA support <*> Freescale QorIQ AHCI SATA support	Enables SATA controller support on ARM-based SoCs

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_SATA_AHCI=y	y/m/n	y	Enables SATA controller
CONFIG_SATA_AHCI_QORIQ=y	y/m/n	y	Enables SATA controller

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/ata/ahci_qoriq.c	Platform AHCI SATA support

Test Procedure

```
Please follow the following steps to use USB in Simics
(1) Boot up the kernel
...
fsl-sata ffe18000.sata: Sata FSL Platform/CSB Driver init
scsi0 : sata_fsl
ata1: SATA max UDMA/133 irq 74
fsl-sata ffe19000.sata: Sata FSL Platform/CSB Driver init
scsi1 : sata_fsl
ata2: SATA max UDMA/133 irq 41
...
(2) The disk will be found by kernel.
...
ata1: Signature Update detected @ 504 msecs
ata2: No Device OR PHYRDY change,Hstatus = 0xa0000000
ata2: SATA link down (SStatus 0 SControl 300)
ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
ata1.00: ATA-8: WDC WD1600AAJS-22WAA0, 58.01D58, max UDMA/133
ata1.00: 312581808 sectors, multi 0: LBA48 NCQ (depth 16/32)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access     ATA      WDC WD1600AAJS-2 58.0 PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 312581808 512-byte logical blocks: (160 GB/149 GiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO
 or FUA
 sda: sda1 sda2 sda3 sda4 < sda5 sda6 >
sd 0:0:0:0: [sda] Attached SCSI disk

(3)play with the disk according to the following log.
[root@ls1046 root]# fdisk -l /dev/sda
Disk /dev/sda: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

    Device Boot      Start         End      Blocks  Id System
/dev/sda1               1         237     1903671  83 Linux
/dev/sda2             238         480     1951897+ 82 Linux swap
/dev/sda3             481        9852    75280590  83 Linux
/dev/sda4            9853       19457    77152162+  f Win95 Ext'd (LBA)
```

```
/dev/sda5            9853       14655    38580066  83 Linux
/dev/sda6           14656       19457    38572033+ 83 Linux
[root@ls1046 root]#
[root@ls1046 root]# mke2fs /dev/sda1
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
65280 inodes, 261048 blocks
13052 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
8160 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 22 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@ls1046 root]#
[root@ls1046 root]# mkdir sata
[root@ls1046 root]# mount /dev/sda1 sata
[root@ls1046 root]# ls sata/
lost+found
[root@ls1046 root]# cp /bin/busybox sata/
[root@ls1046 root]# umount sata/
[root@ls1046 root]# mount /dev/sda1 sata/
[root@ls1046 root]# ls sata/
busybox     lost+found
[root@ls1046 root]# umount sata/
[root@ls1046 root]# mount /dev/sda3 /mnt
[root@ls1046 root]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
rootfs               852019676 794801552  13937948  99% /
/dev/root            852019676 794801552  13937948  99% /
tmpfs                  1036480        52   1036428   1% /dev
shm                    1036480         0   1036480   0% /dev/shm
/dev/sda3             74098076   4033092  66300956   6% /mnt
```

Known Bugs, Limitations, or Technical Issues

- CDROM is not supported due to the silicon limitation

7.15 Serial Peripheral Interface

7.15.1 SPI Flash Over eSPI Controller User Manual

Description

Many NXP's CPU has an eSPI controller which could be connected to different devices, such as SPI flash, ABS ViocePort device, SPI Ethernet Switch etc. The SPI flash is deployed on many NXP development board, so this manual will focus on the test method using SPI flash.

This user manual will take P5020DS board as an example to describes how to select the Configuration options, how to add the DTS file and how to test the SPI flash.

Module Loading

The eSPI controller driver and the SPI flash driver can be built as module or built-in kernel.

Compile-time Configuration Options

Option	Values	Default Value	Description
SPI support --->			
CONFIG_SPI	y/n	y	SPI subsystem support
CONFIG_FSL_ESPI	y/n	y	eSPI controller driver support
Memory Technology Device (MTD) support			
CONFIG_MTD	y/n	y	MTD subsystem support
CONFIG_MTD_PARTITIONS	y/n	y	MTD partitons support
CONFIG_MTD_OF_PARTS	y/n	y	MTD OF partition support
CONFIG_MTD_CHAR	y/n	y	MTD char dev support
CONFIG_MTD_BLOCK	y/n	y	MTD block dev support
Self-contained MTD device drivers --->			
CONFIG_MTD_FSL_M25P80	y/n	y	Spansion SPI flash driver support
CONFIG_M25PXX_USE_FAST_READ	y/n	y	support FAST_READ operation

Device Tree Binding

Preferred node name: spi

Property	Type	Status	Description
Controller's compatible	string	Required	"fsl,mpc8536-espi"
interrupt	integer	Required	platform dependent
Flash's compatible	string	Required	device dependent
spi-max-frequency	integer	Required	device dependent

Default node:

```
eSPI controller DTS node for non-DPAA platform:
"arch/powerpc/boot/dts/fsl/pq3-espi-0.dtsi"
```

```
spi@7000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,mpc8536-espi";
        reg = <0x7000 0x1000>;
        interrupts = <59 0x2 0 0>;
};

eSPI controller DTS node for DPAA platform:
"arch/powerpc/boot/dts/fsl/qoriq-espi-0.dtsi"
spi@110000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,mpc8536-espi";
        reg = <0x110000 0x1000>;
        interrupts = <53 0x2 0 0>;
};

An example to add SPI flash DTS support on P5020DS baord:
"arch/powerpc/boot/dts/fsl/p5020si-post.dtsi"
/include/ "qoriq-espi-0.dtsi"
        spi@110000 {
                fsl,espi-num-chipselects = <4>;
        };

"arch/powerpc/boot/dts/p5020ds.dts"
/include/ "fsl/p5020si-post.dtsi"
                spi@110000 {
                        flash@0 {
                                #address-cells = <1>;
                                #size-cells = <1>;
                                compatible = "spansion,s25sl12801";
                                reg = <0>;
                                spi-max-frequency = <40000000>; /* input clock */
                                partition@u-boot {
                                        label = "u-boot";
                                        reg = <0x00000000 0x00100000>;
                                        read-only;
                                };
                                partition@kernel {
                                        label = "kernel";
                                        reg = <0x00100000 0x00500000>;
                                        read-only;
                                };
                                partition@dtb {
                                        label = "dtb";
                                        reg = <0x00600000 0x00100000>;
                                        read-only;
                                };
                                partition@fs {
                                        label = "file system";
                                        reg = <0x00700000 0x00900000>;
                                };
                        };
                };
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
arch/powerpc/boot/dts/p5020ds.dts arch/powerpc/sysdev/fsl_soc.c	P5020DS board support
drivers/spi/Kconfig	
drivers/spi/Makefile	
drivers/spi/spi_fsl_espi.c	Driver of the eSPI controller
drivers/mtd/devices/Kconfig	
drivers/mtd/devices/Makefile	
drivers/mtd/devices/m25p80.c	Driver of the Spansion SPI flash used on NXP development board

Test Procedure

- boot system until login

```
...

fsl_m25p80 spi32766.0: s25sl128b (16384 Kbytes)
Creating 4 MTD partitions on "SPIFLASH0":
0x000000000000-0x000000100000 : "u-boot"
0x000000100000-0x000000600000 : "kernel"
0x000000600000-0x000000700000 : "dtb"
0x000000700000-0x000001000000 : "file system"

...
 Welcome to NXP Semiconductors Embedded Linux Environment

!!!!! WARNING !!!!!!!

The default password for the root account is: root
please change this password using the 'passwd' command
and then edit this message (/etc/issue) to remove this message

p5020 login:
```

- check valid partition

```
# cat /proc/partitions
```

```
      major minor  #blocks  name

      31        0      32768 mtdblock0
      31        1     262144 mtdblock1
      31        2     131072 mtdblock2
      31        3      65536 mtdblock3
      31        4      16384 mtdblock4
```

- check mtd device

```
[root@p5020 /root]# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 08000000 00020000 "e8000000.flash"
mtd1: 00100000 00010000 "u-boot"
mtd2: 00500000 00010000 "kernel"
mtd3: 00100000 00010000 "dtb"
mtd4: 00900000 00010000 "file system"
```

- mount and read/write file from/to block device

```
[root@p5020 /root]# mke2fs /dev/mtdblock4
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
2304 inodes, 9216 blocks
460 blocks (4.99%) reserved for the super user
First data block=1
Maximum filesystem blocks=9437184
2 block groups
8192 blocks per group, 8192 fragments per group
1152 inodes per group
Superblock backups stored on blocks:
        8193

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 24 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@p5020 /root]# mkdir tmp
/ # mount /dev/mtdblock4 tmp/
[root@p5020 /root]# mkdir tmp
[root@p5020 /root]# mount /dev/mtdblock4 tmp
[root@p5020 /root]# cd tmp
[root@p5020 tmp]# ls
lost+found
[root@p5020 tmp]# echo "hello" > test
[root@p5020 tmp]# cd ..
[root@p5020 /root]# umount tmp
[root@p5020 /root]# mount /dev/mtdblock4 tmp
[root@p5020 /root]# cat tmp/test
hello
[root@p5020 /root]# umount tmp
```

Known Bugs, Limitations, or Technical Issues

- Current driver only works at system clcok = 500MHz. In default RCW, please set SYS_PLL_RAT = 0b00101 (5:1) to make sure system clock = 500MHz.

7.15.2 DSPI Device Driver User Manual

7.15.2.1 DSPI Device Driver User Manual

LS1021-QDS board - U-Boot Configuration

Use QSPI boot mode to boot LS1021A-QDS board.

```
SW1[1:8]+SW2[1]=0100_0101+0
SW6[1:4]=0b'1111
```

QDS X-nor card hardware configuration

```
SW1[1:4]=1000
SW4[1:4]=1111
SW3[1:4]=0001
```

J3： short 1-2, J4： short 1-2. This configuration can make DSPI device and QSPI device work at same time in X-nor card.

Kernel Configure Tree View Options

```
Device Drivers --->
    [*] SPI support --->
         <*> Freescale DSPI controller
```

```
Device Drivers --->
     <*> Memory Technology Device (MTD) support --->
          Self-contained MTD device drivers --->
               <*> Support for AT45xxx DataFlash
```

Compile-time Configuration Options

Config	Values	Default Value	Description
CONFIG_SPI_FSL_DSPI	y/n	y	Enable DSPI module
CONFIG_MTD_DATAFLASH	y/n	y	Enables MTD devices for DSPI flash

Verification in U-Boot LS1021a-qds

Verification in U-Boot:

```
=> sf probe 1:0
SF: Detected AT45DB021D with page size 256 Bytes, erase size 2 KiB, total 256 KiB
=> sf erase 0 10000
SF: 65536 bytes @ 0x0 Erased: OK
=> sf write 82000000 0 1000
SF: 4096 bytes @ 0x0 Written: OK
=> sf read 81100000 0 1000
SF: 4096 bytes @ 0x0 Read: OK
```

```
=> cm.b 81100000 82000000 1000
Total of 4096 byte(s) were the same
```

Verification in Linux:

```
The booting log

......
mtd_dataflash spi0.0: at45db021d (256 KBytes) pagesize 256 bytes (OTP)
6Freescale DSPI master initialized
......

Erase the DSPI flash

~ # mtd_debug erase /dev/mtd0 0x1100000 65536
Erased 65536 bytes from address 0x00000000 in flash


Write the DSPI flash

~ # dd  if=/bin/tempfile.debianutils of=tp bs=4096 count=1
~ # mtd_debug write /dev/mtd0 0 4096 tp
Copied 4096 bytes from tp to address 0x00000000 in flash


Read the DSPI flash

~ # mtd_debug read /dev/mtd0 0 4096 dump_file

Copied 4096 bytes from address 0x00000000 in flash to dump_file

Check Read and Write

Use compare tools(yacto has tools named diff).
~ # diff tp dump_file
~ #
If diff command has no print, the DSPI verification is passed.
```

7.15.3 QuadSPI Driver for TWR-LS1021A User Manual

7.15.3.1 QuadSPI Driver User Manual

U-Boot Configuration

Make sure your boot mode support QSPI.

Use QSPI boot mode to boot an board, please check the board user manual and boot from QSPI. (or some other boot mode decide by your board.)

Kernel Configure Tree View Options

```
Device Drivers --->
     Memory Technology Device (MTD) support
     RAM/ROM/Flash chip drviers --->
```

```
          < > Detect flash chips by Common Flash Interface (CFI) probe
          < > Detect non-CFI AMD/JEDEC-compatible flash chips
          < > Support for RAM chips in bus mapping
          < > Support for ROM chips in bus mapping
          < > Support for absent chips in bus mapping
     Self-contained MTD device drivers --->
          <*> Support most SPI Flash chips (AT26DF, M25P, W25X, ...)
     < >   NAND Device Support  ----
     [*]   the framework for SPI-NOR support
     <*>     Freescale Quad SPI controller
```

```
Device Drivers --->
     [ ] Memory Controller drivers  ----
```

Compile-time Configuration Options

Config	Values	Defualt Value	Description
CONFIG_SPI_FSL_QUADSPI	y/n	y	Enable QSPI module
CONFIG_MTD_SPI_NOR_BASE	y/n	y	Enables the framework for SPI-NOR

Verification in U-Boot

```
=> sf probe 0:0
SF: Detected N25Q128A13 with page size 256 Bytes, erase size 4 KiB, total 16 MiB
=> sf erase 0 100000
SF: 1048576 bytes @ 0x0 Erased: OK
=> sf write 82000000 0 1000
SF: 4096 bytes @ 0x0 Written: OK
=> sf read 81100000 0 1000
SF: 4096 bytes @ 0x0 Read: OK
=> cm.b 81100000 82000000 1000
Total of 4096 byte(s) were the same
```

Verification in Linux:

```
The booting log

......
fsl-quadspi 1550000.quadspi: n25q128a13 (16384 Kbytes)
fsl-quadspi 1550000.quadspi: QuadSPI SPI NOR flash driver
......

Erase the QSPI flash

~ # mtd_debug erase /dev/mtd0 0x1100000 1048576
Erased 1048576 bytes from address 0x00000000 in flash
```

```
Write the QSPI flash

~ # dd if=/bin/tempfile.debianutils of=tp bs=4096 count=1
~ # mtd_debug write /dev/mtd0 0 4096 tp
Copied 4096 bytes from tp to address 0x00000000 in flash


Read the QSPI flash

~ # mtd_debug read /dev/mtd0 0 4096 dump_file

Copied 4096 bytes from address 0x00000000 in flash to dump_file

Check Read and Write

Use compare tools(yacto has tools named diff).
~ # diff tp dump_file
~ #
If diff command has no print log, the QSPI verification is passed.
```

7.16 Time Division Multiplexing (TDM) Interface

7.16.1 TDM User Manual

Description

Time-Division Multiplexing (TDM) is a type of digital or analog multiplexing in which two or more signals or bit streams are transferred apparently simultaneously as sub-channels in one communication channel, but are physically taking turns on the channel. The time domain is divided into several recurrent timeslots of fixed length, one for each sub-channel. A sample byte or data block of sub-channel 1 is transmitted during timeslot 1, sub-channel 2 during timeslot 2, etc. One TDM frame consists of one timeslot per sub-channel. After the last sub-channel the cycle starts all over again with a new frame, starting with the second sample, byte or data block from sub-channel 1, etc.

TDM or Time Division Multiplexing is an essential component to run VoIP applications on NXP Platforms. Its function is to receive and send time division multiplexed voice samples on the physical TDM lines.

This document explains the procedure to test the TDM on FSL MPC85xx platforms.

The test procedure shows the method to run a small TDM demo application which transfers voice from one TDM channel to the other.

The overall TDM software stack and the data flow is depicted below. On the top is a generic TDM framework layer which can ideally integrate with any TDM driver beneath it.

Generally NXP platforms offer two types of TDM interfaces:

1. NXP TDM
2. QE based TDM

This manual specifically talks about NXP TDM

U-Boot Configuration

Compile time options

Please check the platform specific document to check if any specific u-boot configuration is required for TDM feature.

Also please ensure if there is any requirement from pin mux perspective to enable TDM.

Runtime options

Refer to platform specific document for any specific hwconfig or environment variables which may be required for TDM functionality.

Also the FXS ports location will be mentioned in the platform specific document.

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree view options	Description

Device Drivers --->
  <*> TDM support --->
  --- TDM support
  [ ] TDM Core debugging messages (NEW)
  <M> TDM test Module
  TDM Device support --->
  <*> Driver for Freescale TDM controller
  Line Control Devices --->
  <*> Zarlink Slic intialization Module
	Enable TDM Framework
	Enable TDM test as Module
	Enable TDM driver
	Enable SLIC driver

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Value	Default value in BSP	Description
CONFIG_TDM	y/n/m	N	Enable / Disable TDM Framework support
CONFIG_TDM_FSL	y/n/m	N	Enable / Disable TDM driver, depends on TDM framework and CONFIG_FSL_SOC
CONFIG_SLIC_ZARLINK	y/n/m	N	Enable / Disable SLIC driver , depends on TDM driver and TDM framework, and CONFIG_FSL_ESPI
CONFIG_TDM_TEST	y/n/m	N	Enable / disable TDM test module

Device Tree Binding

Below is the definition of the device tree node required by this feature

TDM device dts entries.(as many entries as the number of TDM controllers on the platform)

Property	Type	Status	Description
compatible = "fsl,tdm1.0";	<string>		Should contain "fsl,tdm1.0"
reg = <0x16000 0x200 0x2c000 0x2000>;	<tdm-reg-offset tdm-reg-size dmac-reg-offset dmac-reg-size>		Offset and length of the register set for the NXP TDM and TDM-DMAC
interrupts = <16 8 62 8>;	<tdm-err-intr tdm-err-intr-type dmac-intr dmac-intr-type>		Defines two interrupt specifiers namely interrupt + number and interrupt type for TDM error and TDM DMAC
fsl-max-time-slots = <128>	<u32>		Maximum number of 8-bit time slots in one TDM frame that hardware supports.

SLIC device dts entries (As many entries as the number of SLICs on the platform)

Please note that the below mentioned SLIC entry is for the Legerity SLIC which is connected to the chip through SPI interface.

Property	Type	Status	Description
compatible = "zarlink,le88266";			Should be "zarlink,le88266"
reg = <1>;			Chip select number of the SPI bus SLIC is connected to
spi-max-frequency =<8000000>;			The maximum frequency the SLIC can operate at.

Below is an example device tree node required by this feature. Note that it may have differences among platforms.

```
       tdm@16000 {
               compatible = "fsl,tdm1.0";
               reg = <0x16000 0x200 0x2c000 0x2000>;
               clock-frequency = <0>;
               interrupts = <16 8 62 8>;
               phy-handle = <zarlink1>
               fsl-max-time-slots = <128>
       };


spi@7000 {
    cell-index = <0>;
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,espi";
    reg = <0x7000 0x1000>;
    interrupts = <59 0x2>;
    interrupt-parent = <&mpic>;
    mode = "cpu";
            …….
            ……..
            ……..
               legerity@0{
                       compatible = "zarlink,le88266";
                       reg = <1>;
                       spi-max-frequency = <8000000>;
               };

               legerity@1{
```

```
                        compatible = "zarlink,le88266";
                        reg = <2>;
                        spi-max-frequency = <8000000>;
                };
};
```

Source Files

The following source file are related the this feature in Linux kernel.

Source file	Purpose
include/linux/tdm.h	Header file for TDM framework
drivers/tdm/tdm-core.c	Source file for TDM framework
drivers/tdm/device/tdm_fsl.h	Header file for TDM driver
drivers/tdm/device/tdm_fsl.c	Source file for TDM driver
drivers/tdm/line_ctrl/slic_zarlink.h	Header file for SLIC driver
drivers/tdm/line_ctrl/slic_zarlink.c	Source file for SLIC driver
drivers/tdm/test/tdm_test.c	Source file for TDM test module

Verification in U-Boot

N/A

Verification in Linux

1. Attach two analog phones at the two FXS ports of the board. (Incase there are two SLIC devices there would be 4 FXS ports available).

   NOTE

   Please refer to the platform documentation for specific information on FXS ports.

2. Bring up the platform with the kernel image and dts configured as explained above.

   Look for the below mentioned messages in the kernel boot log.

   This will ensure TDM and SLIC initialization.

```
...
...

EDAC MC: Ver: 2.1.0
fsl_tdm: Freescale TDM Driver Adapter:Init
adapter [fsl_tdm] registered
SLIC: Freescale DEVELOPED ZARLINK SLIC DRIVER
#####################################################
# This driver was created solely by Freescale,     #
# without the assistance, support or intellectual  #
# property of Zarlink Semiconductor. No            #
# maintenance or support will be provided by       #
# Zarlink  Semiconductor regarding this driver.    #
#####################################################
SLIC probed!
SLIC config success
SLIC: product code 1 read is  4
```

```
SLIC: product code 2 read is  b3
SLIC: config read is  ff
SLIC: config read is  8a
DEV reg is 82
DEV reg after is 2
Mask reg before setting is 3f bf
Mask reg after setting is f6 f6
Read Tx Timeslot for CH1 is 0
Read Tx Timeslot for CH2 is 2
Read Rx Timeslot for CH1 is 0
Read Rx Timeslot for CH2 is 2
Operating Fun for channel 1 is 82
Cadence Timer Reg for CH1 before is 7 ff0 0
Cadence Timer Reg for CH1 after is 1 903 20
Switching control for channel 1 is 20
Operating Fun for channel 2 is a0
Cadence Timer Reg for CH2 before is 7 ff0 0
Cadence Timer Reg for CH2 after is 1 903 20
Switching control for channel 2 is 20
SLIC 1 configuration success
TDM_TEST: Test Module for Freescale Platforms with TDM support
TDM_TEST module installed
...
...
```

3. Check /proc/device-tree/soc for tdm and slic nodes.
4. Run cat /proc/interrupts to check for TDM interrupts. Following is an example log details may vary over different platforms.

```
[root@ /root]# insmod tdm_test.ko
TDM_TEST: Test Module for Freescale Platforms with TDM support
TDM Driver(ID=1)is attached with Adapterfsl_tdm(ID = 0) drv_count=1
TDM_TEST module installed
[root@ /root]# cat /proc/interrupts
           CPU0
 20:          0   OpenPIC   Level     fsldma-chan
 21:          0   OpenPIC   Level     fsldma-chan
 22:          0   OpenPIC   Level     fsldma-chan
 23:          0   OpenPIC   Level     fsldma-chan
 28:          0   OpenPIC   Level     ehci_hcd:usb1
 42:         57   OpenPIC   Level     serial
 43:          0   OpenPIC   Level     i2c-mpc, i2c-mpc
 59:          0   OpenPIC   Level     fsl_espi
 62:        993   OpenPIC   Edge      dmac_done_isr
LOC:        698   Local timer interrupts
SPU:          0   Spurious interrupts
CNT:          0   Performance monitoring interrupts
MCE:          0   Machine check exceptions
```

5. To test the TDM functionality Pick up both the phones. Anything spoken on one phone will be heard on the other.

Benchmarking

Voice must be clearly audible and must not break.

Known Bugs, Limitations, or Technical Issues

1. TDM functionality is not supported in 36bit Physical address mode. This is because of hardware limitation on current FSL platforms.
2. TDM_TEST is for demo purpose only and hence runs only for a small duration.

7.17 Universal Serial Bus Interfaces

7.17.1 USB 2.0 Host Driver

7.17.1.1 USB 2.0 Host Driver User Manual

Description
The driver supports USB controller in host mode

Module Loading
The USB Host driver in linux supports either kernel built-in or module driver. U-boot USB driver is always built-in

U-Boot Compile Time Configuration Options

U-Boot Configure Options	Description

`#define CONFIG_HAS_FSL_DR_USB`
 	
`#ifdef CONFIG_HAS_FSL_DR_USB`
`#define CONFIG_USB_EHCI`
 	
`#ifdef CONFIG_USB_EHCI`
`#define CONFIG_CMD_USB`
`#define CONFIG_USB_STORAGE`
`#define CONFIG_USB_EHCI_FSL`
`#define CONFIG_EHCI_HCD_INIT_AFTER_RESET`
`#define CONFIG_CMD_EXT2`
	Enables USB host Dual Role controller support. Defined inside platform config file: include/configs/<platform.h>.

`CONFIG_SYS_FSL_USB_INTERNAL_UTMI_PHY`	Enable internal UTMI Phy support. Required only for SoCs having internal UTMI PHY. Defined inside file: arch/powerpc/include/asm/config_mpc85xx.h
`CONFIG_USB_MAX_CONTROLLER_COUNT`	Tell maximum no. of USB controllers in this SoC. Defined inside file: arch/powerpc/include/asm/config_mpc85xx.h

U-Boot Source Files

The driver source is maintained in the U-boot source in following files

Source File	Description
drivers/usb/host/ehci-fsl.c	EHCI FSL USB host controller driver

Table continues on the next page...

Table continued from the previous page...

Source File	Description
common/cmd_usb.c	Common usb command file
drivers/usb/host/ehci-hcd.c	EHCI USB host controller driver

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers--->`
` USB support --->`
` [*] Support for Host-side USB`
	Enables USB host controller support

`Device Drivers--->`
` USB support --->`
`<*> EHCI HCD (USB 2.0) support`
`-*- Root Hub Transaction Translators`
`[ ] Improved Transaction Translator scheduling (EXPERIMENTAL)`
`[*] Support for Freescale on-chip EHCI USB controller`
`[*] EHCI support for PPC USB controller on OF platform bus`
`<*> OHCI HCD support`
`[*] OHCI support for PPC USB controller on OF platform bus`
`[*] Support big endian HC`
`[*] Support little endian HC`
`[*] OHCI support for PCI-bus USB controllers`
	Enables EHCI Host Controller Driver and transaction translator.

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_USB	y/m/n	y	Enables USB host controller
CONFIG_USB_EHCI_HCD	y/m/n	y	Enables EHCI HCD
CONFIG_USB_EHCI_ROOT_HUB_TT	y/n	y	Enables EHCI to support USB1.1 device

Table continues on the next page...

Table continued from the previous page...

Option	Values	Default Value	Description
CONFIG_USB_OHCI_HCD	y/m/n	y	Enables OHCI HCD

Kernel Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/usb/host/ehci-fsl.c	EHCI USB host controller driver
arch/powerpc/sysdev/fsl_soc.c	Hook between OF tree and platform device
drivers/usb/host/ohci_hcd.c	OHCI USB host controller driver

Device Tree Binding

```
usb@22000 {
 #address-cells = <1>;
 #size-cells = <0>;
 compatible = "fsl-usb2-<controller-type>-v<controller version>",
              "fsl-usb2-<controller-type>";
 reg = <0x22000 0x1000>;
 interrupt-parent = <&mpic>;
 interrupts = <28 0x2>;
 phy_type = "ulpi";                          /* ulpi/utmi/utmi_dual */
 dr_mode = "host"                             /* host, peripheral */
 };
```

NOTE

controller-type: dr(dual-role), mph(multi-port-host) controller-version: 1.6, 2.2, or earlier default mode is always host

Verification in U-Boot

U-boot environment to specify usb phy and usb mode type

```
=> setenv hwconfig 'usb<controller-no>:dr_mode=<mode>,phy_type=<phy_type>;<next usb controller>'

For example:
For socs having single usb controller and ULPI phy
```

```
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=ulpi'

For socs having single usb controller and UTMI phy
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=utmi'

For socs having two usb controllers and ULPI phys only
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=ulpi;usb2:dr_mode=host,phy_type=ulpi'
```

Then use usb start to start the usb device

```
=> usb start

(Re)start USB...

USB: Register 10011 NbrPorts 1

USB EHCI 1.00

scanning bus for devices... 2 USB Device(s) found

 scanning bus for storage devices... 1 Storage Device(s) found

=> usb dev



USB device 0: Vendor: SanDisk Rev: 8.02 Prod: Cruzer Colors+

 Type: Removable Hard Disk

 Capacity: 7663.9 MB = 7.4 GB (15695871 x 512)

=> usb tree

Device Tree:
  1  Hub (480 Mb/s, 0mA)
  |  u-boot EHCI Host Controller
  |
  |+-2  Mass Storage (480 Mb/s, 500mA)
       JetFlash Mass Storage Device 63ZOA56O8TZFZ0AC

=> usb info
1: Hub,  USB Revision 2.0
 - u-boot EHCI Host Controller
 - Class: Hub
 - PacketSize: 64  Configurations: 1
 - Vendor: 0x0000  Product 0x0000 Version 1.0
   Configuration: 1
   - Interfaces: 1 Self Powered 0mA
     Interface: 0
     - Alternate Setting 0, Endpoints: 1
     - Class Hub
     - Endpoint 1 In Interrupt MaxPacket 2048 Interval 255ms

2: Mass Storage,  USB Revision 2.0
 - JetFlash Mass Storage Device 63ZOA56O8TZFZ0AC
```

```
 - Class: (from Interface) Mass Storage
 - PacketSize: 64  Configurations: 1
 - Vendor: 0x8564  Product 0x1000 Version 17.0
   Configuration: 1
   - Interfaces: 1 Bus Powered 500mA
     Interface: 0
     - Alternate Setting 0, Endpoints: 2
     - Class Mass Storage, Transp. SCSI, Bulk only
     - Endpoint 1 In Bulk MaxPacket 512
     - Endpoint 2 Out Bulk MaxPacket 512

=> md 2000000
02000000: 02992004 02060002 08462cc0 84990329    .. ......F,....)
02000010: 00c48e24 82181008 06501810 01a80004    ...$.....P......
02000020: 083d3881 00808270 40a00000 b012a502    .=8....p@.......
02000030: d4000088 28840b45 80028200 40244400    ....(..E....@$D.
02000040: 022b1004 04842482 20610b81 0494d020    .+....$. a.....
02000050: 8012b628 08200100 010c6300 0411b880    ...(. ....c.....
02000060: 42400200 8004a4c8 29802818 904000c0    B@......).(..@..
02000070: 08210200 2040a8c0 448aae00 a0000000    .!.. @..D.......
02000080: 2800c800 04b62080 60199885 02a62324    (..... .`.....#$
02000090: 04870a08 a0008000 18020003 0281a232    ..............2
020000a0: 50414020 4000850b 02044c00 10013018    PA@ @.....L...0.
020000b0: 00208810 000c2280 081805a8 88800010    . ....".........
020000c0: 000c020a 0012b024 01282c02 00808181    .......$.(,.....
020000d0: 00010824 0160b602 81621008 00828082    ...$.`...b......
020000e0: 38d0f028 42010e03 1d242290 02000120    8..(B....$".... 
020000f0: 6c217230 00920800 20200d40 41c08011    l!r0....  .@A...
=> mw 2000000 ffffaaaa 100
=> md 2000000
02000000: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000010: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000020: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000030: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000040: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000050: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000060: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000070: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000080: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000090: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000a0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000b0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000c0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000d0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000e0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000f0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
=> usb write 2000000 0 100

USB write: device 0 block # 0, count 256 ... 256 blocks write: OK
=> md 1000000
01000000: fbf3eae6 2feeffbf dbf7775d ff5bebf7    ..../.....w].[..
01000010: abefefaf 7dbb3e3b bfffb5bb bfb86a7f    ....}.>;......j.
01000020: eff7b68f deaadfff eebf7bff bd7fed1f    ..........{.....
01000030: ffef7deb e7bfbbef dfeff7df 7f3ffcba    ..}..........?..
01000040: ab3bfbfe dfdee69b ffe18fd5 ff3e777f    .;...........>w.
01000050: da7effef bfabff7f f58ef768 9ffffeff    .~.........h....
01000060: cfebf8b0 f1b7dfef e9eefbfe f37bbadb    ............{..
01000070: b7f34ea2 da7efbff dfdff7ff f7effde3    ..N..~..........
01000080: fffbebff fe56ff5d 6ffd7ffd ff87efdf    .....V.]o.......
01000090: b6bfafac ddebfbfb ffacebfd f87bff9f    ............{..
```

```
010000a0: ffebffff ff7e7ff9 aefefd7f 5f7f5ebf    .....~....._.^.
010000b0: 6fe87e7b fabfdbcf d3faefad 6fbb5e7a    o.~{........o.^z
010000c0: f6af86de ffdb7bbf ff5ff6ba bfa4bfdf    ......{..._.....
010000d0: ffbfa87f ffe67fcd efeffb9a 9b7e6a6f    .............~jo
010000e0: fffcf76f efbfeebb ffaceab1 5cfbfffe    ...o........\...
010000f0: edebffde e29fefff deaeafdb f97bdff5    .............{..
=> usb read 1000000 0 100

USB read: device 0 block # 0, count 256 ... 256 blocks read: OK
=> md 1000000
01000000: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000010: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000020: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000030: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000040: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000050: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000060: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000070: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000080: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000090: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000a0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000b0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000c0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000d0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000e0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000f0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
=>
```

Verification in Linux

· Kernel configuration for USB memory stick support

```
 * Device Drivers---> SCSI Device Support---> SCSI Device Support ---> <*> SCSI disk support

* Device Drivers---> SCSI Device Support---> SCSI Device Support ---> <*> SCSI generic support

 * Device Drivers---> USB Support---> [*] USB Mass Storage Support

 (The user can enable it either in kernel mode (set option as True) or as module (set option as
Module)).

* File Systems---> DOS/FAT/NT Filesystems---> <*> MSDOS fs support

* File Systems---> DOS/FAT/NT Filesystems---> <*> VFAT(Window-95)fs support

 * File Systems---> DOS/FAT/NT Filesystems---> VFAT(Window-95)fs support---> Default codepage for
FAT - 437

 * File Systems---> DOS/FAT/NT Filesystems---> VFAT(Window-95)fs support---> Default iocharset
for FAT - "iso8859-1"

 * File Systems---> Partition Types---> [*] Advanced partition selection

 * File Systems---> Partition Types---> [*] PC BIOS (MSDOS partition tables) support

 * File Systems---> Native Language Support---> Base native language support---> (iso8859-1)
Default NLS Option
```

```
* File Systems---> Native Language Support---> Base native language support---> <*> Codepage 437
(United States, Canada)

* File Systems---> Native Language Support---> Base native language support---> <*> NLS ISO
8859-1 (Latin 1; Western European Languages)
```

· plug in memory stick

```
~ # usb 1-1: new high speed USB device using fsl-ehci and address 2

usb 1-1: configuration #1 chosen from 1 choice

scsi6 : SCSI emulation for USB Mass Storage devices

scsi 6:0:0:0: Direct-Access SanDisk Cruzer 7.01 PQ: 0 ANSI: 0 CCS

sd 6:0:0:0: [sda] 3907583 512-byte hardware sectors: (2.00 GB/1.86 GiB)

sd 6:0:0:0: [sda] Write Protect is off

sd 6:0:0:0: [sda] Assuming drive cache: write through

sd 6:0:0:0: [sda] 3907583 512-byte hardware sectors: (2.00 GB/1.86 GiB)

sd 6:0:0:0: [sda] Write Protect is off

sd 6:0:0:0: [sda] Assuming drive cache: write through

 sda: sda1

sd 6:0:0:0: [sda] Attached SCSI removable disk

sd 6:0:0:0: Attached scsi generic sg0 type 0

scsi 6:0:0:1: CD-ROM SanDisk Cruzer 7.01 PQ: 0 ANSI: 0

sr0: scsi3-mmc drive: 48x/48x tray

Uniform CD-ROM driver Revision: 3.20

sr 6:0:0:1: Attached scsi generic sg1 type 5

~ # fdisk -l



Disk /dev/sda: 2000 MB, 2000682496 bytes

64 heads, 63 sectors/track, 969 cylinders

Units = cylinders of 4032 * 512 = 2064384 bytes



 Device Boot Start End Blocks Id System

/dev/sda1 1 969 1953439+ b Win95 FAT32

~ # mount -t vfat /dev/sda1 /mnt/cdrom/
```

```
~ # cd /mnt/cdrom/

/mnt/cdrom # ls

/mnt/cdrom # cp /usr/sbin/wd_keepalive .

/mnt/cdrom # ls

wd_keepalive

/mnt/cdrom # cd ..

/mnt # umount /mnt/cdrom/

================================================================================
To create ext2 file-system on USB flash drive, follow below steps:
# fdisk /dev/sda
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklab
el
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.


Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1011, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1011, default 1011): Using default
 value 1011

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table
sd 2:0:0:0: [sda] Test WP failed, assume Write Enabled
sd 2:0:0:0: [sda] Assuming drive cache: write through
 sda: sda1
[root@p5020 root]# mke2fs /dev/sda1
mke2fs 1.41.4
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
65536 inodes, 262094 blocks
13104 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

```
This filesystem will be automatically checked every 38 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@p5020 /root]# mount /dev/sda1 /mnt
```

· Kernel configuration for USB Human Input Devices support

```
 * Device Drivers--->

        [*] HID Devices --->

        -*- Generic HID support

        <*> USB Human Interface Device (full HID) support

        Input device support --->

        -*- Generic input layer (needed for keyboard, mouse, ...)

        <*> Mouse interface

        [*] Provide legacy /dev/psaux device

        (1024) Horizontal screen resolution

        (768) Vertical screen resolution

        [*] Keyboards --->

        <*> AT keyboard

        [*] Mice --->

        <*> PS/2 mouse

        [*] ALPS PS/2 mouse protocol extension

        [*] Logitech PS/2++ mouse protocol extension

        [*] Synaptics PS/2 mouse protocol extension

        [*] IBM Trackpoint PS/2 mouse protocol extension
```

· plug in USB keyboard

```
~ # usb 1-1: new full speed USB device using fsl-ehci and address 3

usb 1-1: configuration #1 chosen from 1 choice

hub 1-1:1.0: USB hub found

hub 1-1:1.0: 3 ports detected

usb 1-1.1: new full speed USB device using fsl-ehci and address 4

usb 1-1.1: configuration #1 chosen from 1 choice
```

```
input: Dell Dell USB Keyboard Hub as /class/input/input1

generic-usb 0003:413C:2002.0002: input: USB HID v1.10 Keyboard [Dell Dell USB Ke yboard Hub] on
usb-fsl-ehci.0-1.1/input0

input: Dell Dell USB Keyboard Hub as /class/input/input2

generic-usb 0003:413C:2002.0003: input: USB HID v1.10 Device [Dell Dell USB Keyb
```

· plug in USB mouse

```
~ # usb 1-1: new low speed USB device using fsl-ehci and address 2

usb 1-1: configuration #1 chosen from 1 choice

input: HID 413c:3010 as /class/input/input0

generic-usb 0003:413C:3010.0001: input: USB HID v1.00 Mouse [HID 413c:3010] on u
```

Power Management

Following Pwr. Mgmt. features are supported:

1) Sleep

2) Deep Sleep

Pwr. Mgmt. Kernel Configuration Option(s)

Kernel Configure Tree View Options	Description
Kernel options---> [*] Suspend to RAM and standby [*] Hibernation (aka 'suspend to disk') () Default resume partition (NEW)	Enable Power Management feature
Platform support --> CPU Frequency scaling --> [*] CPU Frequency scaling <*> CPU frequency translation statistics Default CPUFreq governor (userspace) --> -*- 'userspace' governor for userspace frequency scaling CPU Frequency drivers --> [*] Support for Freescale MPC85xx CPU freq	Enable the CPU frequency driver

Verification in Linux

Sleep Capability

A system can be put into Suspend state, and can also be Resumed (woken-up) by USB. For this the following needs to be done:

1. Enable USB remote wake-up capability before putting the system into Suspend state

```
~ # echo enabled >/sys/bus/usb/devices/usb1/power/wakeup
```

2. Insert/Remove a USB flash drive into USB port after the system is put into SUSPEND state. This will bring the system out of the SUSPEND state

```
# echo standby > /sys/power/state
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.01 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.01 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
sd 0:0:0:0: [sda] Synchronizing SCSI cache
sd 0:0:0:0: [sda] Stopping disk
PM: suspend of devices complete after 519.108 msecs
PM: late suspend of devices complete after 0.489 msecs
PM: noirq suspend of devices complete after 0.555 msecs
Disabling non-boot CPUs ...

USB Flash drive inserted --->

Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.513 msecs
PM: early resume of devices complete after 0.349 msecs
fsl-lbc ffe05000.localbus: Chip select error: LTESR 0x00080000
/pcie@ffe09000: PCICSRBAR @ 0xfff00000
/pcie@ffe0a000: PCICSRBAR @ 0x0
/pcie@ffe0a000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe0a000: DMA window size is 0x0
/pcie@ffe0b000: PCICSRBAR @ 0x0
/pcie@ffe0b000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe0b000: DMA window size is 0x0
pci 0000:00:00.0: enabling device (0106 -> 0107)
pci 0001:02:00.0: enabling device (0106 -> 0107)
pci 0002:04:00.0: enabling device (0106 -> 0107)
ata2: No Device OR PHYRDY change,Hstatus = 0xa0000000
ata2: SATA link down (SStatus 0 SControl 300)
ata1: Signature Update detected @ 504 msecs
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
ata1.00: configured for UDMA/133
sd 0:0:0:0: [sda] Starting disk
PM: resume of devices complete after 5419.653 msecs
Restarting tasks ... done.
root@p1022ds:~# usb 1-1: new high-speed USB device number 2 using fsl-ehci
scsi2 : usb-storage 1-1:1.0
scsi 2:0:0:0: Direct-Access     SRT      USB              1100 PQ: 0 ANSI: 4
sd 2:0:0:0: Attached scsi generic sg1 type 0
sd 2:0:0:0: [sdb] 15568896 512-byte logical blocks: (7.97 GB/7.42 GiB)
sd 2:0:0:0: [sdb] Write Protect is off
sd 2:0:0:0: [sdb] Mode Sense: 43 00 00 00
sd 2:0:0:0: [sdb] No Caching mode page present
sd 2:0:0:0: [sdb] Assuming drive cache: write through
sd 2:0:0:0: [sdb] No Caching mode page present
```

```
sd 2:0:0:0: [sdb] Assuming drive cache: write through
 sdb: sdb1
sd 2:0:0:0: [sdb] No Caching mode page present
sd 2:0:0:0: [sdb] Assuming drive cache: write through
sd 2:0:0:0: [sdb] Attached SCSI removable disk
FAT-fs (sdb): error, fat_get_cluster: invalid cluster chain (i_pos 0)
FAT-fs (sdb): Filesystem has been set read-only
```

Deep Sleep Capability

USB working across Deep sleep using Timer Interrupt

System is put into deep sleep using the following command :

```
~# echo 30 > /sys/devices/system/mpic/timer_wakeup;echo mem > /sys/power/state
PM: Syncing filesystems ... done.
mmc0: card e624 removed
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
sd 0:0:0:0: [sda] Synchronizing SCSI cache
sd 0:0:0:0: [sda] Stopping disk
PM: suspend of devices complete after 316.061 msecs
PM: late suspend of devices complete after 0.217 msecs
PM: noirq suspend of devices complete after 31.099 msecs
Disabling non-boot CPUs ...
/pcie@ffe240000: PCICSRBAR @ 0xff000000
/pcie@ffe240000: Setup 64-bit PCI DMA window
/pcie@ffe240000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe240000: DMA window size is 0xe0000000
/pcie@ffe250000: PCICSRBAR @ 0xff000000
/pcie@ffe250000: Setup 64-bit PCI DMA window
/pcie@ffe250000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe250000: DMA window size is 0xe0000000
/pcie@ffe260000: PCICSRBAR @ 0x0
/pcie@ffe260000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe260000: DMA window size is 0x0
/pcie@ffe270000: PCICSRBAR @ 0xff000000
/pcie@ffe270000: Setup 64-bit PCI DMA window
/pcie@ffe270000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe270000: DMA window size is 0xe0000000
```

After 30 seconds, system comes out of deep sleep and usb storage device is successfully detected

```
Enabling non-boot CPUs ...
CPU1 is up
CPU2 is up
CPU3 is up
PM: noirq resume of devices complete after 63.844 msecs
PM: early resume of devices complete after 0.166 msecs
caam ffe300000.crypto: Instantiated RNG4 SH0
caam ffe300000.crypto: Instantiated RNG4 SH1
ata2: No Device OR PHYRDY change,Hstatus = 0x80000000
ata2: SATA link down (SStatus 10 SControl 300)
ata1: Signature Update detected @ 504 msecs
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
```

```
ata1.00: configured for UDMA/133
sd 0:0:0:0: [sda] Starting disk
PM: resume of devices complete after 4461.429 msecs
Restarting tasks ... done.
usb 1-1: USB disconnect, device number 3
root@t1040rdb:~# EXT2-fs (sdb1): previous I/O error to superblock detected

EXT2-fs (sdb1): previous I/O error to superblock detected

EXT2-fs (sdb1): previous I/O error to superblock detected

EXT2-fs (sdb1): previous I/O error to superblock detected

mmc0: new high speed SDHC card at address e624
mmcblk0: mmc0:e624 SU08G 7.40 GiB
 mmcblk0: p1
usb 1-1: new high-speed USB device number 4 using fsl-ehci
usb-storage 1-1:1.0: USB Mass Storage device detected
scsi4 : usb-storage 1-1:1.0
scsi 4:0:0:0: Direct-Access     JetFlash Transcend 4GB    8.07 PQ: 0 ANSI: 4
sd 4:0:0:0: Attached scsi generic sg1 type 0
sd 4:0:0:0: [sdb] 7843200 512-byte logical blocks: (4.01 GB/3.73 GiB)
sd 4:0:0:0: [sdb] Write Protect is off
sd 4:0:0:0: [sdb] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
 sdb: sdb1
sd 4:0:0:0: [sdb] Attached SCSI removable disk
```

Deep Sleep using USB wake-up interrupt

This feature is not yet supported.

Known Bugs, Limitations, or Technical Issues

1. Across system Deep Sleep, if a device is already mounted, it may get umounted automatically. Hence, to use it again, the user needs to re-mount the device
2. USB remote wake-up during system Deep-Sleep is not yet supported
3. On some platforms where USB2 controller is muxed with some other IP, USB2 is disabled in default platform configurations inside both U-boot and Linux. For more details, please refer Platform BSP/Board User Manuals
4. Dual-Utmi Phy HW register restoration requirement for System Deep-Sleep feature: Some SoCs have a new utmi phy version called "dual-utmi" phy (for example: T1040, T1042, T1020, T1022, T2080, T2081: rev1.0 and rev1.1). This dual-phy hw registers need to be saved and restored across system Deep-Sleep. Hence, a code is added in u-boot usb driver that identifies all socs having this dual-utmi phy, and adds "dual_utmi" in phy_type property. This is used to determine if all phy registers are to be saved (during system-suspend) and restored (during system-resume). In absence of restoration of dual-phy hw registers, system restore during deep-sleep is going to fail - system hangs and goes into non-recoverable state.

7.17.2 USB Gadget Network Driver User Manual

7.17.2.1 USB 2.0 Gadget Network Driver User Manual

Description

The NXP processor has a High speed Dual-Role(DR) USB controller, which supports device mode

Module Loading

USB device controller driver can be built in kernel or compiled as a module.

Gadget drivers are recommended to be built as modules, because parameters will be passed as module parameter

Table 196. Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
Device Drivers ---> USB support ---> <*> Support for Host-side USB <*> EHCI HCD (USB 2.0) support -*- Root Hub Transaction Translators [ ] Use Xilinx usb host EHCI controller core [*] Support for Freescale PPC on-chip EHCI USB controller	Need to enable CONFIG_USB_FSL_MPH_DR_OF
Device Drivers ---> USB support ---> USB Gadget Support ---> < M > Support for USB Gadgets USB Peripheral Controller (Freescale Highspeed USB DR Peripheral Controller) ---> Freescale Highspeed USB DR Peripheral Controller	Enable NXP USB Device Controller support
Device Drivers ---> USB support ---> USB Gadget Support ---> < M > Support for USB Gadgets USB Gadget Drivers <M> Ethernet Gadget (with CDC Ethernet support) [*]RNDIS support (NEW) [ ]Ethernet Emulation Model (EEM) support (NEW)	Enable USB Gadget support

Table 197. Compile-time Configuration Options

Options	Values	Default	Description
CONFIG_USB_SUPPORT	y/n/m	Ym	Enable USB Support
CONFIG_USB_FSL_MPH_DR_OF	y/n/m	Y	Enable NXP EHCI USB controller
CONFIG_USB_GADGET	y/n/m	m	Enable USB Gadget modules
CONFIG_USB_GADGET_FSL_USB2	y/n/m	m	Enable NXP USB peripheral controller
CONFIG_USB_ETH	y/n/m	m	Enable Ethernet Gadget
CONFIG_USB_ETH_RNDIS	y/n	y	Enable Ethernet Gadget

Source Files

The driver source is maintained in the Linux kernel source tree.

Table 198. Source Files

Source File	Description
drivers/usb/gadget/fsl_usb2_udc.c	NXP USB peripheral controller driver
drivers/usb/host/fsl-mph-dr-of.c	Hook between OF tree and platform device
drivers/usb/gadget/ether.c	Ethernet gadget driver
Drivers/usb/gadget/rndis.[ch]	Microsoft's RNDIS support

Device Tree Entry

```
usb@22000 {

 #address-cells = <1>;
 #size-cells = <0>;
 compatible = "fsl-usb2-<controller-type>-v<controller version>",
              "fsl-usb2-<controller-type>";
 reg = <0x22000 0x1000>;    /* specifies register base addr, soc dependent */
 interrupt-parent = <&mpic>;
 interrupts = <28 0x2>;     /* specifies usb interrupt line, soc dependent */
 phy_type = "ulpi";         /* phy can be ulpi(external)/utmi(internal) */
 dr_mode = "peripheral"     /* this entry specifies usb mode */
 };
```

NOTE

Controller-type: dr(dual-role), mph(multi-port-host) controller-version: 1.6, 2.2, or earlier default mode is always host. It can be either changed to peripheral inside the dts entry like above. In this case re-compilation of dts is required. DR mode can also be changed to peripheral via u-boot command line. This won't require DTS recompilation, and can work with default DTS For USB1 controller.

```
=> setenv hwconfig 'usb1:dr_mode=peripheral,phy_type=<ulpi/utmi>
```

Test Procedure

For board specific changes (required for USB Gadget mode), please refer to the board BSP User Manual.

1. Bring all USB Gadget modules (driver/usb/gadget/*.ko including fs/configfs/configfs.ko) onto the target board.
2. Load device controller driver and test ethernet gadget

   Load FSL gadget driver module **udc-core.ko** & ***fsl_usb2_udc.ko***

   ```
   bash# insmod udc-core.ko
   bash# insmod fsl_usb2_udc.ko
   ```

   Load Ethernet modules

   ```
   bash# insmod configfs.ko
   bash# insmod libcomposite.ko
   bash# insmod u_ether.ko
   bash# insmod u_rndis.ko
   bash# insmod usb_f_rndis.ko
   bash# insmod usb_f_ecm.ko
   bash# insmod usb_f_ecm_subset.ko
   bash# insmod g_ether.ko
   using random self ethernet address
   using random host ethernet address
   usb0: HOST MAC 82:14:b4:63:d1:85
   usb0: MAC 4a:b1:59:3b:b3:bd
   using random self ethernet address
   using random host ethernet address
   g_ether gadget: Ethernet Gadget, version: Memorial Day 2008
   g_ether gadget: g_ether ready

   bash# ifconfig usb0
   usb0      Link encap:Ethernet  HWaddr 5e:0c:de:2f:f9:0f
             BROADCAST MULTICAST  MTU:1500  Metric:1
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
   ```

3. Assign an IP to usb0

   ```
   bash# ifconfig usb0 10.232.1.11 netmask 255.255.255.0 up
   IPv6: ADDRCONF(NETDEV_UP): usb0: link is not ready

   bash# ifconfig usb0
   ```

```
usb0      usb0      Link encap:Ethernet  HWaddr 5e:0c:de:2f:f9:0f
          inet addr:10.232.1.11  Bcast:10.232.1.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

4. Connect a USB cable between target board USB port and the USB port on Windows host machine.

   As soon as USB cable is plugged into a Windows XP host, the following message displays:

```
http://embedded.seattle.intel-research.net/wiki/index.php?
title=Setting_up_USBnet#Install_the_RNDIS_Driver
```

5. Download linux.inf from either of the following, and install the Windows XP RNDIS driver as mentioned in the previous step:

```
http://www.davehylands.com/linux/gumstix/usbnet/linux.inf
http://embedded.seattle.intel-research.net/wiki/files/linux.inf
```

   For Windows 7, driver will automatically install.

6. As soon as driver installed on host, the following message displays on the target:

```
bash# g_ether gadget: high-speed config #2: RNDIS
IPv6: ADDRCONF(NETDEV_CHANGE): usb0: link becomes ready
```

7. Once the RNDIS driver is installed, configured, and loaded, configure the IP address for the new network device.

   For example, assign 10.232.1.10 as IP to the RNDIS device and run ipconfig to verify the network configuration.

8. Now run ping both ways to check the connectivity between RNDIS@Windows and usb0@linux

```
D:\Profiles>ping 10.232.1.11

Pinging 10.232.1.11 with 32 bytes of data:

Reply from 10.232.1.11: bytes=32 time<1ms TTL=64
Reply from 10.232.1.11: bytes=32 time<1ms TTL=64
Reply from 10.232.1.11: bytes=32 time<1ms TTL=64

Ping statistics for 10.232.1.11:
    Packets: Sent = 3, Received = 3, Lost = 0 (0% loss),

Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

bash# ping 10.232.1.10

PING 10.232.1.10 (10.232.1.10): 56 data bytes

64 bytes from 10.232.1.10: seq=0 ttl=128 time=4.352 ms
64 bytes from 10.232.1.10: seq=1 ttl=128 time=1.015 ms
64 bytes from 10.232.1.10: seq=2 ttl=128 time=0.974 ms
64 bytes from 10.232.1.10: seq=3 ttl=128 time=0.935 ms
64 bytes from 10.232.1.10: seq=4 ttl=128 time=1.021 ms

--- 10.232.1.10 ping statistics ---
```

```
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.935/1.659/4.352 ms
```

Known Bugs, Limitations, or Technical Issues

If a board/platform is having multiple USB controller, they cannot be simultaneously used in "gadget/peripheral" mode. Please do not set dr_mode as “peripheral” for both the controllers at the same time.

Supporting Documentation

Linux USB gadget framework - http://www.linux-usb.org/gadget/

Please refer to http://embedded.seattle.intel-research.net/wiki/index.php?title=Setting_up_USBnet for setting up the RNDIS on Windows XP.

Linux USBnet @ http://www.linux-usb.org/usbnet/

7.17.3 USB 3.0 Host/Peripheral Linux Driver User Manual

7.17.3.1 USB 3.0 Host/Peripheral Linux Driver User Manual

Description

The driver supports xHCI SuperSpeed (SS) Dual-Role-Device (DRD) controller

Main features of xHCI controller

- Supports operation as a standalone USB xHCI host controller
- USB dual-role operation and can be configured as host or device
- Super-speed (5 GT/s), High-speed (480 Mbps), full-speed (12 Mbps), and low-speed (1.5 Mbps) operations
- Supports operation as a standalone single port USB
- Supports eight programmable, bidirectional USB endpoints
- OTG (On-The-Go) 2.0 compliant, which includes both device and host capability.

Modes of Operation

- Host Mode: SS/HS/FS/LS
- Device Mode: SS/HS/FS
- OTG: HS/FS/LS

NOTE

Super-speed operation is not supported when OTG is enabled

NOTE

This document explains working of **HS Host and HS Device** in Linux

Module Loading

The default kernel configuration enables support for USB_DWC3 as built-in kernel module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers--->`
 	
` USB support --->`
 	
` [*] Support for Host-side USB`
	Enables USB host controller support

`Device Drivers--->`
 	
` USB support --->`
` <*>     xHCI HCD (USB 3.0) support`
	Enables XHCI Host Controller Driver and transaction translator

`Device Drivers--->`
 	
` USB support --->`
` <*>     USB Mass Storage support`
 	
` [ ]       USB Mass Storage verbose debug`
	Enable support for USB mass storage devices. This is the driver needed for USB flash devices, and memory sticks

`<*> Sound card support  --->`
`<*>   Advanced Linux Sound Architecture   --->`
`   <*>   OSS Mixer API`
`   <*>   OSS PCM (digital audio) API`
`   [*]     OSS PCM (digital audio) API - Include plugin system`
`   [*]   Support old ALSA API`
`   [*]   USB sound devices   --->`
`     <*>   USB Audio/MIDI driver`
	Enables support for USB Audio devices. This driver is needed for USB microphone.

`Device Drivers--->`
 	
` USB support --->`
`  <*>   USB Gadget Support --->`
 	
`    <M>   USB Gadget Drivers`
`    < >     USB functions configurable through configfs`
`    < >     Gadget Zero (DEVELOPMENT)`
`    <M>     Ethernet Gadget (with CDC Ethernet support)`
`    [*]       RNDIS support`
`    [ ]       Ethernet Emulation Model (EEM) support`
`    < >     Network Control Model (NCM) support`
`    < >     Gadget Filesystem`
`    < >     Function Filesystem`
`    <M>     Mass Storage Gadget`
`    < >     Serial Gadget (with CDC ACM and CDC OBEX support)`
	Note: Required only for USB Gadget/ Peripheral Support
	• Enable driver for peripheral/device controller
	• Enable Ethernet Gadget Client driver
	• Enable Mass Storage Client driver

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description
Device Drivers---> <*> DesignWare USB3 DRD Core Support DWC3 Mode Selection (Dual Role mode) --->	Enable XHCI DRD Core Support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_USB	y/m/n	y	Enables USB host controller
CONFIG_USB_XHCI_HCD	y/m/n	y	Enables XHCI HCD
CONFIG_USB_DWC3	y/m/n	y	Enables DWC3 Controller
CONFIG_USB_GADGET	y/m/n	n	Enables USB peripheral device
CONFIG_USB_ETH	y/m/n	n	Enable Ethernet style communication
CONFIG_USB_MASS_STORAGE	m/n	n	Enable USB Mass Storage disk drive
CONFIG_SOUND	y/m/n	y	Enables Sound Card Support
CONFIG_SND	y/m/n	y	Enables ALSA (Advanced Linux Sound Architecture)
CONFIG_SND_MIXER_OSS	y/m/n	y	Enables OSS Mixer API
CONFIG_SND_PCM_OSS	y/m/n	y	Enables OSS PCM (digital audio) API
CONFIG_SND_PCM_OSS_PLUGINS	y/n	y	Enables OSS PCM (digital audio) API - Include plugin system
CONFIG_SND_SUPPORT_OLD_API	y/n	y	Enables old ALSA API
CONFIG_SND_USB	y/n	n	Enables USB sound devices
CONFIG_SND_USB_AUDIO	y/m/n	n	Enables USB Audio/MIDI driver

NOTE: USB Audio configuration options default value is listed for LS1021A platform.

Source Files

The driver source is maintained in the Linux kernel source tree in below files

Table continued from the previous page...

Source File	Description
drivers/usb/host/xhci-*	xhci platform driver
drivers/usb/gadget/mass_storage.c	USB Mass Storage
drivers/usb/gadget/ether.c	Ethernet gadget driver

Device Tree Binding for Host

```
usb@3100000 {
               compatible = "snps,dwc3";
               reg = <0x0 0x3100000 0x0 0x10000>;
               interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
              dr_mode = "host;
            };
```

Device Tree Binding for Peripheral

Note: with multiple USB controller, just one can be peripheral mode at a time.

```
usb@3100000 {
               compatible = "snps,dwc3";
               reg = <0x0 0x3100000 0x0 0x10000>;
               interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
              dr_mode = "peripheral;
               maximum-speed = "super-speed";
          };
```

Host Testing

Following are serial console logs that appear during bootup if dr_mode set to host in device-tree

```
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 1
xhci-hcd xhci-hcd.0.auto: irq 125, io mem 0x03100000
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 2
hub 2-0:1.0: USB hub found
hub 2-0:1.0: 1 port detected
usbcore: registered new interface driver usb-storage
```

Following are serial-console logs after connecting a USB flash drive

```
For High-Speed Device attach
usb 1-1.2: new high-speed USB device number 3 using xhci-hcd
usb-storage 1-1.2:1.0: USB Mass Storage device detected
scsi0 : usb-storage 1-1.2:1.0
scsi 0:0:0:0: Direct-Access     SanDisk  Cruzer           7.01 PQ: 0 ANSI: 0 CCS
sd 0:0:0:0: [sda] 1957887 512-byte logical blocks: (1.00 GB/955 MiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
 sda: sda1
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
```

```
sd 0:0:0:0: [sda] Attached SCSI removable disk

For Super-Speed Device attach
# usb 2-1: new SuperSpeed USB device number 2 using xhci-hcd
usb 2-1: Parent hub missing LPM exit latency info.  Power management will be impacted.
usb-storage 2-1:1.0: USB Mass Storage device detected
scsi0 : usb-storage 2-1:1.0
scsi 0:0:0:0: Direct-Access     SanDisk  Extreme          0001 PQ: 0 ANSI: 6
sd 0:0:0:0: [sda] 31277232 512-byte logical blocks: (16.0 GB/14.9 GiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
 sda:
sd 0:0:0:0: [sda] Attached SCSI removable disk
FAT-fs (sda): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
```

Make filesystem and mount connected USB flash drive using below commands

```
root@freescale /$ fdisk -l

Disk /dev/sda: 16.0 GB, 16013942784 bytes
255 heads, 63 sectors/track, 1946 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks  Id System
/dev/sda1               1        1946    15631213+ 83 Linux
root@freescale /$
root@freescale /$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
shm                     516684         0    516684   0% /dev/shm
rwfs                       512         0       512   0% /mnt/rwfs
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ fdisk /dev/sda

The number of cylinders for this disk is set to 1946.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): d
Selected partition 1

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1946, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1946, default 1946): Using default value 1946

Command (m for help): w
The partition table has been alter sda: sda1
ed!
```

```
Calling ioctl() to re-read partition table
root@freescale /$
root@freescale /$
root@freescale /$ fdisk -l

Disk /dev/sda: 16.0 GB, 16013942784 bytes
255 heads, 63 sectors/track, 1946 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks  Id System
/dev/sda1               1        1946    15631213+ 83 Linux
root@freescale /$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
shm                     516684         0    516684   0% /dev/shm
rwfs                       512         0       512   0% /mnt/rwfs
root@freescale /$ mkdir my_mnt
root@freescale /$
root@freescale /$
root@freescale /$ mkfs.ext2 /dev/sda1
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
977280 inodes, 3907803 blocks
195390 blocks (5%) reserved for the super user
First data block=0
Maximum filesystem blocks=4194304
120 block groups
32768 blocks per group, 32768 fragments per group
8144 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ mount /dev/sda1 my_mnt/
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
shm                     516684         0    516684   0% /dev/shm
rwfs                       512         0       512   0% /mnt/rwfs
/dev/sda1             15385852        20  14604272   0% /my_mnt
root@freescale /$
```

Test by wring/reading data on mount drive

```
root@freescale /$ dd if=/dev/urandom of=/tmp/123 bs=1M count=100
100+0 records in
100+0 records out
104857600 bytes (100.0MB) copied, 54.535026 seconds, 1.8MB/s
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ cp /tmp/123 /my_mnt/.
root@freescale /$ sync
root@freescale /$ ls /my_mnt/
```

```
123         lost+found
root@freescale /$
```

Peripheral testing with Win7 as Host

NOTE

In gadget mode standard USB cables with micro plug should be used.

Below Message will appear during bootup if dr_mode set as peripheral in device-tree

```
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb

usbcore: registered new interface driver usb-storage
```

Make sure "dr_mode" contains "peripheral" string

```
root@freescale /$# cat /proc/device-tree/soc/usb\@3100000/dwc3/dr_mode
peripheral root@freescale /$
```

Move all below modeules to platform

```
fs/configfs/configfs.ko
driver/usb/gadget/libcomposite.ko
driver/usb/gadget/g_mass_storage.ko
driver/usb/gadget/u_rndis.ko
driver/usb/gadget/u_ether.ko
driver/usb/gadget/usb_f_ecm.ko
driver/usb/gadget/usb_f_ecm_subset.ko
driver/usb/gadget/usb_f_rndis.ko
driver/usb/gadget/g_ether.ko
```

Mass Storage Gadget

```
To use ramdisk as a backing store use the following

root@freescale /$ mkdir /mnt/ramdrive
root@freescale /$ mount -t tmpfs tmpfs /mnt/ramdrive -o size=600M
root@freescale /$ dd if=/dev/zero of=/mnt/ramdrive/vfat-file bs=1M count=500
root@freescale /$ mke2fs -F /mnt/ramdrive/vfat-file
root@freescale /$ insmod configfs.ko
root@freescale /$ insmod libcomposite.ko
root@freescale /$ insmod usb_f_mass_storage.ko
root@freescale /$ insmod g_mass_storage.ko file=/mnt/ramdrive/vfat-file stall=n
```

We will get below messages

```
[   39.987594] g_mass_storage gadget: Mass Storage Function, version: 2009/09/11
[   39.994822] g_mass_storage gadget: Number of LUNs=1
[   39.989240]  lun0: LUN: file: /home/backing_file_20mb
[   39.994367] g_mass_storage gadget: Mass Storage Gadget, version: 2009/09/11
[   39.990902] g_mass_storage gadget: userspace failed to provide iSerialNumber
[   39.987547] g_mass_storage gadget: g_mass_storage ready
```

Attached ***USB3.0 only*** gadget cable to host and you will get below message. Now Storage is ready to use.

```
g_mass_storage gadget: super-speed config #1: Linux File-Backed Storage
```

Speaker and Microphone

1. Aplay utility can be used to list the available sound cards e.g. Here Jabra 410 USB speaker is detected as a second sound card and can be addressed as **–D hw:1,0 OR –c1**:

```
[root@freescale ~]$ aplay -l**** List of PLAYBACK Hardware Devices ****
  card 0: FSLVF610TWRBOAR [FSL-VF610-TWR-BOARD], device 0: HiFi sgtl5000-0 [ ]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
  card 1: USB [Jabra SPEAK 410 USB], device 0: USB Audio [USB Audio]  Subdevices: 1/1  Subdevice
#0: subdevice #0
```

2. Sample wav file can be played using the below command:

```
[root@freescale ~]$ aplay -D hw:1,0 LYNC_fsringing.wav
Playing WAVE 'LYNC_fsringing.wav' : Signed 16 bit Little Endian, Rate 48000 Hz, Stereo
```

3. Sample wav file can be recorded using the below command:

```
[root@freescale ~]$ arecord -f S16_LE -t wav -Dhw:1,0 -r 16000 foobar.wav -d 5
Recording WAVE 'foobar.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Mono
```

NOTE: If recorded audio is not played, try to use "-D plughw:1,0" in above command.

4. Audio controls can be checked using the below command, control details and name of the controls can be checked from output of “amixer –c1” as below:

```
[root@freescale ~]$ amixer -c1 controls
numid=3,iface=MIXER,name='PCM Playback Switch'
numid=4,iface=MIXER,name='PCM Playback Volume'
numid=5,iface=MIXER,name='Headset Capture Switch'
numid=6,iface=MIXER,name='Headset Capture Volume'
numid=2,iface=PCM,name='Capture Channel Map'
numid=1,iface=PCM,name='Playback Channel Map'

[root@freescale ~]$ amixer -c1
Simple mixer control 'PCM',0  Capabilities: pvolume pvolume-joined pswitch pswitch-joined penum
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 4 [36%] [-20.00dB] [on]

Simple mixer control 'Headset',0  Capabilities: cvolume cvolume-joined cswitch cswitch-joined
penum
  Capture channels: Mono
  Limits: Capture 0 - 7
  Mono: Capture 5 [71%] [0.00dB] [on]
```

For Example, in above output there are two controls named “PCM” and “Headset” for Speaker and microphone respectively.

Sample Audio controls Usage:

a. mute/unmute

```
[root@freescale ~]$ amixer -c1 set PCM  mute
Simple mixer control 'PCM',0
  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 2 [18%] [-28.00dB] [off]
[root@freescale ~]$ amixer -c1 set PCM  unmute
Simple mixer control 'PCM',0
  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 2 [18%] [-28.00dB] [on]
```

b. volume up/down – Below commands are trying to set volume to 11 and 2 performing volume up and down respectively.

```
root@freescale ~]$ amixer -c1 set PCM 11
Simple mixer control 'PCM',0
Capabilities: pvolume pvolume-joined pswitch pswitch-joined
Playback channels: Mono
Limits: Playback 0 - 11
Mono: Playback 11 [100%] [8.00dB] [on]
[root@freescale ~]$ amixer -c1 set PCM 2
Simple mixer control 'PCM',0
Capabilities: pvolume pvolume-joined pswitch pswitch-joined
Playback channels: Mono
Limits: Playback 0 - 11
Mono: Playback 2 [18%] [-28.00dB] [on]
```

Ethernet Gadget

```
To use Ethernet gadget use the following

root@freescale /$ insmod configfs.ko
root@freescale /$ insmod libcomposite.ko
root@freescale /$ insmod u_ether.ko
root@freescale /$ insmod u_rndis.ko
root@freescale /$ insmod usb_f_ecm.ko
root@freescale /$ insmod usb_f_ecm_subset.ko
root@freescale /$ insmod usb_f_rndis.ko
root@freescale /$ insmod g_ether.ko
```

We will get below messages

```
[   28.692611] using random self ethernet address
[   28.697156] using random host ethernet address
[   28.694271] usb0: HOST MAC 82:96:69:7e:a5:7d
[   28.698928] usb0: MAC 72:00:a5:80:2b:e8
[   28.692586] using random self ethernet address
[   28.697080] using random host ethernet address
[   28.691368] g_ether gadget: Ethernet Gadget, version: Memorial Day 2008
[   28.698028] g_ether gadget: g_ether ready
```

Make sure USB0 ethernet interface is available after this

```
root@freescale /$ ifconfig -a
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:158

can1      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:159

eth0      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:60
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:61
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth2      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:62
          inet addr:10.232.132.212  Bcast:10.232.135.255  Mask:255.255.252.0
          inet6 addr: fe80::2e0:cff:febc:e562/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2311 errors:0 dropped:3 overruns:0 frame:0
          TX packets:66 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:290810 (283.9 KiB)  TX bytes:8976 (8.7 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:100 (100.0 B)  TX bytes:100 (100.0 B)

sit0      Link encap:IPv6-in-IPv4
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

usb0      Link encap:Ethernet  HWaddr 72:00:A5:80:2B:E8
          BROADCAST MULTICAST  MTU:1500  Metric:1
```

```
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Attached the cable with Win7 and Configure RNDS interface in windows under "Control Panel -> Network and Internet -> Network Connections" and set IP Address

Set IP Address in Platform and start Ping

```
root@freescale /$ ifconfig usb0 10.232.1.11
root@freescale /$
root@freescale /$
root@freescale /$ ping usb   10.232.1.10
PING 10.232.1.10 (10.232.1.10): 56 data bytes
64 bytes from 10.232.1.10: seq=0 ttl=128 time=5.294 ms
64 bytes from 10.232.1.10: seq=1 ttl=128 time=6.101 ms
64 bytes from 10.232.1.10: seq=2 ttl=128 time=4.170 ms
64 bytes from 10.232.1.10: seq=3 ttl=128 time=4.233 ms
```

Known Bugs, Limitations, or Technical Issues

- Some issue with Pen drives from Kingston/Transcend. This have noticed some patches floating in open-source for these issues, and also found that open-source USB community trying to fix.
- Linux allow only one peripheral at one time. Please make sure When DWC3 set as Peripheral the other should not be set in same mode.
- Erratum:A-009116 (Frame length of USB3 controller for USB2.0 and USB3.0 operation is incorrect) impacts some socs like LS1020A/LS1021A because of which some USB2.0 and USB3.0 devices may not work properly, and hence, a sw workaround is needed. This sw workaround involves programing following registers of XHCI controller as: GFLADJ[5:0] = 20H and GFLADJ[7] = 1. This is already done via u-boot and linux codebase.

7.18 Watchdog Timers

7.18.1 Watchdog Device Driver (PowerPC)

Module Loading
Watchdog device driver support kernel built-in mode.

U-Boot Configuration

Runtime options

Env Variable	Env Description	Sub option	Option Description
bootargs	Kernel command line argument passed to kernel	setenv othbootargs wdt_period=35	Sets the watchdog timer period timeout

Kernel Configure Options

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
Device Drivers ---> [*] Watchdog Timer Support ---> [*] Disable watchdog shutdown on close [*] PowerPC Book-E Watchdog Timer	PowerPC Book-E Watchdog Timer

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_BOOKE_WDT	y/n	y	PowerPC Book-E Watchdog Timer

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/char/watchdog/booke_wdt.c	PowerPC Book-E Watchdog Timer

User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name
watch	watchdog is a daemon for watchdog feeding	watchdog

Verification in Linux

- Set NFS rootfs. Build a rootfs image which includes watchdog daemon.
- Set boot parameter. On the U-Boot prompt, set following parameter:

  Set nfsargs:

```
setenv bootargs wdt_period=35 root=/dev/nfs rw nfsroot=$serverip:$rootpath ip=$ipaddr:$serverip:
$gatewayip:$netmask:$hostname:$netdev:off
console=$consoledev,$baudrate $othbootargs
```

  Set nfsboot

```
run nfsargs;tftp $loadaddr $bootfile;tftp $fdtaddr $fdtfile;bootm $loadaddr - $fdtaddr
run nfsboot
```

NOTE

`wdt_period` is a watchdog timeout period. Set this parameter with the proper value depending on your board bus frequency.

`wdt_period` is inversely proportional to watchdog expiry time ie. the higher the `wdt_period`, the lower the watchdog expiry time. So if `wdt_period` is increased to high, watchdog will expiry early.

- Check watchdog feeding operation. After the system boots up, check the screen output, if you see:

```
...



PowerPC Book-E Watchdog Timer Enabled (wdt_period=35)



...
```

it means the watchdog module loaded successfully.

- Login in system. Run command `watchdog /dev/watchdog`

```
root@p1020rdb:~# watchdog /dev/watchdog
root@p1020rdb:~# ps -ae | grep watchdog
 3285 ?        00:00:00 watchdog
root@p1020rdb:~#
```

Wait a few minutes. If the system is still alive, watchdog feeding is OK.

- Check watchdog reboot operation. Run command `killall`

```
root@p1020rdb:~# killall -9 watchdog
root@p1020rdb:~#
root@p1020rdb:~# ps -ae | grep watchdog
root@p1020rdb:~#
root@p1020rdb:~# PowerPC Book-E Watchdog Exception
```

Wait for a few seconds. If the system reboots, watchdog reboot operation is OK.

Known Bugs, Limitations, or Technical Issues

On T4240RDB, if you use watchdog, disable the following menu configuration in kernel location. Otherwise these will conflict with each other:

```
|--> Device
Drivers
                |--> Hardware Monitoring support (HWMON [=n])
```

7.18.2 Watchdog Device Driver (LS1012A)

Module Loading

Watchdog device driver support kernel built-in mode.

U-Boot Configuration

Runtime options

Env Variable	Env Description	Sub option	Option Description
bootargs	Kernel command line argument passed to kernel	setenv othbootargs wdt_period=35	Sets the watchdog timer period timeout

Kernel Configure Options

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
Device Drivers ---> [*] Watchdog Timer Support ---> [*] Disable watchdog shutdown on close [*] IMX2+ Watchdog	IMX2 Watchdog Timer

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_IMX2_WDT	y/n	y	IMX2 Watchdog Timer

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/watchdog/imx2_wdt.c	IMX2 Watchdog Timer

User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name
watch	watchdog is a daemon for watchdog feeding	watchdog

Verification in Linux

- Set NFS rootfs. Build a rootfs image which includes watchdog daemon.
- Set boot parameter. On the U-Boot prompt, set following parameter:

Set nfsargs:

```
setenv bootargs wdt_period=35 root=/dev/nfs rw nfsroot=$serverip:$rootpath ip=$ipaddr:$serverip:
$gatewayip:$netmask:$hostname:$netdev:off
console=$consoledev,$baudrate $othbootargs
```

Set nfsboot

```
run nfsargs;tftp $loadaddr $bootfile;tftp $fdtaddr $fdtfile;bootm $loadaddr - $fdtaddr
run nfsboot
```

NOTE

`wdt_period` is a watchdog timeout period. Set this parameter with the proper value depending on your board bus frequency.

`wdt_period` is inversely proportional to watchdog expiry time ie. the higher the `wdt_period`, the lower the watchdog expiry time. So if `wdt_period` is increased to high, watchdog will expiry early.

7.19 Miscellaneous Drivers

7.19.1 SPE Floating Point User Manual

Linux SDK for QorIQ Processors

Description

This document explains the procedure to test the Floating point support of e500 for scalar Single Precision Floating Point (SPFP), vector SPFP and Double Precision Floating Point (DPFP).

U-Boot Configuration

Compile time options

N/A

Runtime options

N/A

Kernel Configure Tree View Options

Tree View

Below are the configure options which needs to be set while doing "make menuconfig" for kernel

Kernel Configure Tree view options	Description
`Processor Support ->` `[*] SPE Support`	Enables SPE so that user processes can execute SPE instructions.

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree view options	Description
Kernel options -> [*] Math Emulation	Enables Math Emulation

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Value	Default value in BSP	Description
CONFIG_85xx	y/n	y	Dependency: MPC85xx silicon family support SPE
CONFIG_SPE	y/n	y	Enables SPE support
CONFIG_MATH_EMULATION	y/n	y	Enables Math Emulation

Device Tree Binding

N/A

Source Files

The driver source is maintained in the Linux kernel source tree.

Source file	Purpose
arch/powerpc/math-emu/math_efp.c	The emulation of SPE Floating point instructions

User Space Application

Please select the following package for testing floating point. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name
testfloat	Testfloat is a program for testing that an implementation of floating point conforms to standards	testfloat

Verification in U-Boot

N/A

Verification in Linux

1. run command "testfloat", you can get the usage

```
[root@P2020RDB /root]# testfloat
testfloat [<option>...] <function>
    <option>:  (* is default)
```

```
-help            --Write this message and exit.

-list            --List all testable functions and exit.

-level <num>     --Testing level <num> (1 or 2).

*  -level 1

-errors <num>    --Stop each function test after <num> errors.

    *  -errors 20

    -errorstop       --Exit after first function with any error.

    -forever         --Test one function repeatedly (implies `-level 2').

    -checkNaNs       --Check for bitwise correctness of NaN results.

    -nearesteven     --Only test rounding to nearest/even.

    -tozero          --Only test rounding to zero.

    -down            --Only test rounding down.

    -up              --Only test rounding up.

    -tininessbefore  --Underflow tininess detected before rounding.

    -tininessafter   --Underflow tininess detected after rounding.

    <function>:

    int32_to_<float>                <float>_add   <float>_eq

    <float>_to_int32                <float>_sub   <float>_le

      <float>_to_int32_round_to_zero   <float>_mul   <float>_lt

      <float>_to_<float>               <float>_div   <float>_eq_signaling

      <float>_round_to_int             <float>_rem   <float>_le_quiet

      <float>_sqrt                                   <float>_lt_quiet

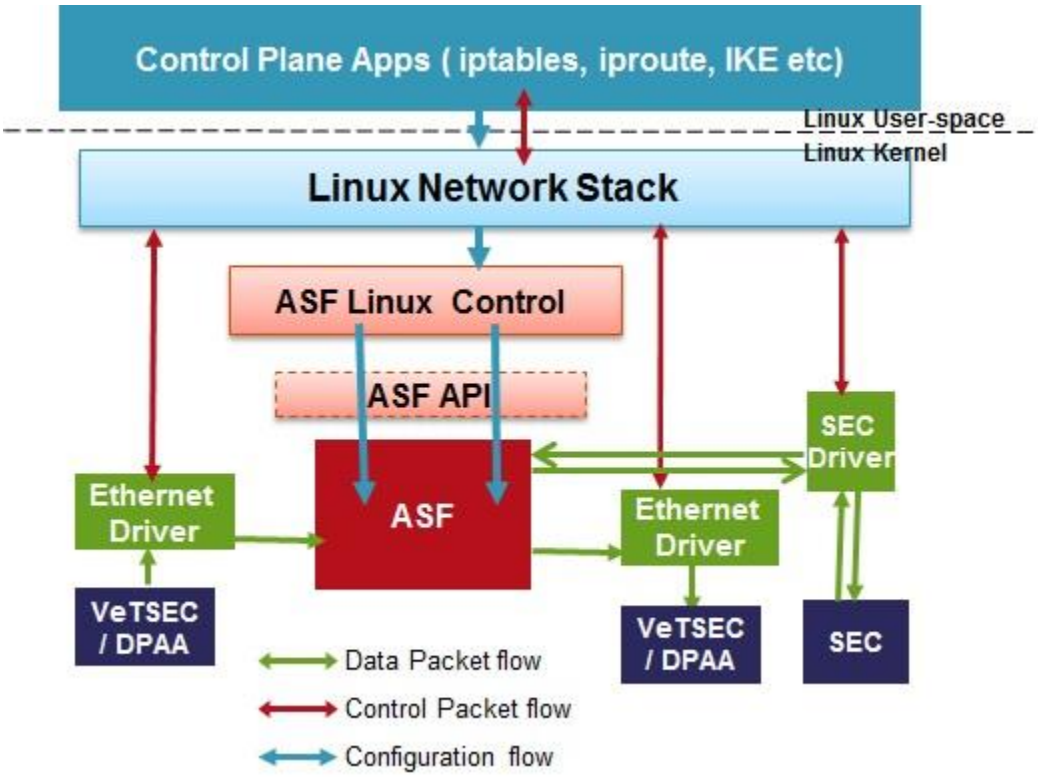
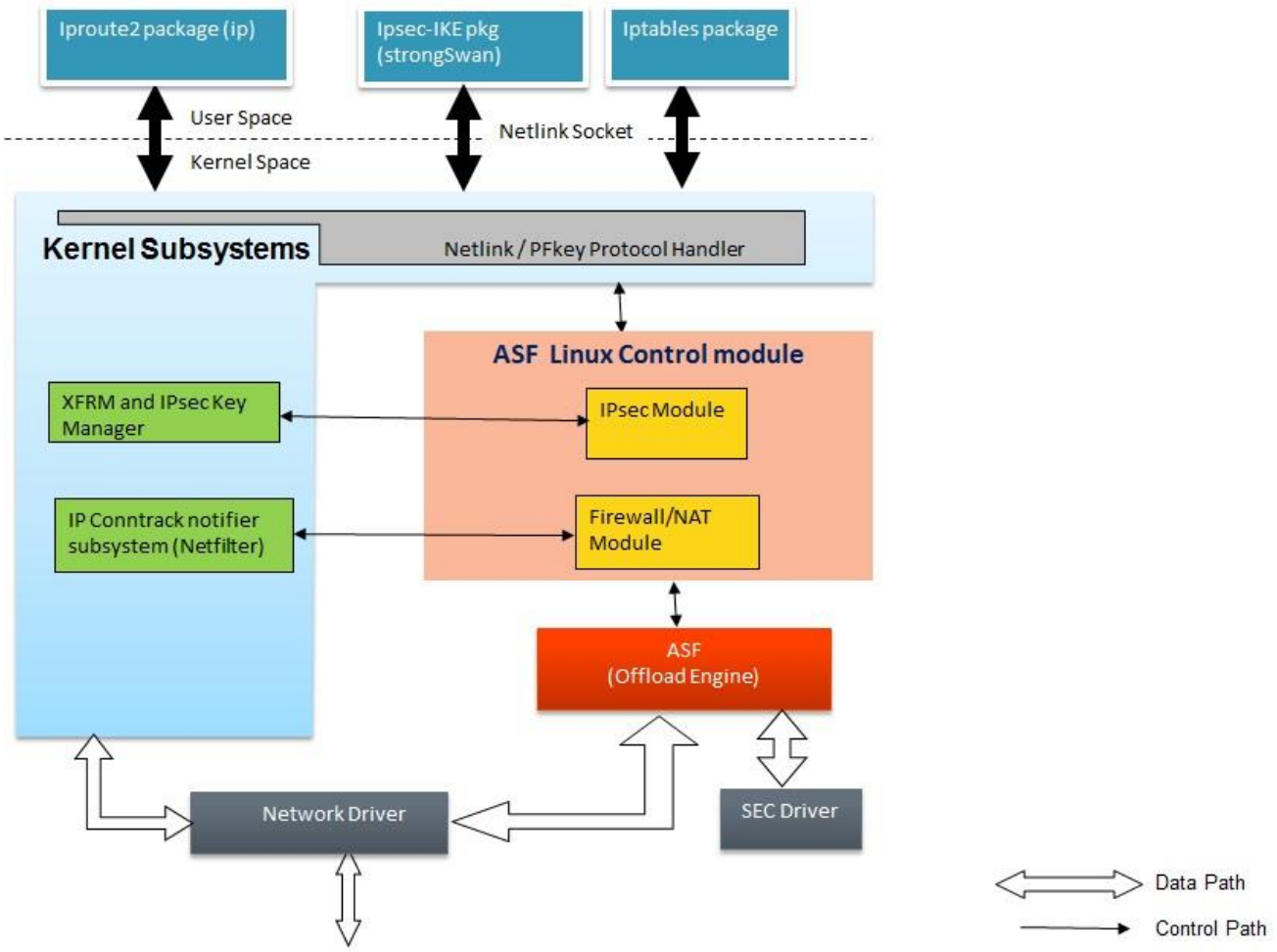
        -all1            --All 1-operand functions.

        -all2            --All 2-operand functions.

        -all             --All functions.

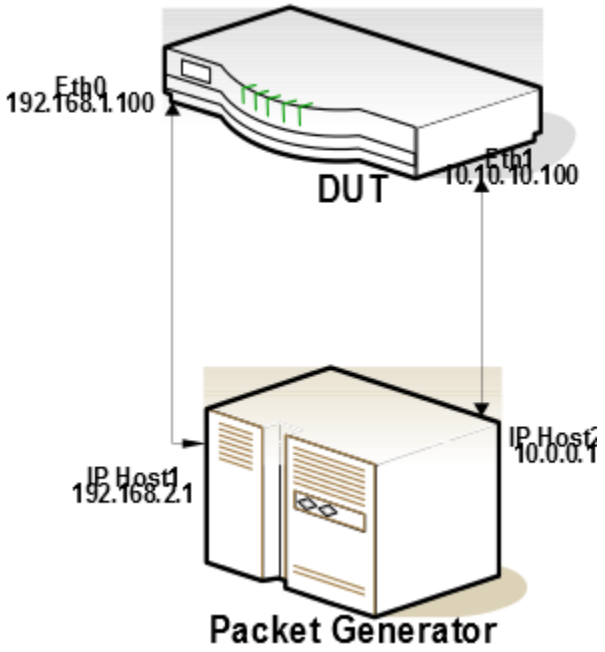
        <float>:

        float32          --Single precision.

        float64          --Double precision.
```

2. run command "testfloat -all" to perform an overall test.

Benchmarking

N/A

Known Bugs, Limitations, or Technical Issues

1. So far, TestFloat has never reported any data error. All the errors it reports concern to exception flags.
2. The exception flags errors TestFloat reports below are harmless.

   Errors that are only produced in either of these cases:

   - option "-tininessaffter" is enable or option "-tininessbefore" is disable.
   - option "-tininessbefore" is enable

   These two options decide checking underflow before or after rounding.

   The standard allows the vendor of the floating-point system to choose whether an underflow condition is detected before or after rounding.

   ```
   float32_le (float le) errors, float32_lt (float lt) errors, float64_le (double le) errors,
   float64_lt (double lt) errors
   ```

   IEEE 754 defines that Arithmetic operations upon NaNs other than SNaNs never signal INVALID.

   ```
   float32_to_int32_round_to_zero (float to int round zero), float64_to_int32_round_to_zero (double to
   int round zero)
   ```

   IEEE 754 defines that conversion from floating-point to other formats can be INVALID too, if their limits are violated, even if no NaN can be delivered.

Chapter 8
Additional Linux Use Cases

8.1 Application Specific Fastpath (ASF) User Manual

8.1.1 Description

8.1.1.1 Overview

ASF is a 'C' programmable software based fast-path in Linux Kernel, supporting data path acceleration seamlessly with Linux Kernel Networking Stack on the NXP family of processors.

ASF implementation is compiled as Kernel dynamic loadable modules that can be plugged in to Linux stack at run time. Once plugged into Linux stack, packets are snooped from the Linux network device driver (currently Ethernet) to the ASF data path module.

ASF targets the most common data applications.



Figure 180. ASF high level view

8.1.1.2 Key Target Applications

ASF is used for accelerating throughput for

1. IPv4 Firewall and NAT/NAPT Forwarding
2. IPv6 Firewall Forwarding

3. IPv4 & IPv6 IPSec (ESP & AH) Forwarding

8.1.1.3 ASF Diagram

ASF integrates with the various network control modules of Linux to extract information for data processing. Figure below identifies how the ASF module fits in the overall system in typical network processor software architecture.



Figure 181. ASF Software Architecture

8.1.1.4 Features

As a part of delivery user will receive ASF as part of Yocto source release.

Following features are supported in this release

Table 199. Features

Feature	Detailed Description
IPv4 Firewall Forwarding	IPv4 packet forwarding for UDP and TCP packets. Firewall support only based on 5-tuple information.

Table continues on the next page...

Table 199. Features (continued)

Feature	Detailed Description
IPv6 Firewall Forwarding	IPv6 packet forwarding for UDP and TCP packets. Firewall support only based on 5-tuple information.
NAT	NAT for TCP and UDP packets
IPsec	ESP & AH in Tunnel mode
Fragmentation	Fragmentation and Re-assembly for NAT, IPsec
IKEv1 and IKEv2 Support	Dynamic key Exchange for IPsec in forwarding
VLAN	Forwarding between VLAN and non-VLAN interfaces.

8.1.1.5 Specifications

Compliance with Standards

1. RFC 791 - Internet Protocol
2. RFC2460 - Internet Protocol version 6
3. RFC 793 - Transmission Control Protocol
4. RFC 768 - UDP
5. RFC2406 - IPsec ESP
6. RFC4302 - IPsec AH
7. RFC2409 - Internet Key Exchange
8. RFC3022 - Traditional IP NAT

Limitations

1. ASF firewall/NAT support only for TCP and UDP packets.
2. Firewalling supported based on 5-tuple information i.e. Source IP, Destination IP, Source Port, Destination Port and Protocol.
3. IPsec ESP & AH supported in Tunnel mode only.
4. ASF impacts performance for data traffic not handled by ASF.
5. IEEE 1588 PTP feature will not work when “RX and TX ring buffer exchange for Routed packet” is enabled. This is valid only for non-DPAA boards.
6. To test the Jumbo frames please enable the CONFIG_FSL_DPAA_ETH_JUMBO_FRAME from kernel configuration and also increase the L2 maximum frame size to 9600 (either set CONFIG_FSL_FM_MAX_FRAME_SIZE from menuconfig or set the fsl_fm_max_frm bootarg). This option is available on DPAA based platforms.
7. ASF does not support handling of NR_FRAGS and system behavior is not known.
8. ASF is not integrated with Linux QoS/TC framework.
9. ASF only work for eTSEC or DPAA based Ethernet ports.

Platforms Supported

- T1040D4RDB

- P4080DS
- LS1021ATWR

8.1.2 Execution

8.1.2.1 Getting started

8.1.2.1.1 Installing the SDK

Refer SDK Documentation provided with this release for installing and using Yocto for image compilation and building.

8.1.2.1.2 Compiling Linux

Execute below command for a particular board

```
#> source ./poky/fsl-setup-poky -m <board-type> -j 12 -t 12
```

NOTE

Board type can be chosen for 32/64 bit architecture. Yocto can build images for both 32/64 bit architecture systems.

Supported QorIQ(powerpc) machines: p4080ds p4080ds-64b t1040d4rdb-64b t1040d4rdb ls1021atwr

```
#>bitbake -c menuconfig virtual/kernel

From “make menuconfig”
Select:
Application Specific FastPath ( Disabled by Default )
  |   Location:
  |     -> Device Drivers

Please change CONFIG_FSL_DPAA_ETH_MAX_BUF_COUNT to 512. Location of this variable is
Device Drivers -> Ethernet Driver Support -> Freescale Devices -> DPAA Ethernet ->
FSL_DPAA_ETH_MAX_BUF_COUNT
```

Execute Command to generate RFS with ASF .kos

```
#>bitbake fsl-image-core
```

ASF .ko’s will be copied to “/usr/driver/asf/min/” and “/usr/driver/asf/full/”. IPsec and FMC (for DPAA) scripts will be copied to /usr/driver/asf/scripts. In case a modified asf module is required, it can be TFTP to the board dynamically. Sections below describe how to use these dynamic loadable modules.

Boot Arguments

The following bootargs need to be configured at the U-Boot prompt.

DPAA

```
#setenv bootargs "root=/dev/ram rw rootdelay=5 console=ttyS0,115200 ramdisk_size=70000000"
```

Non-DPAA

```
#setenv bootargs "root=/dev/ram rw rootdelay=5 console=ttyS0,115200 ramdisk_size=70000000 nousb"
```

8.1.2.1.3 Booting the board

Once the build completes you will get the following directory/image files:

- **build_<bsp>_release** - project folder.
- **tmp/deploy/images/** - subdirectory with images.
- **uImage-<bsp>.bin** - kernel image that can be loaded with U-Boot
- **fsl-image-core-<bsp>.ext2.gz.u-boot** - ramdisk image that can be loaded with U-Boot.
- **u-boot-<bsp>.bin** - U-Boot binary image that can be programmed into board Flash.
- **uImage-<bsp>.dtb** - device tree binary (dtb) for kernel boot up.

The ASF related kernel modules will be placed in the /lib/modules/${KERNEL_VERSION}/asf directory in the root file system.

Once the images have been built, load the images on board:

- Store fsl-image-core-<bsp>.ext2.gz.u-boot, uImage-<bsp>.bin and uImage-<bsp>.dtb on the tftp server
- Connect board's eth0 interface to tftp server.
- Configure the board IP and tftp server IP on the u-boot environment using following commands

```
      #> setenv ipaddr <board ip >
      #> setenv serverip <tftp server ip >
      #> save
```

- Execute the following commands at u-boot prompt to boot the board with ASF specific pre-compiled images

```
For 32b images
      #> tftp 1000000 uImage-<bsp>.bin
      #> tftp 2000000 fsl-image-core-<bsp>.ext2.gz.uboot
      #> tftp c00000 uImage-<bsp>.dtb
      #> bootm 1000000 2000000 c00000

For 64b images
      #> tftp 10000000 uImage-<bsp>.bin
      #> tftp 20000000 fsl-image-core-<bsp>.ext2.gz.uboot
      #> tftp 30000000 uImage-<bsp>.dtb
      #> bootm 10000000 20000000 30000000
```

- Once the Linux Image boots, enter username=**root** and password=**root** to logon.

Next sections describes the setup and board configuration example to run ASF in the mode supported.

8.1.2.2 Firewall Forwarding

8.1.2.2.1 Firewall Forward Hardware Setup



Figure 182. IPv4 Diagram

•Note : For DPAA interface name convention is different for example it will be of the kind “fm1-gb1”.

8.1.2.2.2 Software instructions

Configuration at the DUT

- Configure Ethernet address for eth0 and eth1

```
* IPv4
#>ifconfig eth0 192.168.1.100 netmask 255.255.0.0 up
#>ifconfig eth1 10.10.10.100 netmask 255.0.0.0 up

* IPv6
#> ip -6 addr add 2001::1/64 dev eth0
#> ip -6 addr add 2000::1/64 dev eth1
```

NOTE

For DPAA interface name convention is different for example it will be of the kind “fm1-gb1

- Enable routing at the DUT

```
#> echo 1 > /proc/sys/net/ipv4/ip_forward
#> echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

- Load ASF modules

```
#> insmod asf.ko
#> insmod asfctrl.ko
```

- Increase number of conntrack entries in Linux for measuring performance for 128K flows

```
#> echo 140000 > /proc/sys/net/nf_conntrack_max
```

- Change the default UDP timeout to 9000 seconds. (for performance testing)

```
#> echo 9000 > /sys/asfctrl/ffp/asfctrl_ffp_udp_tmout
#> echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout
#> echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout_stream
```

Execute commands for non-DPAA platforms only

- Set the coalescing parameters

```
#> ethtool -C eth0 rx-frames 12 rx-usecs 32
#> ethtool -C eth1 rx-frames 12 rx-usecs 32
```

- Disable PTP

```
#> echo 0 > /sys/class/net/eth0/ptp_1588
#> echo 0 > /sys/class/net/eth1/ptp_1588
```

- Set the coalescing parameters for LS1021A TWR

```
#> ethtool -C eth0 rx-frames 22 rx-usecs 32
#> ethtool -C eth1 rx-frames 22 rx-usecs 32
```

- Set affinities on LS1021ATWR boards:

```
#> echo 2 > /proc/irq/182/smp_affinity ( eth1 TX interrupt )
#> echo 2 > /proc/irq/177/smp_affinity ( eth0 RX interrupt )
#> echo 1 > /proc/irq/176/smp_affinity ( eth0 TX interrupt )
#> echo 1 > /proc/irq/184/smp_affinity ( eth1 RX interrupt )
```

Execute commands for DPAA platforms only

- Execute fmc command for packet distribution

```
#> fmc –s Soft_FragParser.xml –p <policy file> –c <configuration file> –a

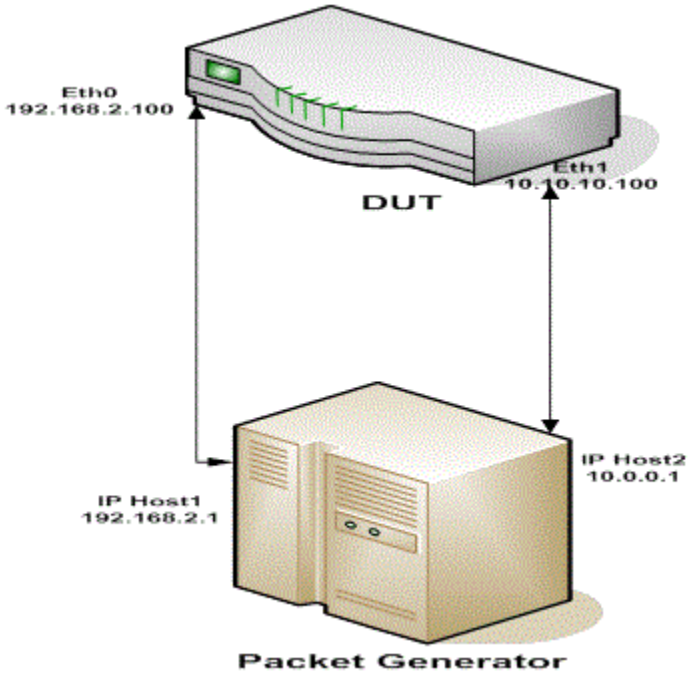
•Policy file for “performance” and “functionality” for DPAA boards are as below
1.“asf-fman-perf-policy.xml ” => performance policy file for P4080, T1040D4RDB boards.
2.“asf-fman-func-policy.xml” => functionality policy file for P4080, T1040D4RDB boards.

•Configuration files for “performance” and “functionality” for various DPAA boards are as below
1."asf-cfg-4080-20g.xml” => for P4080, 2X10G ports enabled
2.“asf-cfg-4080-5g.xml” => for P4080, 5X1g ports enabled
3.“asf-cfg-func-1040.xml” => for T1040D4RDB functionality testing
4.“asf-cfg-perf-1040.xml” => for T1040D4RDB performance testing

To test functionality, use only two port (ports which are functioning on the board) entries in
this file.

NOTE:
-Scripts used above are copies at /usr/lib/asf/scripts/fmc in the rootfs built with ASF enabled.
-Please make sure all the fm interfaces are up that are mentioned in the scripts. In case some
fm interface need to be put down then script shall be modified to remove that port from script.
-Once executed, fmc configuration cannot be reverted back, system need to be restarted.
```

NOTE: Disable the network interface that are not used while taking the performance numbers.

•Configuration IPv4 Flows on Traffic Generator

- Create hosts on the traffic generator Ports connected to DUTs eth1 and eth2 with IP Addresses 192.168.2.1 and 10.0.0.1 respectively.
- Create Routes at the Host

```
#> route add -net 192.169.0.0/16 gw 192.168.2.1
#> route add -net 11.0.0.0/8 gw 10.0.0.1
```

- From Host 1 connected to eth1, create flows for IP/UDP traffic with

```
* Source IP      192.169.1.1  to 192.169.X.X ( as per number of flows)
* Destination IP 11.11.1.1  to   11.11.X.X ( as per number of flows)
```

- From Host 2 connected to eth2, create flows for IP/UDP traffic with

```
* Source IP           11.11.1.1    to  11.11.X.X ( as per number of flows)
* Destination IP  192.169.1.1  to 192.169.X.X (  as per number of flows)
```

- Start the sending traffic from Packet Generator from both Host1 and Host2.
- Verify that packet send from Host1 are received at Host2 and vice-versa and take the Performance numbers.
- Configuration for IPv6 Flows on Traffic Generator

```
•Create hosts on the traffic generator ports connected to DUTs eth0 and eth1 with IP addresses
2001::1/64 and 2000::1/64 respectively.

•From Host 1 connected to eth0, create flows for IP/UDP traffic with
    * Source IP      2001::2 to 2001::x ( as per number of flows)
    * Destination IP 2000::2 to 2000::x ( as per number of flows)

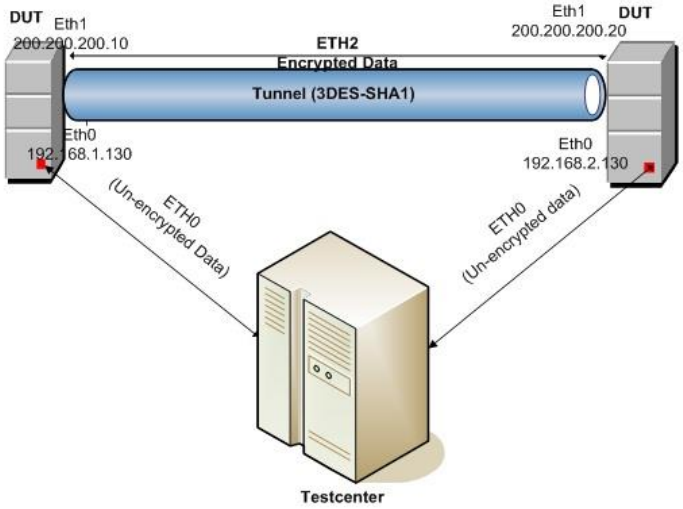
•From Host 2 connected to eth1, create flows for IP/UDP traffic with
    * Source IP           2000::2    to  2000::X ( as per number of flows)
    * Destination IP    2001::2  to 2001::X (  as per number of flows)
```

- Start the sending traffic from Packet Generator from both Host1 and Host2. Verify that packet send from Host1 are received at Host2 and vice-versa and take the Performance numbers.

8.1.2.3 Nat mode

8.1.2.3.1 NAT Hardware instructions



Figure 183. NAT Diagram

8.1.2.3.2 IPv4 NAT Software instructions

Configuration at the DUT

- Configure Ethernet address for eth0 and eth1

```
#> ifconfig eth0 192.168.2.100 netmask 255.255.0.0 up
#> ifconfig eth1 10.10.10.100 netmask 255.0.0.0 up
```

- Enable routing at the DUT

```
#> echo 1 > /proc/sys/net/ipv4/ip_forward
```

- Load ASF modules in IPv4 mode

```
#> insmod asf.ko
#> insmod asfctrl.ko
```

- Enable masquerading at eth1, so that NATing is done on packets going out of eth1 interface

```
#> iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE

NOTE: By default  below rule should be applied while testing NAT/Firewall
iptables -A OUTPUT -p 0x72 -j ACCEPT    /* HERE 0x72 is ASFCTRL_IPPROTO_DUMMY_TERM_L2BLOB */
```

- Increase number of conntrack entries in Linux for measuring performance for 128K flows

```
#> echo 140090 > /proc/sys/net/nf_conntrack_max
```

- Change the default UDP timeout to 9000 seconds. (*for performance testing*)

```
#> echo 9000 > /sys/asfctrl/ffp/asfctrl_ffp_udp_tmout
#> echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout
#> echo 9000 >/proc/sys/net/netfilter/nf_conntrack_udp_timeout_stream
```

Execute commands for non-DPAA platforms only

- Set the coalescing parameters

```
#> ethtool -C eth0 rx-frames 12 rx-usecs 32
#> ethtool -C eth1 rx-frames 12 rx-usecs 32
```

- Disable PTP

```
#> echo 0 > /sys/class/net/eth0/ptp_1588
#> echo 0 > /sys/class/net/eth1/ptp_1588
```

- Set affinities on LS1021ATWR boards:

```
#> echo 2 > /proc/irq/182/smp_affinity ( eth1 TX interrupt )
#> echo 2 > /proc/irq/177/smp_affinity ( eth0 RX interrupt )
#> echo 1 > /proc/irq/176/smp_affinity ( eth0 TX interrupt )
#> echo 1 > /proc/irq/184/smp_affinity ( eth1 RX interrupt )
```

Execute commands for DPAA platforms only

- Execute fmc command for packet distribution

```
#> fmc –s Soft_FragParser.xml –p <policy file> –c <configuration file> –a

    •Policy file for “performance” and “functionality” for DPAA boards are as below
    1.“asf-fman-perf-policy.xml” => performance policy file for P4080, T1040D4RDBboards.
    2.“asf-fman-func-policy.xml” => functionality policy file for P4080, T1040D4RDB boards.

    •Configuration files for “performance” and “functionality” for various DPAA boards are as
below
    1."asf-cfg-4080-20g.xml” => for P4080, 2X10G ports enabled
    2.“asf-cfg-4080-5g.xml” => for P4080, 5X1g ports enabled
    3.“asf-cfg-func-1040.xml” => for T1040D4RDB  functionality testing
    4.“asf-cfg-perf-1040.xml” => for T1040D4RDB performance testing

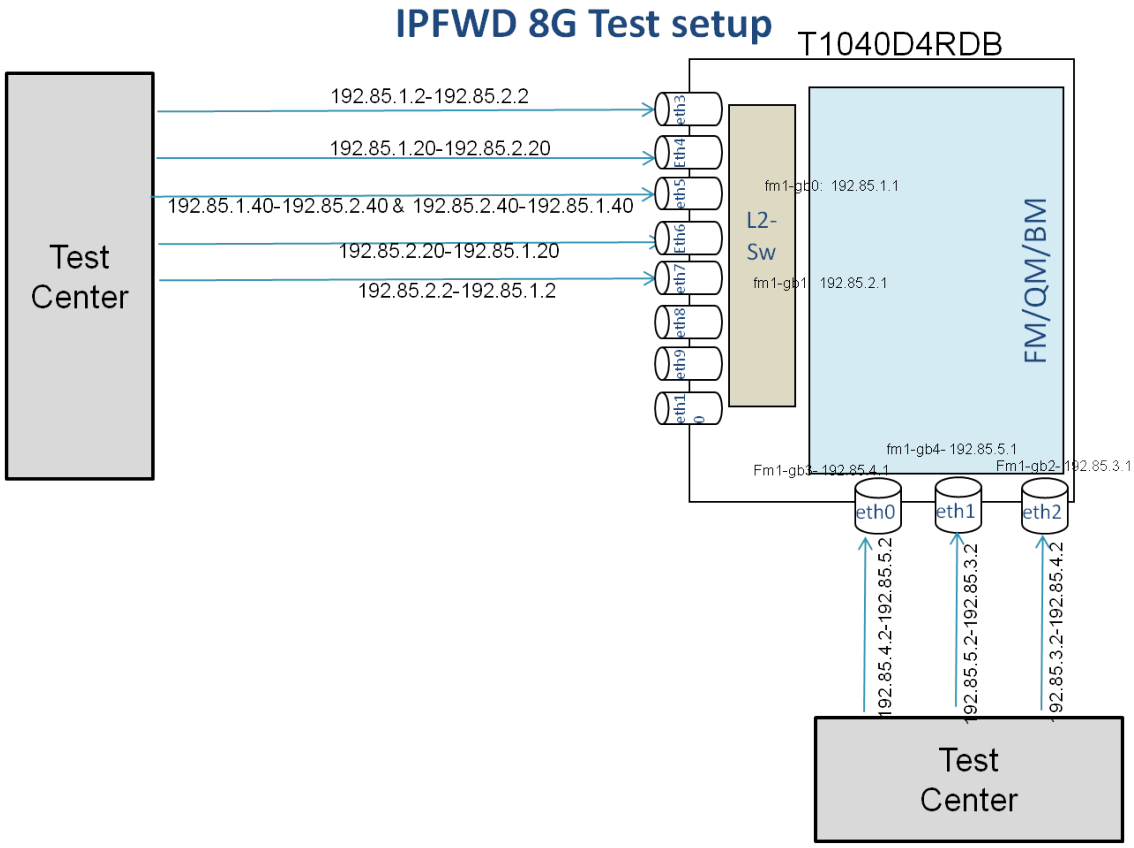
 To test functionality, use only two port (ports which are functioning on the board) entries in
this file.

    NOTE:
    - Scripts used above are copies at /usr/lib/asf/scripts/fmc in the rootfs built with ASF
enabled.
    - Please make sure all the fm interfaces are up that are mentioned in the scripts. In case
some fm interface need to be put down then script shall be modified to remove that port from
script.
    - Once executed, fmc configuration cannot be reverted back, system need to be restarted.
```

- *NOTE: Disable the network interface that are not used while taking the performance numbers.*

Configuration at the Packet Generator

- Create hosts on the traffic generator Ports connected to DUT eth0 and eth1.

- From Host 1 connected to eth0, create flow for UDP traffic with

```
* Source IP as 192.168.2.1.
* Destination IP as 10.0.0.1
* Configure flows with UDP Source Port as (10000 + X) where X takes the value form 1…N and N is
the number of flows. The UDP destination Port as 10000.
```

- From Host 2 connected to eth1, create flows for IP traffic with

```
* Source IP as 10.0.0.1.
* Destination IP as 10.10.10.100
* Configure flows with UDP Source Port as 10000 and UDP destination Port as (10000 + X) where X
takes the value form 1…N and N is the number of flows.
```

•Start the sending traffic from Packet Generator from Host1 and Host2 and then Host2 to Host1.

- Verify that packet send from Host1 are received at Host2, with

```
* Source IP as 10.10.10.100.
* Destination IP as 10.0.0.1
* UDP Source Port as (10000 + X) where X takes the value form 1…N and N is the number of flows.
* UDP destination Port as 10000.
```

- Verify that packets sent from Host2 are received at Host 1, with

```
* Source IP as 10.0.0.1.
* Destination IP as 192.168.2.1.
* UDP source Port as 10000
* UDP Destination Port as (10000 + X) where X takes the value form 1…N and N is the number of
flows.
```

- Start the sending traffic from Packet Generator from Host1 and Host2 and then Host2 to Host1.
- Verify that packet send from Host1 are received at Host2, with

```
•Source IP as 10.10.10.100.
•Destination IP as 10.0.0.1
•UDP Source Port as (10000 + X) where X takes the value form 1…N and N is the number of flows.
•UDP destination Port as 10000.
```

- Verify that packets sent from Host2 are received at Host1, with

```
•Source IP as 10.0.0.1.
•Destination IP as 192.168.2.1.
•UDP source Port as 10000
•UDP Destination Port as (10000 + X) where X takes the value form 1…N and N is the number of
flows.
```

Optional Configurations at the DUT

ASF has exported following SYSFS configuration which can be used to configure the system. The SYSFS interface and their default values are shown as below.

Table 200. NAT SYSFS interface and default values

Commands	Description	Mode	Default Value
/sys/asfctrl/ffp/asfctrl_ffp_activity_divisor	The divisor value for the activity refresh interval.	Read/Write	4
/sys/asfctrl/ffp/asfctrl_ffp_tcp_state_check	Enable/Disable the TCP state check for the new flows.	Read/Write	1
/sys/asfctrl/ffp/asfctrl_ffp_tcp_tm_stmp_check	Enable/Disable the TCP timestamp check for the new flows.	Read/Write	1
/sys/asfctrl/ffp/asfctrl_ffp_tcp_tmout	TCP idle flow timeout.	Read/Write	432000
/sys/asfctrl/ffp/asfctrl_ffp_udp_tmout	UDP idle flow timeout.	Read/Write	180

Also, while loading the ASF Forwarding module we can configure the following module parameters as follows.

Table 201. Module Parameters

Module parameter	Description	Mode	Default Value
/sys/module/asf/parameters/asf_enable	Enable/Disable the ASF	Read/Write	Y (Enable)
/sys/module/asf/parameters/asf_l2blob_refresh_interval	Time period (in sec) for layer 2 header refresh for the flows	Read/Write	180
/sys/module/asf/parameters/asf_max_ifaces	Maximum no. of network interface supported by ASF.	Read	16
/sys/module/asf/parameters/asf_max_vsgs	Maximum no. of VSG interface supported by ASF	Read	2
/sys/module/asf/parameters/asf_reasm_hash_list_size	Hash list size used by reassembly functionality.	Read	256
/sys/module/asf/parameters/asf_reasm_num_cbs		Read	1024

Table continues on the next page...

Table 201. Module Parameters (continued)

Module parameter	Description	Mode	Default Value
/sys/module/asf/parameters/asf_tcp_drop_oos	TCP out of sequence drop.	Read/Write	N
/sys/module/asf/parameters/ffp_hash_buckets	Flow hash bucket size	Read	8192
/sys/module/asf/parameters/ffp_max_flows	Max no. of flows supported by ASF.	Read	131072

NOTE

All the module parameters can be used at module insert time as described below. The read only parameters can only be configured during the module install time and the read/write module parameters can be changed at runtime.

```
#> insmod asf.ko  <MODULE_PARAMETER>=<VALUE>
```

8.1.2.4 IPSec mode

8.1.2.4.1 IPSec Hardware instructions



Figure 184. IPSec Diagram

8.1.2.4.2 IPSec Software instructions

IPv4 Configuration at Both the DUT Platforms

- At DUT1 run following script to verify normal IP forwarding

```
#>./setup_left
```

- At DUT2 run following script to verify normal IP forwarding

```
#>./setup_right
```

IPv6 Configuration at Both the DUT Platforms

- At DUT1 run following script to verify normal IP forwarding

```
#>./setup6_left
```

- At DUT2 run following script to verify normal IP forwarding

```
#>./setup6_right
```

After executing these scripts verify that DUT1 is able to ping Ref Board.

NOTE

For DPAA platforms, update the scripts with proper interface names like fm1-gb1.

- Bring the board up and load ASF modules

```
#> insmod asf.ko
#> insmod asfctrl.ko
#> insmod asfipsec.ko
#> insmod asfctrl_ipsec.ko
```

- Change the default UDP timeout to 9000 (for performance testing)

```
#> echo 9000 > /sys/asfctrl/ffp/asfctrl_ffp_udp_tmout
#> echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout
#> echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout_stream
```

- Disable xfrm_larval_drop

```
#> echo 0 > /proc/sys/net/core/xfrm_larval_drop
```

Execute commands for non-DPAA platforms only

- Set the coalescing parameters

```
#> ethtool -C eth0 rx-frames 22 rx-usecs 32
#> ethtool -C eth1 rx-frames 22 rx-usecs 32
```

- On LS1021ATWR board configure Rx and Tx BD ring sizes

```
#> ethtool -G eth0 rx 128 tx 128
#> ethtool -G eth1 rx 128 tx 128
```

- Set affinities on LS1021ATWR boards:

```
#> echo 2 > /proc/irq/182/smp_affinity ( eth1 TX interrupt )
#> echo 2 > /proc/irq/177/smp_affinity ( eth0 RX interrupt )
#> echo 1 > /proc/irq/176/smp_affinity ( eth0 TX interrupt )
#> echo 1 > /proc/irq/184/smp_affinity ( eth1 RX interrupt )
#> echo 1 > /proc/irq/135/smp_affinity ( CAAM (sec_hw) interrupt )
#> echo 1 > /proc/irq/136/smp_affinity ( CAAM (sec_hw) interrupt )
#> echo 2 > /proc/irq/137/smp_affinity ( CAAM (sec_hw) interrupt )
#> echo 2 > /proc/irq/138/smp_affinity ( CAAM (sec_hw) interrupt )
```

Note: In the above command, the interrupt affinities for RX and Talitos interrupts are configured for the optimal performance.

- *1) For single flow scenarios : eth RX and talitos interrupt shall be affined to different cores.*
- *2) For 2 opposite flows scenario : eth RX and talitos interrupt shall be affined to same cores.*

Execute commands for DPAA platforms only

- Execute fmc command for packet distribution

```
#> fmc -s Soft_FragParser.xml -p <policy file> -c <configuration file> -a
```

```
    •Policy file for "performance" and "functionality" for DPAA boards are as below
    1."asf-fman-perf-policy.xml" => performance policy file for P4080 and T1040D4RDB boards.
    2."asf-fman-func-policy.xml" => functionality policy file for P4080 and T1040D4RDB boards.



    •Configuration files for "performance" and "functionality" for various DPAA boards are as
below
    1."asf-cfg-4080-20g.xml" => for P4080, 2X10G ports enabled
    2."asf-cxfg-4080-5g.xml" => for P4080, 5X1g ports enabled
    3."asf-cfg-func-1040.xml" => for T1040D4RDB functionality testing
    4."asf-cfg-perf-1040.xml" => for T1040D4RDB performance testing

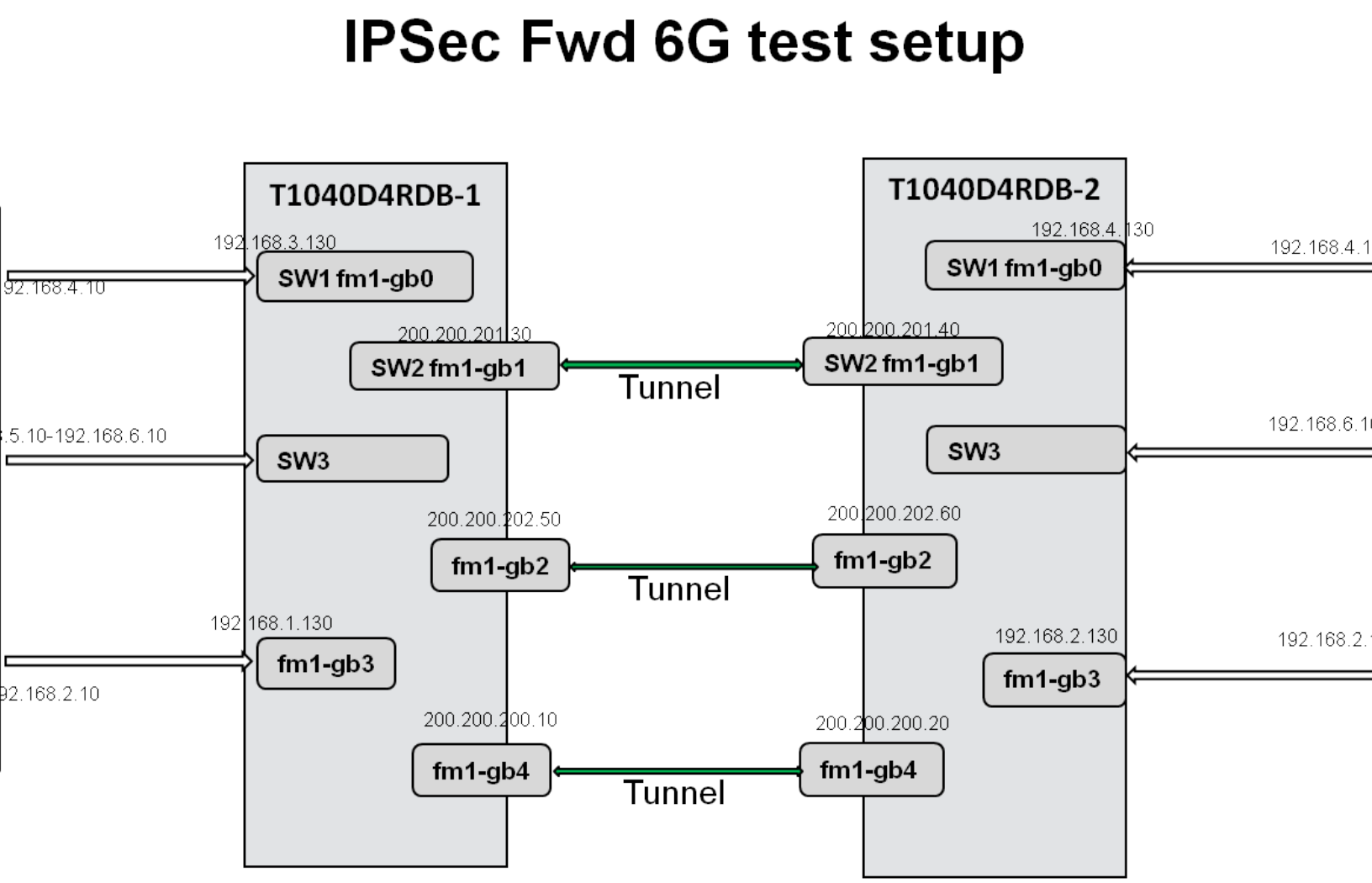
To test functionality, use only two port (ports which are functioning on the board) entries in
this file.

    NOTE:
    - Scripts used above are copies at /usr/lib/asf/scripts/fmc in the rootfs built with ASF
enabled.
    - Please make sure all the fm interfaces are up that are mentioned in the scripts. In case
some fm interface need to be put down then script shall be modified to remove that port from
script.
    - Once executed, fmc configuration cannot be reverted back, system need to be restarted.
```

NOTE: Disable the interface that are not used while taking the performance numbers.

IPv4 Configuration

Configuring ESP IPsec using setkey:

- At DUT1 run flowing script to configure IPsec policy

```
    #>./left.conf-3des-sha1-tunnel
```

- At DUT2 run flowing script to configure IPsec policy

```
    #>./right.conf-3des-sha1-tunnel
```

- *Note : setup_left, setup_right, right.conf-3des-sha1-tunnel and left.conf-3des-sha1-tunnel are copied at /usr/lib/asf/scripts/ipsec/setkey in rootfs compiled with ASF enabled.*

Configuring AH IPsec using setkey:

- At DUT1 run flowing script to configure IPsec policy

```
#>./left.conf-ah-sha1-tunnel
```

- At DUT2 run flowing script to configure IPsec policy

```
#>./right.conf-ah-sha1-tunnel
```

- **NOTE: Dynamic re-keying is supported only in FULL mode of ASF.**

Configuring IPsec using Racoon:

In case system needs to be configured for dynamic key exchange in IPsec using Racoon then the following commands shall be used.

- At DUT1 run flowing script to configure IPsec policy

```
#>./left-ipv4-tunnel
```

- At DUT2 run flowing script to configure IPsec policy

```
#>./right-ipv4-tunnel
```

The above commands will configure the SPD database and make the system ready to start negotiating SA's for flows matching the policies. To start dynamic key exchange and SA negotiation run following commands:

- At DUT1 run flowing script to configure IPsec policy

```
#> chmod 400 psk.txt
#> racoon –f racoon.conf
```

- At DUT2 run flowing script to configure IPsec policy

```
#> chmod 400 psk.txt
#> racoon –f racoon.conf
```

left-ipv4-tunnel, right-ipv4-tunnel, psk.txt and racoon.conf are copied at /usr/lib/asf/scripts/ipsec/raccon in rootfs compiled with ASF enabled.

These commands will start the racoon which reads system configuration from the racoon.conf file present in the system and runs as a background process.

Note :- In case the debug prints needs to be enabled then while running raccoon please can specify the level of debug prints using "-dddd" option wherein the number of "d" tells the debug level. Another important option for racoon is in case it needs to be run in foreground then specify it via –F option.

Configuring IPSec using Strongswan:

In case system needs to be configured for dynamic key exchange in IPSec using Strongswan then the following commands shall be used.

- At DUT1 run following script/commands to configure IPSec policy

```
#>./strongswan_left
#> ipsec start
```

- At DUT2 run following script/commands to configure IPSec policy

```
#>./strongswan_right
#> ipsec start
#> ipsec up net-net
```

strongswan_left, strongswan_right, are copied at /usr/lib/asf/scripts/ipsec/sswan in rootfs compiled with ASF enabled.

The above commands will configure the SPD database and make the system ready to start negotiating SA's for flows matching the policies. To start dynamic key exchange and SA negotiation run following commands:

Configuration at the Packet Generator:

- Create host on the traffic generator Ports connected to DUT1 and DUT2
- For Host 1 connected to DUT1 eth0, create flows for IP traffic with

```
* Source IP 192.168.1.21
* Destination IP as 192.168.2.21
* Gateway as 192.168.1.130
```

- For Host 2 connected to eth1, create flows for IP traffic with

```
* Source IP as 192.168.2.21
* Destination IP as 192.168.1.21
* Gateway as 192.168.2.130
```

- Start the sending traffic from Packet Generator from both Host1 and Host2.
- Verify that packet send from Host1 are received at Host2 and vice- versa.
- Packet between DUT1 and DUT2 will be IPSec encrypted.
- For testing multiple flows configure Host1 and Host2 at the Packet generator to send packets with incrementing values of Source IP and Destination IP.

Ipv6 Configuration

Configuring ESP IPsec using setkey:

- At DUT1 run flowing script to configure IPsec policy

```
./left_ipv6.conf-3des-sha1-tunnel
```

- At “Reference board” run flowing script to configure IPsec policy

```
./right_ipv6.conf-3des-sha1-tunnel
```

Configuring AH IPsec using setkey:

- At DUT1 run flowing script to configure IPsec policy

```
#> ./ipv6_left.conf-ah-sha1-tunnel
```

- At “Reference board” run flowing script to configure IPsec policy

```
#> ./ipv6_right.conf-ah-sha1-tunnel
```

- *Note : setup_left, setup_right, right.conf-3des-sha1-tunnel and left.conf-3des-sha1-tunnel are copied at /usr/lib/asf/scripts/ipsec/setkey in rootfs compiled with ASF enabled.*
- Above mentioned scripts will configure p4080 board as per the configuration shown in Test bed above.
- *NOTE: After above configuration ping6 from DUT to reference board tunnel end and vice-versa. Tunnel ends should ping6 each other.*

```
•Configuration at the Packet Generator
   •Create host on the traffic generator Ports connected to DUT1 and Reference Board
   •For Port1 connected to DUT1 fm2-gb0, create flows for IP traffic with
     * Source IP 2001::1
     * Destination IP as 2002::1
     * Gateway as 2001::130
   •For  port2 connected to eth0, create flows for IP traffic with
     * Source IP as 2002::1
     * Destination IP as 2001::1
     * Gateway as 2002::130
```

- Start the sending traffic from Packet Generator from both Port1 and Port2.
- Verify that packet send from Port1 are received at Port2 and vice- versa.
- Packet between DUT1 and DUT2 will be IPSec encrypted.
- For testing multiple flows configure Port1 and Port2 at the packet generator to send packets with incrementing values of source IP and destination IP.

The tables below summarizes the supported encryption and authentication algorithm combinations for various sec versions.

Table 202. SEC4.X onwards

Mode	Encryption	Authentication
ESP	AES-CBC(256)	HMAC-SHA-1/HMAC-SHA-256
ESP	3DES_EDE-CBC(168)	HMAC-SHA-1/HAMC-SHA-256
AH		HMAC-SHA-1/HMAC-MD5

Table 203. SEC3.X

Mode	Encryption	Authentication
ESP	3DES-CBC(256)	HMAC-SHA-1/HMAC-MD5
ESP	AES-CBC(256)	HMAC-SHA-1/HMAC-MD5

ASF IPsec has exported following PROC/SYS file system configuration which can be used to configure the system. The parameters and their default values are shown as below.

Table 204. SYSFS Interface Default Values

Commands	Description	Mode	Default Value
/proc/sys/asfipsec/ulL2BlobRefreshPktCnt_g	The number of packets count after which refresh for Layer 2 header will be triggered by ASF IPsec. The value '0' indicates that refresh of Layer 2 header based on packet count is *disabled*.	Read/Write	0
/proc/sys/asfipsec/ulL2BlobRefreshTimeInSec_g	The time in seconds after which refresh for Layer 2 header will be triggered by ASF IPsec.	Read/Write	180
/proc/sys/asfipsec/ulMaxSPDContainers_g	The maximum number of Security Policies supported by ASF IPsec.	Read	256
/proc/sys/asfipsec/ulMaxSupportedIPSecSAs_g	The maximum number of Security Association's supported by ASF IPsec.	Read	256
/proc/sys/asfipsec/ulMaxTunnels_g	The maximum number of tunnels supported by ASF IPsec.	Read	256

Table continues on the next page...

Table 204. SYSFS Interface Default Values (continued)

Commands	Description	Mode	Default Value
/proc/sys/asfipsec/ulMaxVSGs_g	The maximum number of VSG's supported by ASF. *Note: Only 1 VSG is supported with Linux	Read	2*

Also, while loading the ASF IPsec Control module one can configure the following module parameters as follows.

Table 205. ASF IPsec Control Module Parameters

Module Parameter	Description	Mode	Default Value
/sys/module/asfctrl_ipsec/parameters/bRedSideFragment	Enable/Disable the Red side fragmentation(Pre IPSec fragmentation)	Read	Y (Enable)
/sys/module/asfctrl_ipsec/parameters/bAntiReplayCheck	Enable/Disable Anti Replay Check	Read	Y (Enable)

8.1.2.5 T1040D4rdb IP Fwd and IPSec Fwd

8.1.2.5.1 Hardware instructions for T1040D4RDB -8G setup



Figure 185. T1040D4RDB -IP Firewall forwarding Diagram

8.1.2.5.2 Software instructions for T1040D4RDB - 8G performance

Configuration at the DUT

- Configure Ethernet address for fm1-gb0 to fm1-gb4

```
#> ifconfig  fm1-gb0 192.85.1.1 netmask 255.255.255.0 up
#> ifconfig  fm1-gb1 192.85.2.1 netmask 255.255.255.0 up
#> ifconfig  fm1-gb2 192.85.3.1 netmask 255.255.255.0 up
#> ifconfig  fm1-gb3 192.85.4.1 netmask 255.255.255.0 up
#> ifconfig  fm1-gb4 192.85.5.1 netmask 255.255.255.0 up
```

- Enable routing at the DUT

```
#> echo 1 > /proc/sys/net/ipv4/ip_forward
```

- Load ASF modules

```
#> insmod asf.ko
#> insmod asfctrl.ko
```

- Increase number of conntrack entries in Linux for measuring performance for 128K flows

```
#> echo 140000 > /proc/sys/net/nf_conntrack_max
```

- Change the default UDP timeout to 9000 seconds. (for performance testing)

```
#> echo 9000 > /sys/asfctrl/ffp/asfctrl_ffp_udp_tmout
```

- Execute fmc command for packet distribution

```
#> fmc -s Soft_FragParser.xml -p asf-fman-perf-policy.xml -c asf-cfg-perf-1040.xml -a
```

NOTE

- Scripts used above are copies at /usr/lib/asf/scripts/fmc in the rootfs built with ASF enabled.

- Please make sure all the fm interfaces are up that are mentioned in the scripts. In case some fm interface need to be put down then script shall be modified to remove that port from script.

- Once executed, fmc configuration cannot be reverted back, system need to be restarted.

- Change the default UDP timeout to 9000 seconds. (*for performance testing*)

```
#> echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout
#> echo 9000 >/proc/sys/net/netfilter/nf_conntrack_udp_timeout_stream
```

•Configuration of switch on T1040D4RDB board

Run the switch binary on T1040D4rdb:

```
# l2sw_bin
```

Add the source MAC address of the hosts configured in the SPIRENT streams:

```
# l2switch>mac add 00:10:94:00:00:01 2
# l2switch>mac add 00:10:94:00:00:02 3
# l2switch>mac add 00:10:94:00:00:03 4
# l2switch>mac add 00:10:94:00:00:06 4
# l2switch>mac add 00:10:94:00:00:05 5
# l2switch>mac add 00:10:94:00:00:04 6

Note: Port numbers (mentioned above) of the switch can vary based on the mapping done with the
Spirent.

Also add the MAC address of switch ports(fm1-gb0 and fm1-gb1) that board has initialized (MAC will
vary from board to board)

# l2switch>mac add 00:E0:0C:00:55:00 8
# l2switch>mac add 00:E0:0C:00:55:01 9

Switch off the polling on the board:
# l2switch>poll off
```

Configuration IPv4 Flows on Traffic Generator

- Refer to the Hardware setup
- Create hosts on the traffic generator Ports connected to DUTs interface with IP addresses 192.85.1.1, 192.85.2.1, 192.85.3.1, 192.85.4.1 and 192.85.5.1 respectively.

- From Host 1 connected to switch eth3 as in diagram(fm1-gb0), create flows for UDP traffic with

```
Source IP      192.85.1.2 (as per number of flows)
Destination IP 192.85.2.2 (as per number of flows)
```

- From Host 2 connected to switch eth4 as in diagram(fm1-gb0), create flows for UDP traffic with

```
Source IP      192.85.1.20 (as per number of flows)
Destination IP 192.85.2.20 (as per number of flows)
```

- From Host 3 connected to switch eth5 as in diagram(switch port), create flows for UDP traffic with

```
Source IP      192.85.1.40 (as per number of flows)
Destination IP 192.85.2.40 (as per number of flows)
           &
Source IP      192.85.2.40 (as per number of flows)
Destination IP 192.85.1.40 (as per number of flows)
```

- From Host 4 connected to switch eth6 in diagram(fm1-gb1), create flows for UDP traffic with

```
Source IP      192.85.2.20 (as per number of flows)
Destination IP 192.85.1.20 (as per number of flows)
```

- From Host 5 connected to switch eth7 in diagram(fm1-gb1), create flows for UDP traffic with

```
Source IP      192.85.2.2 (as per number of flows)
Destination IP 192.85.1.2 (as per number of flows)
```

- From Host 6 connected to switch eth0 (fm1-gb3), create flows for UDP traffic with

```
Source IP      192.85.4.2 (as per number of flows)
Destination IP 192.85.5.2 (as per number of flows)
```

- From Host 7 connected to switch eth1 in diagram(fm1-gb4), create flows for UDP traffic with

```
Source IP      192.85.5.2 (as per number of flows)
Destination IP 192.85.3.2 (as per number of flows)
```

- From Host 8 connected to switch eth2 in diagram(fm1-gb2), create flows for UDP traffic with

```
Source IP      192.85.3.2 (as per number of flows)
Destination IP 192.85.4.2 (as per number of flows)
```

- Start the sending traffic from Packet Generator from all the Hosts.

8.1.2.5.3 Hardware instructions for T1040D4RDB-IPSec 6G setup



Figure 186.

8.1.2.5.4 Software instructions for T1040D4RDB IPSec - 6g

Configuration at both the DUT

- On DUT1 run flowing commands

```
# ifconfig  fm1-gb0 192.168.3.130 netmask 255.255.255.0 up
# ifconfig  fm1-gb1 200.200.201.30 netmask 255.255.255.0 up
# ifconfig  fm1-gb2 200.200.202.50 netmask 255.255.255.0 up
# ifconfig  fm1-gb3 192.168.1.130  netmask 255.255.255.0 up
```

```
  # ifconfig  fm1-gb4 200.200.200.10 netmask 255.255.255.0 up
  # ip route add 192.168.2.0/24 dev fm1-gb4
  # ip route add 192.168.4.0/24 dev fm1-gb1
  # ip route add 192.168.6.0/24 dev fm1-gb2
  # ip route add 192.168.5.0/24 dev fm1-gb0
```

- On DUT2 run flowing commands

```
# ifconfig  fm1-gb0 192.168.4.130 netmask 255.255.255.0 up
# ifconfig  fm1-gb1 200.200.201.40 netmask 255.255.255.0 up
# ifconfig  fm1-gb2 200.200.202.60 netmask 255.255.255.0 up
# ifconfig  fm1-gb3 192.168.2.130  netmask 255.255.255.0 up
# ifconfig  fm1-gb4 200.200.200.20 netmask 255.255.255.0 up
# ip route add 192.168.1.0/24 dev fm1-gb4
# ip route add 192.168.3.0/24 dev fm1-gb1
# ip route add 192.168.5.0/24 dev fm1-gb2
# ip route add 192.168.6.0/24 dev fm1-gb0
```

- **Configuration at both the DUT**

```
#> insmod asf.ko
#> insmod asfctrl.ko
#> insmod asfipsec.ko
#> insmod asfctrl_ipsec.ko
```

- Change the default UDP timeout to 9000 (for performance testing)

```
#> echo 9000 > /sys/asfctrl/ffp/asfctrl_ffp_udp_tmout

•Enable routing at the DUT
#> echo 1 > /proc/sys/net/ipv4/ip_forward
```

- Disable xfrm_larval_drop

```
#> echo 0 > /proc/sys/net/core/xfrm_larval_drop
```

- Change the default UDP timeout to 9000 seconds. (for performance testing)

```
#> echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout
#> echo 9000 >/proc/sys/net/netfilter/nf_conntrack_udp_timeout_stream
```

- Execute fmc command for packet distribution

```
#> fmc –s Soft_FragParser.xml –p asf-fman-perf-policy.xml –c asf-cfg-perf-1040.xml –a
```

NOTE

NOTE:

- Scripts used above are copies at /usr/lib/asf/scripts/fmc in the rootfs built with ASF enabled.

- Please make sure all the fm interfaces are up that are mentioned in the scripts. In case some fm interface need to be put down then script shall be modified to remove that port from script.

- Once executed, fmc configuration cannot be reverted back, system need to be restarted.

Configuring ESP IPsec using setkey:

- At T1040D4RDB-1, run flowing embedded script to configure IPsec policy

```
#> ./left_t1040D4rdb_6g_esp-sha1.txt


Create the above executable using following:

#!/usr/sbin/setkey -f
```

```
flush;
spdflush;

spdadd 192.168.1.10 192.168.2.10 any -P out ipsec esp/tunnel/200.200.200.10-200.200.200.20/
unique:1;
spdadd 192.168.2.10 192.168.1.10 any -P in ipsec esp/tunnel/200.200.200.20-200.200.200.10/
unique:1;

add 200.200.200.10 200.200.200.20 esp 0x207 -u 1 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;
add 200.200.200.20 200.200.200.10 esp 0x307 -u 1 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;

spdadd 192.168.3.10 192.168.4.10 any -P out ipsec esp/tunnel/200.200.201.30-200.200.201.40/
unique:2;
spdadd 192.168.4.10 192.168.3.10 any -P in ipsec esp/tunnel/200.200.201.40-200.200.201.30/
unique:2;

add 200.200.201.30 200.200.201.40 esp 0x407 -u 2 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;
add 200.200.201.40 200.200.201.30 esp 0x507 -u 2 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;

spdadd 192.168.5.10 192.168.6.10 any -P out ipsec esp/tunnel/200.200.202.50-200.200.202.60/
unique:3;
spdadd 192.168.6.10 192.168.5.10 any -P in ipsec esp/tunnel/200.200.202.60-200.200.202.50/
unique:3;

add 200.200.202.50 200.200.202.60 esp 0x607 -u 3 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;
add 200.200.202.60 200.200.202.50 esp 0x707 -u 3 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;
```

- At T1040D4RDB-2, run flowing embedded script to configure IPsec policy

```
#>./right_t1040D4rdb_6g_esp-sha1.txt

Create the above executable using following:


#!/usr/sbin/setkey -f
flush;
spdflush;

spdadd 192.168.1.10 192.168.2.10 any -P in ipsec esp/tunnel/200.200.200.10-200.200.200.20/
unique:1;
spdadd 192.168.2.10 192.168.1.10 any -P out ipsec esp/tunnel/200.200.200.20-200.200.200.10/
unique:1;

add 200.200.200.10 200.200.200.20 esp 0x207 -u 1 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;
add 200.200.200.20 200.200.200.10 esp 0x307 -u 1 -m tunnel -E aes-cbc
```

```
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;

spdadd 192.168.3.10 192.168.4.10 any -P in ipsec esp/tunnel/200.200.201.30-200.200.201.40/
unique:2;
spdadd 192.168.4.10 192.168.3.10 any -P out ipsec esp/tunnel/200.200.201.40-200.200.201.30/
unique:2;

add 200.200.201.30 200.200.201.40 esp 0x407 -u 2 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;
add 200.200.201.40 200.200.201.30 esp 0x507 -u 2 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;

spdadd 192.168.5.10 192.168.6.10 any -P in ipsec esp/tunnel/200.200.202.50-200.200.202.60/
unique:3;
spdadd 192.168.6.10 192.168.5.10 any -P out ipsec esp/tunnel/200.200.202.60-200.200.202.50/
unique:3;

add 200.200.202.50 200.200.202.60 esp 0x607 -u 3 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;
add 200.200.202.60 200.200.202.50 esp 0x707 -u 3 -m tunnel -E aes-cbc
0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831 -A hmac-sha1
0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;
```

-

L2 switch configuration on both DUT:

- Configuration of switch on T1040D4RDB-1 board.
- Run the switch binary on T1040D4rdb

```
# l2sw_bin


Note: For Performance testing, before adding MAC address statically, start traffic for all the
streams Configured, so that both the T1040D4RDB boards creates a MAC table dynamically.
Use l2switch>mac_dump command to dump all the addresses leaned. Use the dynamically learned mac
address entries to add static mac address as below:

Add the MAC address of switch ports (fm1-gb0 and fm1-gb1) that board has initialized (MAC will
vary from board to board)

l2switch> mac add 00:e0:0c:00:8a:00 8
l2switch> mac add 00:e0:0c:00:8a:01 9

Also add the source MAC address of the hosts configured in the SPIRENT streams and for ESP
tunnel interface connected with switch ports:

l2switch> mac add 00:10:94:00:00:02 2
l2switch> mac add 00:10:94:00:00:03 3
l2switch> mac add 00:e0:0c:00:82:02 9

Switch off the polling on the board:
l2switch>poll off
```

- Configuration of switch on T1040D4RDB-2 board

- Run the switch binary on T1040D4rdb

```
# l2sw_bin

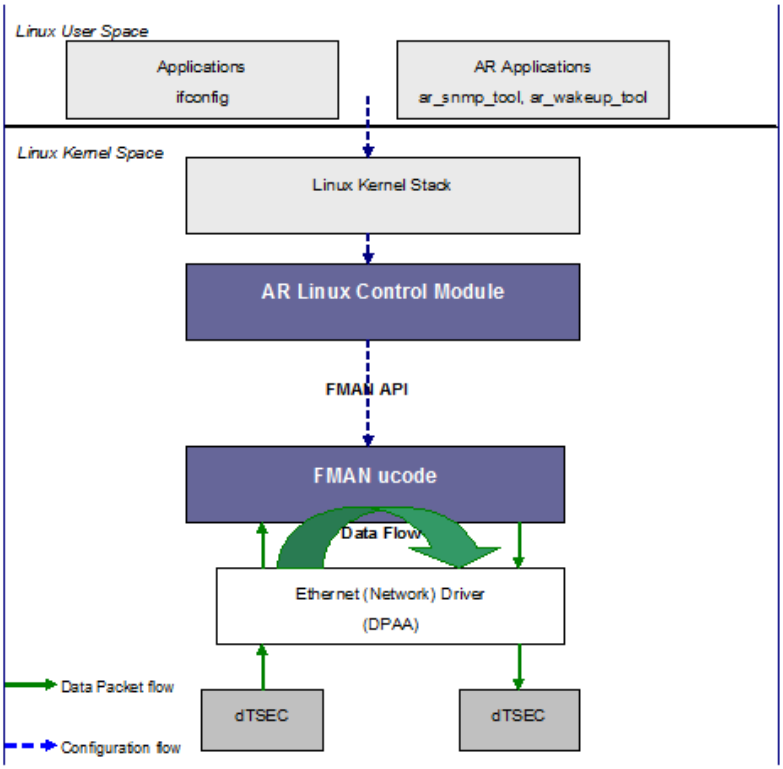
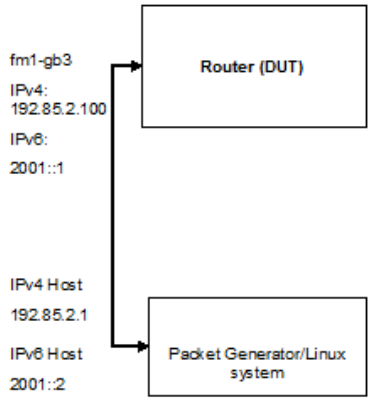
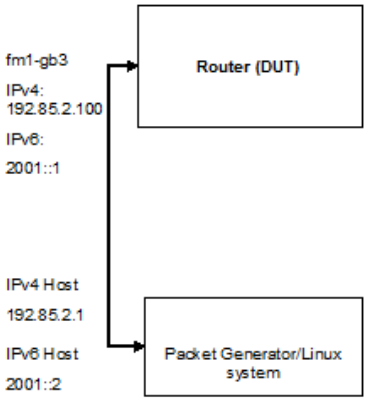

Note: For Performance testing, before adding MAC address statically, start traffic for all the
streams Configured, so that both the T1040D4RDB boards creates a MAC table dynamically.
Use l2switch>mac_dump command to dump all the addresses leaned. Use the dynamically learned mac
address entries to add static mac address as below:

Add the MAC address of switch ports (fm1-gb0 and fm1-gb1) that board has initialized (MAC will
vary from board to board)

l2switch> mac add 00:e0:0c:00:82:01 8
l2switch> mac add 00:e0:0c:00:82:02 9

Also add the source MAC address of the hosts configured in the SPIRENT streams and for ESP
tunnel interface connected with switch ports:

l2switch> mac add 00:10:95:00:00:02 2
l2switch> mac add 00:10:95:00:00:03 3
l2switch> mac add 00:e0:0c:00:8a:01 9

Switch off the polling on the board:
l2switch>poll off
```

- **NOTE**

  Port numbers (mentioned above) of the switch can vary based on the mapping done with the Spirent and other board. Use dynamic learning to identify correct ports and mac learning.

Configuration at the Packet Generator:

- Create hosts on the traffic generator Ports connected to T1040D4RDB-1 and T1040D4RDB-2
- For T1040D4RDB-1 connected to spirent, create flows for UDP traffic with

```
* Src IP 192.168.1.10, Dst IP as 192.168.2.10, GW 192.168.1.130
* Src IP 192.168.3.10, Dst IP as 192.168.4.10, GW 192.168.3.130
* Src IP 192.168.5.10, Dst IP as 192.168.6.10, GW 192.168.3.130
```

- For T1040D4RDB-2 connected to spirent, create flows for UDP traffic with

```
* Src IP 192.168.2.10, Dst IP as 192.168.1.10, GW 192.168.2.130
* Src IP 192.168.4.10, Dst IP as 192.168.3.10, GW 192.168.4.130
* Src IP 192.168.6.10, Dst IP as 192.168.5.10, GW 192.168.4.130111
```

- Start sending traffic from Packet Generator from both sides.
- Verify that packet send from DUT1 are received at DUT2 and vice- versa.
- Packet between DUT1 and DUT2 will be IPSec encrypted.

8.1.3 Troubleshooting

8.1.3.1 Viewing Statistics/Information via /proc interface

To verify that ASF is processing the packets; the proc interface which provides various options that can be used to see what is happening in the system. The proc interface gets populated based on mode in which ASF is compiled in.

In Maximum mode all the stats are visible through proc interface, in Minimal Mode various statistics are displayed as 0.

The following section shows the proc statistics

Table 206. *proc* Statistics

Commands	Description
cat /proc/asf/global_stats	Provide information for • Packets received and transmitted by the ASF. • NAT/Firewall flows allocated in ASF. • Error Counters • Number of packets sent for Linux stack for processing
cat /proc/asf/flow_stats	Displays Connection tracking entries configured in the system in the NAT mode.
cat /proc/asf/ifaces	Displays the interfaces on which ASF is enabled
cat /proc/asfipsec/global_stats	Displays IPsec statistics
cat /proc/asfipsec/global_error	Displays IPsec error statistics.
cat /proc/asfipsec/out_spd	Display Encryption IPsec Policy and SA offloaded to ASF
cat /proc/asfipsec/in_spd	Display Decryption IPsec Policy and SA offloaded to ASF

8.1.3.2 Common Usage Issues

Table 207. Common Usage Issues

Symptoms	Reason and/or Recommended actions
NAT/Firewall performance is much lower than expected.	Verify that conntrack entries are offloaded and packets are getting routed through ASF using commands cat /proc/asf/flow_stats cat /proc/asf/global_stats Make sure that ASF is loaded before starting the traffic for flows to be offloaded to ASF. Verify that no USB and MMC interrupts are coming. If interrupts come, disable the MMC and USB from Linux.
In Minimal mode Flows NAT/Firewall are offloaded but performance is still down	Dynamic ARP entries may not be offloaded at time flows were offloaded. Try with creating static ARP entries before the traffic is started for flows. If it works check why Dynamic ARP is not getting resolved at the setup.
Statistics are not updated	Verify that Image is compiled for FULL ASF mode. In Minimal mode statistics are disabled.

Table continues on the next page...

Table 207. Common Usage Issues (continued)

Symptoms	Reason and/or Recommended actions
IPsec forwarding in minimal mode is not working.	Verify that ARP for Ethernet interfaces are resolved at the DUT. Refer section 3.4
IPsec forwarding not working in minimal mode and full mode	Verify that ASF is running in NAT/Firewall mode, In IPv4 forwarding mode IPsec is not supported.
IPsec ESP/AH SHA256 is not working with Linux.	Verify that the ipsec-tools are overriding the default ICV configured by Linux. Try using strongswan for IPsec configuration.
Strongswan v5.1.1 is unable to start with "*ipsec start*" command.	Remove "auth=esp" from "/etc/ipsec.conf" file

8.2 Auto Response

8.2.1 Introduction

8.2.1.1 Purpose

This document details the usage of AR solution. AR is a Linux kernel implementation for offloading the network configuration to the ucode and processing the incoming based on the configuration.

8.2.1.2 Scope

This document is intended to help developers, test engineers and customers to use NXP AR implementation on NXP's family (QORIQ) of Network processors.

8.2.1.3 Definitions and Acronyms

AR	Auto Response
ARP	Address Resolution Protocol
ND	Neighbor Discovery
SNMP	Simple Network Management Protocol
ICMP	Internet Control Message Protocol
FMAN	Frame Manager
Ucode	FMAN Micro Code
PM	Power Management
DUT	Device Under Test

8.2.2 Product Description

8.2.2.1 Overview

AR is a Linux kernel implementation to make the device intelligent enough so that it can respond to the network packets when device is in power saving mode (Deep Sleep Mode).

AR implementation is compiled as Kernel dynamic loadable module that can be plugged in to Linux at run time. Once plugged into Linux, AR will be ready to get trigger from Power management to into deep sleep state. Once a user instructs the device to enter into deep sleep, Network configuration, for the Auto response port i.e fm1-gb3, will be offloaded to the ucode and ucode will start responding to the network packets.

8.2.2.2 Key Target Applications

AR is capable to respond the following protocols packets:

- 1)ARP
- 2)ND
- 3)ICMPv4
- 4)ICMPv6
- 5)SNMP

Also AR will be capable to wake the device for specified packets which will be configured by AR as per the user requirement.

8.2.2.3 Product Diagram

Figure below identifies how the AR module fits in the overall system in typical network processor software Architecture.



Figure 187. AR Packet Processing

8.2.2.4 Features

As a part of delivery user will receive AR as part of Yocto source release.

Following features are supported in this release for Auto response port. Following feature will be supported for both, physical and logical, interfaces.

Following features are supported in this release

Table 208. Features

Feature	Detailed Description
ARP	ARP packets will be processed by ucode as its configuration.
ND	ND packets will be processed by ucode as its configuration.
ICMPv4	ICMPv4 packets will be processed by ucode as its configuration.
ICMPv6	ICMPv6 packets will be processed by ucode as its configuration.
SNMP	SNMP packets will be processed by ucode as its configuration
WAKEup	Device will be waken-up for the configured packet rules

8.2.2.5 Specifications

Platform Supported

- T1040D4RDB
- T1042D4RDB

Limitations:

- SNMP is not functional in current release.

8.2.2.6 Environment Required

8.2.2.6.1 Hardware Configuration

Refer platform Specific Documentation for Board Configuration.

8.2.2.6.2 Software Configuration

NA

8.2.2.6.3 Software Tools

1. SNMP Client needs to be installed on some machine (Windows or Linux/Unix) for sending SNMP requests for testing.
2. SNMP Server needs to be installed on DUT to process SNMP request.

8.2.2.7 Product Directory Structure

NA

8.2.3 Product Execution

8.2.3.1 Getting started

Following are the images which are required to run Auto Response.

1. Linux Image (uImage)
2. Rootfs (Containing auto response kernel module and user space utilities)
3. Device tree blob with AR supported changes
4. Ucode which will be running in independent mode in deep sleep state

8.2.3.2 ARP/ND/ICMP Packet Processing

8.2.3.2.1 Hardware setup



8.2.3.2.2 Software instructions

Configuration at the DUT

1. Installing Auto Response module

```
insmod ar.ko ar_resport="fm1-gb3"
```

2. Configure Ethernet address for fm1-gb3

- IPv4

```
ifconfig fm1-gb3 192.85.2.100 netmask 255.255.255.0 up
```

- IPv6

```
ip -6 addr add 2001::1/64 dev fm1-gb 3
```

3. Configure Ethernet address for VLAN interface over fm1-gb3 (Optional configuration)
   - IPv4

```
vconfig add fm1-gb3 1
ifconfig fm1-gb3.1 192.85.3.100 netmask 255.255.255.0 up
```

   - IPv6

```
vconfig add fm1-gb3 1
ip -6 addr add 2002::1/64 dev fm1-gb3.1
ifconfig fm1-gb3.1 up
```

4. Configure Ethernet address for logical interface over fm1-gb3 (Optional configuration)
   - IPv4

```
ifconfig fm1-gb3:1 192.85.4.100 netmask 255.255.255.0 up
```

   - IPv6

```
ip -6 addr add 2003::1/64 dev fm1-gb3:1
ifconfig fm1-gb3:1 up
```

5. Enable/Disable conflict detection flag
   - ARP conflict detection

```
echo 1 > /sys/arctrl/arp_conflict_detect
echo 0 > /sys/arctrl/arp_conflict_detect
```

   Where:

   0 - Conflict detection is disabled.
   The host CPU will not be woken upon conflict detection and the packet will be processed by ucode as a normal frame.

   1 - Conflict detection is enabled.
   Ethernet-IPv4 ARP frames, which contain address conflict, will cause the FMan Controller to proceed to the 'Active' mode processing flow. According to the 'Active mode' configuration, this can cause a system wake-up or frame will be discarded. By default conflict detection flag is disabled.

6. Configuring rollback option during deep sleep entrance. There could be a possibility that system has some frames to be processed while getting into the deep sleep state then in this case system should not get into the deep sleep and

should roll back from suspend sequence. It needs to be configured so that system will rollback while getting into deep sleep entrance. The following is the command to configure the same:

```
echo `cat /sys/power/wakeup_count ` > /sys/power/wakeup_count && echo mem >
/sys/power/state
```

7. Getting into deep sleep state without enabling rollback option

```
  echo mem > /sys/power/state
```

Configuration of hosts on Traffic Generator

1. Create an IPv4 host on the traffic generator on port connected to DUT i.e. fm1-gb3
2. Configure the IPv4 address as on the host
3. Create an IPv6 host on the traffic generator on port connected to DUT i.e. fm1-gb3
4. Configure the IPv6 address as on the host

Run test

1. Resolve ARP Request for the configured IPv4 addresses on DUT.
2. Resolve NDP advertisement request for the configured IPv6 addresses on DUT.
3. Ping from the host to the configured IPv4/IPv6. Ping response will be successful.
4. Send any packet other than ping i.e. UDP/TCP packet. System will wake up.

8.2.3.3 Wake up processing

8.2.3.3.1 Hardware setup



8.2.3.3.2 Software instructions

Configuration at the DUT

1. Installing Auto Response module

```
insmod ar.ko ar_resport="fm1-gb3"
```

2. Configure Ethernet address for fm1-gb3
   - IPv4

```
ifconfig fm1-gb3 192.85.2.100 netmask 255.255.255.0 up
```

   - IPv6

```
ip -6 addr add 2001::1/64 dev fm1-gb3
```

3. Configure user defined wakeup rules
   a. Modify ar_wakeup_rule_cfg file to configure wakeup rules.

      The file contains rules for following filters:

      - IP protocol filters

This filter identifies the list of protocols which are to be considered to wake the system up. Protocol entry in this ar_wakeup_rule_cfg is written as a string representing the standard IP protocols. For example:

- TCP for tcp packets
- UDP for udp packets
- ICMP for IPv4 ICMP packets
- IPv6-ICMP for IPv6 ICMP packets
- IGMP for IGMP (v1/v2/v3) packets

To add more protocols, mention the protocol as per standard IP protocol keyword.

- UDP ports filters

  This filter identifies the rules on UDP port number which will be used to compare with UDP traffic for packet filtering. Rules are written in following format:

  Source port | Destination port | Source port Mask | Destination port Mask

- TCP ports filters

  This filter identifies the rules on TCP port number which will be used to compare with UDP traffic for packet filtering. Rules are written in following format:

  Source port | Destination port | Source port Mask | Destination port Mask

b. Configure action if packet hits any of the filter

```
echo (1/0) > /sys/arctrl/ip_prot_pass_on_hit
echo (1/0) > /sys/arctrl/udp_port_pass_on_hit
echo (1/0) > /sys/arctrl/tcp_port_pass_on_hit
```

where,

0 - Discard the packet in case of hit in the table. Pass the packet in case of miss.

1 - Pass the packet in case of hit in the table. Discard the packet in case of miss.

c. Configure TCP control bit flags

This flag is used to configure a bit mask for TCP packets which will cause a wakeup of the system. This must be considered with TCP port filter rules.

```
echo 003f > /sys/arctrl/tcp_flag_mask
```

If no user defined rules are configured then system will be waken-up for the following types of packets actions:

```
TCP State (SYN/FIN/RST)      Wake-up
TCP Data (Push/Ack)          Wake-up
UDP packet                   Wake-up
Default                      Wake-up
```

4. Configure wakeup rules which will be offloaded to the ucode

```
./ar_wakeup_tool ar_wakeup_rule_cfg
```

5. Configuring MPIC timer (Optional configuration)

MPIC timer is used to wake the system after the specified time interval in seconds:

```
echo (time interval in secs) > /sys/devices/system/mpic/timer_wakeup
```

To avoid time difference, always run this command with:

```
echo mem > /sys/power/state
```

6. Getting into deep sleep state

```
echo mem > /sys/power/state
```

Configuration of hosts on Traffic Generator

- Create an IPv4 host on the traffic generator on port connected to DUT i.e. fm1-gb3
  1. Configure the IPv4 address as 192.85.2.100 on the host
- Create an IPv6 host on the traffic generator on port connected to DUT i.e. fm1-gb3
  1. Configure the IPv6 address as 2001::2 on the host
- Configure a IPv4/IPv6 UDP stream with the same port numbers as given in UDP filter table
- Configure a IPv4/IPv6 TCP stream with the same port numbers as given in TCP filter table
- Configure an IP stream with the same protocol as given in IP protocol filter table

Run Test

1. Ping from the host to the configured IPv4/IPv6. Ping response will be successful.
2. Send UDP packet which can cause wakeup or will be dropped (depends on action configured).
3. Send TCP packet which can cause wakeup or will be dropped (depends on action configured).
4. Send IP protocol packet which can cause wakeup or will be dropped (depends on action configured).

8.3 Power Management

8.3.1 Power Management User Manual

Linux SDK for QorIQ Processors

Description

QorIQ Processors have features to minimize power consumption at several different levels. All processors support a sleep mode (LPM20). Some processors, such as T1040, LS1021, LS1046, also support a deep sleep mode (LPM35).

The following power management features are supported on various QorIQ processors:

- Dynamic power management
- Shutting down unused IP blocks
- Cores support low power modes (such as PW15)
- Processors enter low power state (LPM20, LPM35)
  - LPM20 mode: most part of processor clocks are shut down

- LPM35 mode: power is removed to cores, cache and IP blocks of the processor such as DIU, eLBC, PEX, eTSEC, USB, SATA, eSDHC etc.

- CPU hotplug: If cores are down at runtime, they will enter low power state.

The wake-up event sources caused quitting from low power mode are listed as below:

- Wake on LAN (WoL) using magic packet
- Wake by MPIC timer or FlexTimer
- Wake by Internal and external interrupts

For more information on a specific processor, refer to processor Reference Manual.

Kernel Configure Tree View Options

For ARM platforms

Kernel Configure Tree View Options	Description
`Power management options  -->` `  [*] Suspend to RAM and standby`	Enable sleep feature

`Device Drivers  --->`
`  SOC (System On Chip) specific Drivers  --->`
`  [*]  Layerscape Soc Drivers`
`  [*]  FTM alarm driver`
	Enable the FTM alarm (FlexTimer module) driver

`CPU Power Management  --->`
`CPU Idle  --->`
`[*] CPU idle PM support`
`[*]  Ladder governor (for periodic timer tick)`
`-*-  Menu governor (for tickless system)`
`  ARM CPU Idle Drivers  --->`
`  [*] Generic ARM/ARM64 CPU idle Driver`
	Enable the CPU Idle driver

Table continues on the next page...

Compile-time Configuration Options

Linux Framework	Hardware Feature	Platform	Kernel Config
Suspend	LPM20	LS1012, LS1021, LS1046	CONFIG_SUSPEND
	wake by Flextimer	LS1012, LS1021, LS1046	CONFIG_FTM_ALARM
CPU idle	PW15	LS1012, LS1021, LS1046	CONFIG_ARM_CPUIDLE

Device Tree Binding

Property	Type	Description
fsl,#rcpm-wakeup-cells	unsigned int	the number of cells in "rcpm-wakeup" except the pointer to "rcpm"
rcpm-wakeup	unsigned int	require if the IP block can work as a wakeup source

For processors integrated RCPM

```
rcpm: rcpm@1ee2000 {
    compatible = "fsl,ls1046a-rcpm", "fsl,qoriq-rcpm-2.1";
    reg = <0x0 0x1ee2000 0x0 0x1000>;
    fsl,#rcpm-wakeup-cells = <1>;
};

ftm0: ftm0@29d0000 {
    compatible = "fsl,ftm-alarm";
    reg = <0x0 0x29d0000 0x0 0x10000>;
    interrupts = <0 86 0x4>;
    big-endian;
    rcpm-wakeup = <&rcpm 0x0 0x20000000>;
    status = "okay";
};
```

Refer to the Linux document: Documentation/devicetree/bindings/soc/fsl/rcpm.txt

Source Files

The source files are maintained in the Linux kernel source tree.

Source File	Description
drivers/soc/fsl/layerscape/rcpm.c	the RCPM driver needed by the sleep feature
drivers/soc/fsl/layerscape/ftm_alarm.c	the FTM timer driver worked as a wakeup source
drivers/cpuidle/cpuidle-arm.c	the cpuidle driver for ARM core

Verification in Linux

- Cpuidle Driver

  The cpuidle driver can switch CPU state according to the idle policy (governor). For more information, please see "Documentation/cpuidle/sysfs.txt" in kernel source code.

```
/* Check the cpuidle driver which is currently used. */
# cat /sys/devices/system/cpu/cpuidle/current_driver

/* Check the following directory to see the detailed statistic information of each state on
each CPU. */
/sys/devices/system/cpu/cpu0/cpuidle/state0/
/sys/devices/system/cpu/cpu0/cpuidle/state1/
```

- Sleep and Wake up by FTM timer

```
/* Start a FTM timer. It will trigger an interrupt to wake up the system in 5 seconds. */
echo 5 > /sys/devices/platform/soc/29d0000.ftm0/ftm_alarm && echo mem > /sys/power/state
```

Supporting Documentation

- QorIQ processor reference manuals

8.3.2 CPU Frequency Switching User Manual

Linux SDK for QorIQ Processors

Abbreviations and Acronyms

DFS: Dynamic Frequency Scaling

Description

QorIQ Processors support DFS (Dynamic Frequency Switching) feature, also known as CPU Frequency Switch, which can change the frequency of cores dynamically.

For more information on a specific processor, refer to processor Reference Manual.

Kernel Configure Tree View Options	Description

CPU Power Management -->
  CPU Frequency scaling -->
  [*] CPU Frequency scaling
  <*> CPU frequency translation statistics
  Default CPUFreq governor (userspace) -->
  -*- 'userspace' governor for userspace frequency scaling
  ARM CPU frequency scaling drivers -->
  <*> CPU frequency scaling driver for Freescale QorIQ SoCs
	Enable the CPU frequency driver

Compile-time Configuration Options

Linux Framework	Hardware Feature	Platform	Kernel Config
cpufreq	DFS	ALL	CONFIG_CPU_FREQ, CONFIG_CPU_FREQ_DEFAULT_GOV_USERSPACE
cpufreq	DFS	Layerscape	CONFIG_QORIQ_CPUFREQ

User Space Application

Simply using command "cat" and "echo" can verify this feature.

Device Tree Binding

Property	Type	Status	Description
#clock-cells	unsigned int	Required	The number of cells in a clock-specifier
clocks	handle	Required	Clock source handle
compatible	String	Required	Compatible strings
reg	unsigned int	Required	register address range

Table continues on the next page...

```
clockgen: clocking@1ee1000 {
    compatible = "fsl,ls1012a-clockgen";
    reg = <0x0 0x1ee1000 0x0 0x1000>;
    #clock-cells = <2>;
    clocks = <&sysclk>;
};
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Table continued from the previous page...

Source File	Description
drivers/cpufreq/qoriq_cpufreq.c	CPU frequency scaling driver for qoriq chips

Verification in Linux

- CPU frequency mode

```
In order to test the CPU frequency scaling feature, we need to enable the CPU frequency feature
on the
menuconfig and choose the USERSPACE governor.
You can learn more about CPU frequency scaling feature by referring to the kernel documents.
They all are put under Documentation/cpu-freq/ directory.
For example: all the information about governors is put in Documentation/cpu-freq/governors.txt.

Test step:
1. list all the frequencies a core can support (take cpu 0 for example) :
 # cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
1199999 599999 299999 799999 399999 199999 1066666 533333 266666

2.  check the CPU's current frequency
# cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
1199999

3. change the CPU's frequency we expect:
# echo 799999 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed

You can check the CPU's current frequency again to confirm if the frequency transition is
```

```
successful.

Please note that if the frequency you want to change to doesn't support by current CPU, kernel
will
round up or down to one CPU supports.
```

8.3.3 Thermal Management User Manual

Description

The thermal management function is based on TMU (Thermal Monitoring Unit).

The driver sets two thresholds for management function. If the CPU temperature crosses the first one (75 C for LS2080. 85 C for other platforms), the driver will trigger CPU frequency limitation auto-scaling according to the temperature trend; If the CPU temperature crosses the second one (85 C for LS2080, 95 C for other platforms, critical for core) the driver will shut down the system.

User could also get current temperature through sysfs interface.

Specifications

Target boards:	T1040RDB, T1042RDB, T1023RDB, T1024RDB, LS1021ATWR, LS1043ARDB, LS2080ARDB.
Operating system:	Linux 3.12+

Kernel Configure Tree View Options (For PowerPC platform)

Kernel Configure Tree View Options	Description

`Platform support  --->`
`    CPU Frequency scaling  --->`
`        PowerPC CPU frequency scaling drivers  --->`
`            <*> CPU frequency scaling driver for NXP QorIQ SoCs`
	Enable CPUfreq driver.

`Device Drivers  --->`
`    [*] Generic Thermal sysfs driver  --->`
`        [*] generic cpu cooling support`
`        [*] Freescale QorIQ Thermal Monitoring Unit`
	Enable thermal management framework, cpu cooling device support and QorIQ thermal driver.

Kernel Configure Tree View Options (For ARM platform)

Kernel Configure Tree View Options	Description

`CPU Power Management  --->`
`    CPU Frequency scaling  --->`
`        ARM CPU frequency scaling drivers  --->`
`            <*> CPU frequency scaling driver for NXP QorIQ SoCs`
	Enable CPUfreq driver.

`Device Drivers  --->` `    [*] Generic Thermal sysfs driver  --->`	Enable thermal management framework, cpu cooling device support and QorIQ thermal driver.

Table continued from the previous page...

Kernel Configure Tree View Options	Description
[*] generic cpu cooling support [*] Freescale QorIQ Thermal Monitoring Unit	

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_QORIQ_CPUFREQ	y/n	n	Enable QorIQ CPUfreq driver
CONFIG_THERMAL	y/m/n	n	Enable thermal management support
CONFIG_CPU_THERMAL	y/m/n	n	Enable cpu cooling device support
CONFIG_QORIQ_THERMAL	y/m/n	n	Enable QorIQ thermal driver

Device Tree Binding

```
tmu: tmu@f0000 {
        compatible = "fsl,qoriq-tmu";
        reg = <0xf0000 0x1000>;
        interrupts = <18 2 0 0>;
        fsl,tmu-range = <0x000a0000 0x00090026 0x0008004a 0x0001006a>;
        fsl,tmu-calibration = <0x00000000 0x00000025
                0x00000001 0x00000028
                0x00000002 0x0000002d
                0x00000003 0x00000031
                0x00000004 0x00000036
                0x00000005 0x0000003a
                0x00000006 0x00000040
                0x00000007 0x00000044
                0x00000008 0x0000004a
                0x00000009 0x0000004f
                0x0000000a 0x00000054

                0x00010000 0x0000000d
                0x00010001 0x00000013
                0x00010002 0x00000019
                0x00010003 0x0000001f
                0x00010004 0x00000025
                0x00010005 0x0000002d
                0x00010006 0x00000033
                0x00010007 0x00000043
                0x00010008 0x0000004b
                0x00010009 0x00000053

                0x00020000 0x00000010
                0x00020001 0x00000017
                0x00020002 0x0000001f
                0x00020003 0x00000029
                0x00020004 0x00000031
                0x00020005 0x0000003c
                0x00020006 0x00000042
                0x00020007 0x0000004d
                0x00020008 0x00000056
```

```
                0x00030000 0x00000012
                0x00030001 0x0000001d>;
};
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/thermal/qoriq_thermal.c	QorIQ thermal driver.

Verification in Linux

There are two parts for verification: management and monitor.

[Management:]

1. When CPU temperature cross the first threshold, CPU frequency may be reduced by changing frequency limitation, use the following command to check the current frequency:

```
~$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
```

2. When CPU temperature cross the first threshold, system will shutdown.

[Monitor:]

```
You can manually read the thermal interfaces in sysfs:
~$ cat /sys/class/thermal/thermal_zone0/temp
35000
# This means the current temperature is 35 C.
```

8.3.4 System Monitor

8.3.4.1 Power Monitor User Manual

Description

There are two methods currently we can use to measure the power consumption which are called online and offline power monitoring respectively. The difference between them is that offline power monitoring support measuring power consumption during sleep or deep sleep.

The Power Monitor can be supported on P4080DS/P5020DS/P5040DS/T4240QDS board.

This User guide uses the T4240QDS board as an example.

Online Power Monitoring

The Lm-sensors tool (download from http://dl.lm-sensors.org/lm-sensors/releases) will be used to read the power/ temperature from on-boards sensors. The drivers vary from sensor to sensor. Basically they would be INA220, ZL6100 and ADT7461 etc.

The device driver support either a built-in kernel or module loading.

Kernel Configure Tree View Options

Option	Description

`Device Drivers --->`
` <*> Hardware Monitoring support  --->`
`    <*>   Texas Instruments INA219 and compatibles`
	Enables INA220

`Device Drivers --->`
`    [*]   Enable compatibility bits for old user-space`
`    <*>   I2C device interface`
`    [*]   Autoselect pertinent helper modules`
`          I2C Hardware Bus support  --->`
`              <*> MPC107/824x/85xx/512x/52xx/83xx/86xx`
	Enables I2C block device driver support

`Device Drivers --->`
`      <*>   I2C bus multiplexing support`
`          Multiplexer I2C Chip support  --->`
`              <*> Philips PCA954x I2C Mux/switches`
	Enables I2C bus multiplexing PCA9547

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_I2C_MPC	y/n	y	Enable I2C bus protocol
SENSORS_INA2XX	y/n	y	Enables INA220
CONFIG_I2C_MUX_PCA954x	y/n	y	Enables I2C multiplexing PCA9547

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	"Philips,pca9547" for pca9547
reg	integer	Required	reg = <0x77>
compatible	String	Required	"ti,ina220" for ina220
reg	integer	Required	reg = <the i2c address of ina220>

```
Default node:
       i2c@118000 {
            pca9547@77 {
                compatible = "philips,pca9547";
                reg = <0x77>;
                #address-cells = <1>;
                #size-cells = <0>;

                channel@2 {
                    #address-cells = <1>;
                    #size-cells = <0>;
                    reg = <0x2>;

                    ina220@40 {
                        compatible = "ti,ina220";
```

```
                        reg = <0x40>;
                        shunt-resistor = <1000>;
                    };

                    ina220@41 {
                        compatible = "ti,ina220";
                        reg = <0x41>;
                        shunt-resistor = <1000>;
                    };

                    ina220@44 {
                        compatible = "ti,ina220";
                        reg = <0x44>;
                        shunt-resistor = <1000>;
                    };

                    ina220@45 {
                        compatible = "ti,ina220";
                        reg = <0x45>;
                        shunt-resistor = <1000>;
                    };

                    ina220@46 {
                        compatible = "ti,ina220";
                        reg = <0x46>;
                        shunt-resistor = <1000>;
                    };

                    ina220@47 {
                        compatible = "ti,ina220";
                        reg = <0x47>;
                        shunt-resistor = <1000>;
                    };
                };
            };
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/i2c/muxes/i2c-mux-pca954x.c	PCA9547 driver
drivers/hwmon/ina2xx.c	ina220 driver

Test Procedure

Do the following to validate under the kernel

1. The bootup information is displayed:

```
......
i2c /dev entries driver
mpc-i2c ffe118000.i2c: timeout 1000000 us
mpc-i2c ffe118100.i2c: timeout 1000000 us
mpc-i2c ffe119000.i2c: timeout 1000000 us
mpc-i2c ffe119100.i2c: timeout 1000000 us
i2c i2c-0: Added multiplexed i2c bus 6
i2c i2c-0: Added multiplexed i2c bus 7
```

```
i2c i2c-0: Added multiplexed i2c bus 8
i2c i2c-0: Added multiplexed i2c bus 9
i2c i2c-0: Added multiplexed i2c bus 10
i2c i2c-0: Added multiplexed i2c bus 11
i2c i2c-0: Added multiplexed i2c bus 12
i2c i2c-0: Added multiplexed i2c bus 13
pca954x 0-0077: registered 8 multiplexed busses for I2C mux pca9547
ina2xx 8-0040: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0041: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0045: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0046: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0047: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0044: power monitor ina220 (Rshunt = 1000 uOhm)
......
```

2.

```
# sensors
ina220-i2c-8-40
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.08 V
in1:          +1.06 V
power1:       35.00 W
curr1:       +32.77 A

ina220-i2c-8-41
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.01 V
in1:          +0.01 V
power1:       60.00 mW
curr1:        +6.62 A

ina220-i2c-8-45
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.01 V
in1:          +0.00 V
power1:       60.00 mW
curr1:        +8.20 A

ina220-i2c-8-46
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.01 V
in1:          +0.01 V
power1:       60.00 mW
curr1:        +6.60 A

ina220-i2c-8-47
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.01 V
in1:          +0.02 V
power1:      140.00 mW
curr1:        +7.40 A

ina220-i2c-8-44
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.00 V
in1:          +1.51 V
power1:        1.86 W
curr1:        +1.23 A
```

NOTE

Please make sure to include the "sensors" command in your rootfs

Offline Power Monitoring

Inside the FPGA of some NXP QorIQ (PowerPC) reference boards is a microprocessor called the General Purpose Processor (GSMA). Running on the GSMA is the Data Collection Manager (DCM), which is used to periodically read and tally voltage, current, and temperature measurements from the on-board sensors. You can use this feature to measure power consumption while running tests, without having the host CPU perform those measurements.

This method support measuring power consumption when kernel is in sleep or deep sleep status. It gets the average power value of period from the time DCM starts to the time it ends.

Module Loading

The device driver support either kernel built-in or module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`[*] Misc devices  --->`
`    <*>   Freescale Data Collection Manager (DCM) driver`
	Enables DCM driver

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_FSL_DCM	y/n	y	Enable DCM module

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	"fsl,t4240qds-fpga", "fsl,fpga-qixis"
reg	Integer	Required	reg = <3 0 0x300>

```
Default node:
       ifc: localbus@ffe124000 {
             board-control@3,0 {
             compatible = "fsl,t4240qds-fpga", "fsl,fpga-qixis";
             reg = <3 0 0x300>;
       };
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/misc/fsl_dcm.c	DCM driver

Test Procedure

Do the following to validate under the Kernel:

1. The bootup information is displayed:

```
......
Freescale Data Collection Module is installed.
......
```

2. Start measuring measure power

```
# echo 1 > /sys/devices/platform/fsl-dcm.0/control
```

3. Stop measuring power

```
#echo 0 > /sys/devices/platform/fsl-dcm.0/control
```

4. Display the average power consumption

```
#cat /sys/devices/platform/fsl-dcm.0/result

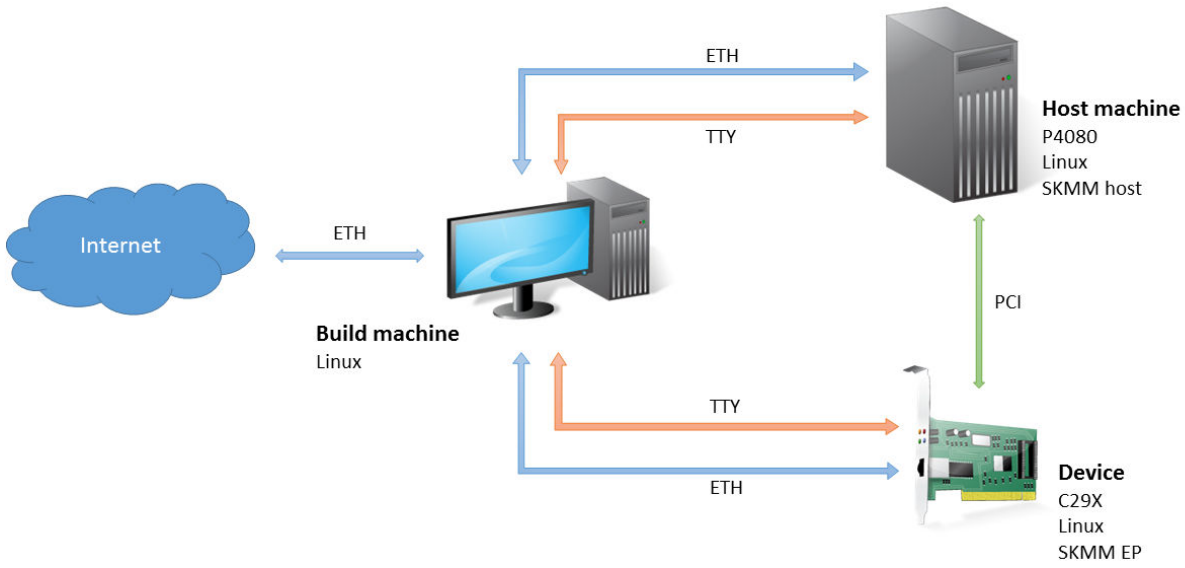
Name                        Average
====================        ================
CPU voltage:                1068    (mV)
CPU current:                25910   (mA)
DDR voltage:                1348    (mV)
DDR current:                740     (mA)
CPU temperature:            38      (C)
```

8.3.4.2 Thermal Monitor User Manual

Description

The Temperature Monitoring function is provided by the chip ADT7461.

This driver exports the values of Temperature to SYSFS. The user space lm-sensors tools can get and display these values.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
Device Drivers ---> [*] Hardware Monitoring support ---> [*] National Semiconductor LM90 and compatibles	Enable thermal monitor chip driver like ADT7461.
Device Drivers ---> <*> I2C bus multiplexing support ---> Multiplexer I2C Chip support ---> <*> Philips PCA954x I2C Mux/switches	Enable I2C PCA954x multiplexer support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_HWMON	y/m/n	n	Enable Hardware Monitor
CONFIG_SENSORS_LM90	y/m/n	n	Enable ATD7461 driver
CONFIG_I2C_MUX	y/m/n	n	Enable I2C bus multiplexing support
CONFIG_I2C_MUX_PCA954x	y/m/n	n	Enable PCA954x driver

Device Tree Binding

```
    adt7461@4c {
        compatible = "adi,adt7461";
        reg = <0x4c>;
    };

    pca9547@77 {
        compatible = "philips,pca9547";
        reg = <0x77>;
    };
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/hwmon/hwmon.c	Linux hwmon subsystem support
drivers/hwmon/lm90.c	ADT7461 chip driver
drivers/i2c/i2c-mux.c	I2C bus multiplexing support
drivers/i2c/muxes/pca954x.c	PCA954x chip driver

Verification in Linux

There are two ways to get temperature results.

```
1. You can manually read the thermal interfaces in sysfs:
~$ ls /sys/class/hwmon/hwmon1/devices
alarms            temp1_crit        temp1_min_alarm   temp2_max_alarm
driver            temp1_crit_alarm  temp2_crit        temp2_min
hwmon             temp1_crit_hyst   temp2_crit_alarm  temp2_min_alarm
modalias          temp1_input       temp2_crit_hyst   temp2_offset
name              temp1_max         temp2_fault       uevent
power             temp1_max_alarm   temp2_input       update_interval
subsystem         temp1_min         temp2_max

~$ cat /sys/class/hwmon/hwmon1/devices/temp1_input
29000

2. You can use lm_sensors tools as follows.
~ # sensors

adt7461-i2c-1-4c
```

```
Adapter: MPC adapter
temp1:       +34.0 C  (low  =  +0.0 C, high = +85.0 C)
                      (crit = +85.0 C, hyst = +75.0 C)
temp2:       +48.5 C  (low  =  +0.0 C, high = +85.0 C)
                      (crit = +85.0 C, hyst = +75.0 C)
```

lm_sensors is integrated into Yocto file system by default. If there is no "sensors" command in your rootfs just add lmsensors-sensors package and build your own rootfs using Yocto:

```
IMAGE_INSTALL += "lmsensors-sensors"
```

8.3.4.3 Web-based System Monitor User Guide

Monitors the health of a system using a web browser in real time.

Description

Web-based System Monitor is a tool for monitoring the health of your system using a web browser in real time. The following procedures will guide you to setup the system monitor.

Kernel requirements

The raw data of this monitor system is collected from hardware monitor chips. So before you setup this monitor system you should enable the hwmon subsystem and drivers of monitor chips in the kernel. Kernel configure details are listed below.

Kernel Configure Tree View Options	Description
Device Drivers ---> [*] Hardware Monitoring support ---> [*] National Semiconductor LM90 and compatibles [*] Texas Instruments INA219 and compatibles	Enable monitor chip drivers like ADT7461(ADT7481)/INA220, etc.
Device Drivers ---> <*> I2C bus multiplexing support ---> Multiplexer I2C Chip support ---> <*> Philips PCA954x I2C Mux/switches	Enable I2C PCA954x multiplexer support

Some monitor chips may not be included in the device tree. In this case you could add the device manually. Take adt7461 on T4240QDS as an example: the monitor is attached to I2C multiplexer PCA954x channel 3 in address 0x43. T4240QDS has 4 I2C controllers so the channel index of multiplexer start from 4 (represent the channel 0). ADT7461 is connected to channel 3 which indexed as 7. Use the flowing command to add the device to kernel:

```
~$ echo adt7461 0x4c > /sys/bus/i2c/devices/i2c-7/new_device
```

Rootfs requirements

You could use the fsl-image-full rootfs in which all packages needed are included.

Or you can build your own rootfs using Yotco. Please follow the steps below.

1. Add following package group to your rootfs recipes like fsl-image-core.bb:

```
IMAGE_INSTALL += "packagegroup-fsl-monitor"
```

2. If you are using ramdisk boot please add following settings to local.conf to get enough space for monitor systems:

```
IMAGE_ROOTFS_EXTRA_SPACE = "100000"
```

NOTE

This will add 100000KB (100MB) more space to rootfs for monitor database. Each sensor needs about 10MB more space for logging raw data.

Setting up system monitor

The monitor system will be setup automatically. What you need to do is to make sure that the network on board is working. Then you can monitor the system via any web browser by visiting: http://your.ip.address/senspix/sensors.cgi. This results page will refresh itself for every 10 seconds.

If you need to re-setup the system you could enter /usr/rrd directory and run:

```
$ make clean
$ make
```

NOTE

The System Monitor only works when system time is right. So you should guarantee that.

How to configure the system monitor

The monitor results you see is based on the configuration file: "monitor.conf". It's automatically generated by scanning the hwmon subsystem. You could manually modify it too. Here is how:

1. Each line of this configuration file represents one monitor curve. It contains four fields formatted as follows:

   ```
   SENSDEV:MONITOR_TERM:DURATION:DESCRIPTION
   ```

   SENSDEV: The sensor data can be monitored from /sys/class/hwmon/ interfaces. Each sensor has a corresponding folder distinguished by hwmon# like hwmon0 or hwmon1. SENSDEV is the folder name.

   MONITOR_TERM: This is the item you want to read like temp1 or temp2 etc.

   DURATION: This is how long you want to see the results. You can set minute/hour/day here.

   DESCRIPTION: This DESCRIPTION will show up on the result picture helping you to understand the contents of the curve.

2. You could add/remove/resort the configuration file. After modifying it, you could enter the /usr/rrd directory and run:

   ```
   $ make config
   ```

3. Then the monitor results will be updated to what you configured.

Run demo

We also provide scripts to cycle the system through different PM low power states to form a out-of-box demo for PM features. Please enter /usr/pm_demo directory and simply run:

```
$ ./pm_demo.sh
```

The output of the script will state the current PM features. It helps you to understand the system monitor results better.

NOTE

The demo could be terminated by CTRL-C and will apply the default PM features back.

8.4 Real Time Application Note

8.4.1 Application Note on Real Time

Introduction

Baseband use-cases like 3G/4G have strict timelines to accomplish some particular jobs. Real Time (RT) feature available in the operating system aims at creating an environment to meet these time critical processing requirements. There are various approaches available for providing RT feature.

NXP uses Linux PREEMPT_RT patch (also known as RT patch) to meet these requirements. PREEMPT_RT patch can be pulled from kernel.org git repository

For more information regarding PREEMPT_RT refer to kernel.org wiki page

PREEMPT_RT Patch in SDK

PREEMPT_RT patch is applied in the kernel available with this SDK. By default, RT feature is disabled in all the defconfigs of this SDK, except the one mentioned in later section.

Please note that, once one enable RT feature, throughput-performance of the system might decrease (and this decrease is expected as per design of RT).

Support Status

Hardware:	Currently supported only for P4080DS, B4860qds, TWR-LS1021A, LS1012A, LS2080
Software:	Linux (with PREEMPT_RT patch), (SMP-Linux: non KVM)

Compilation Option

Default Kernel Defconfig

Kernel Configure Option

Tree view

For enabling RT in defconfig using "make menuconfig" for kernel

Kernel Configure Tree View Options	Description

`Kernel options  --->`
`  Preemption Model (Fully Preemptible Kernel (RT))  --->`
`  (X) Fully Preemptible Kernel (RT)`
	These options enable RT in Linux.

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value to enable RT	Description
CONFIG_SLAB	y/n	y	Enable SLAB Support

Table continues on the next page...

Table continued from the previous page...

Option	Values	Default Value to enable RT	Description
CONFIG_HIGHMEM	y/n	n	Disable highmem support
CONFIG_PREEMPT_RT_FULL	y/n	y	Enable Full RT support

Device Tree Binding

No RT specific changes required

Verification in Linux

To verify that PREEMPT_RT Patch is applied and RT is enabled in Linux configuration after Linux has booted, check Linux version on Linux prompt, one should see pattern “PREEMPT RT” in the version string. Eg:

```
root@bsc913x:~# uname -a

Linux bsc913x 3.8.13-rt6+ #52 PREEMPT RT Wed May 22 12:26:51 IST 2013 ppc GNU/Linux
```

Test Tool

RT-tests suit contains various test applications like cyclictest, hackbench, etc to measure latencies and induce various test loads. It come as package in yocto (can be build in rootfs) or can be downloaded from kernel.org git repository.

NOTE

PREEMPT_RT feature provides RTT (Real Time throttling) feature. For details on RTT, refer to “Documentation/scheduler/sched-rt-group.txt” in Linux source code. RTT might get triggered in case of heavy traffic leading to high latency. It can be disabled by:

```
[root@bsc913x]#echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

Supporting Documentation

https://rt.wiki.kernel.org

Known Limitations

On non-DPAA SoCs like LS1021, LS1021 with PREEMPT_RT enabled kernel, while running IPv4 forwarding benchmarking scenarios (including the IPSec forwarding benchmark) or pi-stress test-case, which are inherently CPU and traffic intensive, sometimes RCU stalls dumps were observed in dmesg. There is no negative impact on occurrence of this RCU stall dumps, the system continues to run as before.

(For IPv4, IPSEC forwarding test-cases, refer to Benchmark Reproducibility Guides in SDK documentation). Note: TCP/UPD Termination testing does not cause this issue even though CPU utilization is 100%.

Workaround: The above kind of use-case for a real-time device is unlikely. If the above kind of scenario is expected to occur in a device, the device should have some provisions to reduce the CPU load by throttling the low priority jobs.Or limit the traffic. Or use TCP/UDP terminating type traffic

8.5 C29X Card Use Cases

8.5.1 C29x SKMM User Manual

8.5.1.1 Introduction

SKMM is a software application for managing (generating, storing and retrieving) keys in a secure manner. This kind of applications is knwon in the technical literature as Secure Key Management software. The security is provided by the use of a C293 Crypto Coprocessor, which connects to the host via the PCIe bus and is optimized for public key operations.

SKMM is made up of two software components:

- skmm-ep - a userspace application running on the C293 device
- skmm-host - a driver used on the host which can be either P4080 development board or x86 machine

The SKMM supports only one of the following hardware configurations:

- P4080 host with C293 device
- x86 host with C293 device

This document details the procedure to configure, build, run and tests for the SKMM software application.

Prerequistes

Before building the images download the latest NXP SDK on a Linux machine which will be called the build machine. For the rest of the documentation the base directory for the sdk will be specified as <sdk-dir>.

The code for both components is located in the <sdk-dir>/sources directory.

8.5.1.2 Hardware setup

This section details the hardware setup and configuration used.

SKMM Setup for PPC Based Host Systems

This section describes the basic setup used for configuring SKMM as a host-device application.



The device is a C293 board on which a user space application SKMM-EP (SKMM end-point) runs over a custom built Linux image. The device is connected via PCI to the Host, which in this setup is a P4080 machine, that contains the SKMM host module which is a kernel driver that exposes through different interfaces the cryptographic capabilities of the C293 board. Both the device and the host run a custom Linux image that is prepared on the build machine via the bitbake utility. A Linux image consists of three files: the kernel image, the device tree file and the ramdisk file. These files are transfered to the corresponding destination machine using the TFTP protocol via the ETH link and then booted from the UBoot console. The serial link (TTY) serves as a means to access the UBoot consoles from the build machines and later, after boot, the Linux console. To access the images from the destination machines the build machine should have a TFTP server installed and the build files should be accessible from the TFTP shared folder. Additionally the build machine should be able to access the internet in order to retrieve third party software that is bundled in with the Linux image created.

SKMM Setup for X86 Hosts

When using an Intel x86 machine as a host the usual setup is the one in the image above. Here the host is also used as the build machine. For this setup the host Linux image is the one provided by the chosen distro and the SKMM host driver is built against the distos kernel header files. Currently we support only Ubuntu 12.04 and CentOS 3.10.0-229.7.2 64 bits.

C293 card switch settings

This section provides the details applicable to the NXP C293 PCIe development platform. These switch settings may not be valid for other version of cards.

For SKMM mode booting from NOR flash, the switches are set as the following:

```
SW4: 01011000
SW5: 11110000
SW6: 00001111
SW7: 10000111
SW8: 00000001
```

1 = OFF, 0 = ON.

Here is the corresponding frequency of different blocks:

```
CPU: 800 MHz
CCB: 400 MHz
DDR: 400 MHz (800MT/s)
IFC: 100 MHz
```

Device Detection

For x86 you can validate the presence of the C293 PCIe Card by running "sudo lspci -vvv" command. Please note that you must run the command as root to have a usefull degree of verbosity. Otherwise not enough information may be presented to the console. The following is an example output:

```
$sudo lspci -vvv
04:00.0 Power PC: Freescale Semiconductor Inc Device 0800 (rev 10) (prog-if 01)
   Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B-
DisINTx+
   Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR-
INTx-
   Latency: 0, Cache Line Size: 32 bytes
   Interrupt: pin A routed to IRQ 111
   Region 0: Memory at fbb00000 (32-bit, non-prefetchable) [size=1M]
   Region 1: Memory at a0000000 (32-bit, prefetchable) [size=512K]
   Capabilities: [44] Power Management version 3
            Flags: PMEClk- DSI- D1+ D2+ AuxCurrent=0mA PME(D0+,D1+,D2+,D3hot+,D3cold-)
            Status: D0 NoSoftRst- PME-Enable- DSel=0 DScale=0 PME-
   Capabilities: [4c] Express (v2) Endpoint, MSI 00
            DevCap:   MaxPayload 256 bytes, PhantFunc 0, Latency L0s <64ns, L1 <1us
                      ExtTag- AttnBtn- AttnInd- PwrInd- RBE+ FLReset-
            DevCtl:   Report errors: Correctable- Non-Fatal+ Fatal+ Unsupported+
                      RlxdOrd+ ExtTag- PhantFunc- AuxPwr- NoSnoop+
                      MaxPayload 128 bytes, MaxReadReq 512 bytes
            DevSta:   CorrErr- UncorrErr- FatalErr- UnsuppReq- AuxPwr- TransPend-
            LnkCap:   Port #0, Speed 5GT/s, Width x4, ASPM L0s, Latency L0 <2us, L1 unlimited
                      ClockPM- Surprise- LLActRep- BwNot-
            LnkCtl:   ASPM Disabled; RCB 64 bytes Disabled- Retrain- CommClk-
                      ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
            LnkSta:   Speed 5GT/s, Width x4, TrErr- Train- SlotClk- DLActive- BWMgmt- ABWMgmt-
            DevCap2: Completion Timeout: Range ABC, TimeoutDis+
            DevCtl2: Completion Timeout: 50us to 50ms, TimeoutDis-
            LnkCtl2: Target Link Speed: 2.5GT/s, EnterCompliance- SpeedDis-, Selectable De-emphasis: -6dB
                     Transmit Margin: Normal Operating Range, EnterModifiedCompliance- ComplianceSOS-
                     Compliance De-emphasis: -6dB
            LnkSta2: Current De-emphasis Level: -6dB
   Capabilities: [88] MSI: Enable+ Count=1/16 Maskable- 64bit+
            Address: 00000000fee00000  Data: 4056
   Capabilities: [100 v1] Advanced Error Reporting
            UESta:    DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC-
UnsupReq- ACSViol-
            UEMsk:    DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC-
UnsupReq- ACSViol-
            UESvrt:   DLP+ SDES- TLP- FCP+ CmpltTO- CmpltAbrt- UnxCmplt- RxOF+ MalfTLP+ ECRC-
UnsupReq- ACSViol-
            CESta:    RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr-
            CEMsk:    RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr+
            AERCap:   First Error Pointer: 00, GenCap+ CGenEn- ChkCap+ ChkEn-
   Kernel driver in use: FSL-Crypto-Driver
```

On P4080 board, device detection may be verified from u-boot logs:

```
Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0c, vBank: 0
SERDES Reference Clocks: Bank1=100MHz Bank2=125MHz Bank3=125MHz
I2C:   ready
SPI:   ready
DRAM:  Initializing....using SPD
Detected UDIMM EBJ21EE8BAFA-DJ-E
Detected UDIMM EBJ21EE8BAFA-DJ-E
2 GiB left unmapped
4 GiB (DDR3, 64-bit, CL=9, ECC on)
     DDR Controller Interleaving Mode: cache line
     DDR Chip-Select Interleaving Mode: CS0+CS1
Testing 0x00000000 - 0x7fffffff
Testing 0x80000000 - 0xffffffff
Remap DDR 2 GiB left unmapped

POST memory PASSED
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 2 MiB enabled
SRIO1: disabled
SRIO2: disabled
MMC:  FSL_SDHC: 0
EEPROM: Invalid ID (ff ff ff ff)
PCIe1: Root Complex, no link, regs @ 0xfe200000
PCIe1: Bus 00 - 00
PCIe2: Root Complex, x4 gen2, regs @ 0xfe201000
 02:00.0     - 1957:0800 - Processor
PCIe2: Bus 01 - 02
PCIe3: disabled
In:    serial
Out:   serial
Err:   serial
Net:   Fman1: Uploading microcode version 106.2.14
PHY reset timed out
Fman2: Uploading microcode version 106.2.14
PHY reset timed out
FM1@DTSEC2, FM1@TGEC1, FM2@TGEC1
```

8.5.1.3 Build Procedure

The build process is done via the bitbake tool. For C293 and P4080 SKMM software components (skmm-ep for C293 and skmm-host driver for P4080) are included in the images. For x86 building and running is done on the same machine and has some particularities that will be provided in the corresponding chapter.

C293 - Firmware (building and deployment)

Building C293 images cand be done in the following sequence of steps:

- **Generate build directories**

  This step is only required if the build directory has not been previously generated.

```
$ cd <sdk-devel>
$ source fsl-setup-env -m c293pcie -j 4 -t 4 -l
```

- **Clear configuration**

This step is only required if P4080 images are generated using the same sdk directory and a P4080 image has been previously built.

```
$ bitbake -c cleansstate virtual/kernel
```

- **Configure the kernel**

```
$ bitbake -c menuconfig virtual/kernel
```

Disable hardware crypto devices, MMC/SD/SDIO card support and enable userspace I/O driver:

```
Cryptographic API  ---
    [ ]   Hardware crypto devices  ----

Device Drivers  --->
    < > MMC/SD/SDIO card support ----
    <*> Userspace I/O drivers  --->
        <*> Freescale SEC support
```

- **Build image**

```
$ bitbake fsl-image-core
```

- **Copy binaries**

The resulting binaries will be copied to the tftp server directory called <tftpboot-dir>

```
$ cd tmp/deploy/images/c293pcie/
$ cp uImage-c293pcie.bin <tftpboot-dir>
$ cp uImage-c293pcie.dtb <tftpboot-dir>
$ cp fsl-image-core-c293pcie.ext2.gz.u-boot <tftpboot-dir>
$ cp u-boot-c293pcie.bin <tftpboot-dir>
```

P4080 - Linux Kernel SKMM Host Driver (build and deployment)

Build P4080 images with the following sequence of steps:

- **Generate build directories**

This step is only required if the build directory has not been previously generated.

```
$ cd <sdk-devel>
$ source fsl-setup-env -m p4080ds -j 4 -t 4 -l
```

- **Clear configuration**

This step is only required if C293 images are generated using the same sdk directory and a C293 image has been previously build.

```
$ bitbake -c cleansstate virtual/kernel
```

- **Build image**

```
$ bitbake fsl-image-core
```

- **Copy binaries**

The resulting binaries will be copied to the tftp server directory called <tftpboot-dir>

```
$ cd tmp/deploy/images/p4080ds
$ cp uImage-p4080ds.bin <tftpboot-dir>
$ cp uImage-p4080ds.dtb <tftpboot-dir>
$ cp fsl-image-core-p4080ds.ext2.gz.u-boot <tftpboot-dir>
$ cp u-boot-P4080DS.bin <tftpboot-dir>
$ cp rcw/R_PPPNN_0x5/rcw_rc_1500mhz.bin <tftpboot-dir>
```

x86 - Linux Kernel SKMM Host Driver (build and deployment)

It is assumed that the build machine is also the target for the purpose of testing. Cross-building for x86 machines is out of the scope of this manual. Driver compilation has been verified for CentOS 3.10.0-229.7.2 64bit and Ubuntu 12.04. Please use the same version to avoid dependency related errors.

NOTE

If the SDK directory doesn't contain the skmm-host director (which is the case when downloading the ISO image from the NXP site) you can get it from the following freescale repository:

```
$ git clone git://git.freescale.com/ppc/sdk/skmm-host.git -b sdk-
v2.0.x
```

Follow the following sequence of steps to generate the skmm-host and cryptodev kernel modules and OpenSSL userspace application:

- **Apply patches**

  Paching is required before building and runnign skmm-host driver. The patches are located at <sdk-dir>/sources/skmm-host/crypto-patches. These patches were created on kernel version 3.11.0-15 and if their application fails, apply them manually.

```
$ cd /lib/modules/$(uname -r)/build/include/linux/
$ patch -p0 < <sdk-dir>/sources/skmm-host/crypto-patches/crypto-add-pkc-support.patch
$ cd /lib/modules/$(uname -r)/build/include/crypto/
$ patch -p0 < <sdk-dir>/sources/skmm-host/crypto-patches/algapi-add-pkc-support.patch
```

- **Build driver**

```
$ cd <sdk-dir>/sources/skmm-host
```

- **Configure driver**

  Edit Makefile:

```
EXTRA_PKC = 'y'
```

- **Build driver**

```
$ make ARCH=x86
```

Building OpenSSL

Building OpenSSL is only needed for functional tests and not required for performance tests.

For PPC hosts (P4080) both cryptodev and OpenSSL are built during kernel build through the bitbake recipe. After booting the kernel the user can simply insert the cryptodev.ko module and run the OpenSSL commands as both resources are available at boot time.

For x86 hosts additional steps must be followed.

- **Build cryptodev**

```
$ cd <sdk-dir>/sources/cryptodev
$ make
$ sudo make modules_install
$ sudo install -D crypto/cryptodev.h /usr/include/crypto/cryptodev.h
```

- **Build OpenSSL**

```
$ cd <sdk-dir>/sources/openssl
$ make clean
$ ./config --prefix=/usr/local --openssldir=/usr/local/openssl shared -DHAVE_CRYPTODEV
$ make
$ sudo make install
$ sudo ldconfig
```

8.5.1.4 Run SKMM

For SKMM to work properly the following startup order must be followed:

- boot C293 device
- run SKMM-EP
- boot host
- insert SKMM driver

Boot C293 device - Boot Kernel Image and Run SKMM EP Application

- **Boot C293 Kernel Image**

  Booting cand be done in one of two ways:

  1. **Flash boot**

```
setenv bootargs "root=/dev/ram rw ramdisk_size=800000 console=ttyS0,115200 cache-
sram=0xfffa00000,0x40000 $otherbootargs"
setenv linuxfile <tftpboot-dir>/uImage-c293pcie.bin
setenv fdtfile <tftpboot-dir>/uImage-c293pcie.dtb
setenv ramdiskfile <tftpboot-dir>/fsl-image-core-c293pcie.ext2.gz.u-boot

setenv loadaddr 1000000

setenv linuxaddr ec000000
setenv fdtaddr ecf00000
setenv ramdiskaddr ed000000

protect off all
tftp $loadaddr $linuxfile && erase $linuxaddr +$filesize && cp.b $loadaddr $linuxaddr $filesize
tftp $loadaddr $fdtfile && erase $fdtaddr +$filesize && cp.b $loadaddr $fdtaddr $filesize
tftp $loadaddr $ramdiskfile && erase $ramdiskaddr +$filesize && cp.b $loadaddr $ramdiskaddr
$filesize
protect on all

setenv bootcmd 'bootm $linuxaddr $ramdiskaddr $fdtaddr'
saveenv

boot
```

2. **TFTP boot**

```
setenv bootargs "root=/dev/ram rw ramdisk_size=800000 console=ttyS0,115200 cache-
sram=0xfffa00000,0x40000 $otherbootargs"
setenv linuxfile <tftpboot-dir>/uImage-c293pcie.bin
setenv fdtfile <tftpboot-dir>/uImage-c293pcie.dtb
setenv ramdiskfile <tftpboot-dir>/fsl-image-core-c293pcie.ext2.gz.u-boot

setenv linuxaddr 1000000
setenv fdtaddr c00000
setenv ramdiskaddr 2000000

setenv bootcmd 'tftp $linuxaddr $linuxfile && tftp $fdtaddr $fdtfile && tftp $ramdiskaddr
$ramdiskfile && bootm $linuxaddr $ramdiskaddr $fdtaddr'
saveenv

boot
```

Run SKMM-EP Application

After booting the board using the SDK image, login as root (no password) and run SKMM application:

```
$ flash_erase /dev/mtd3 0 0
$ skmm fffa00000
```

At this point the skmm application will wait for host driver handshake.

You can specify the l2sram address by giving one argument to skmm application to avoid the platform SRAM LAW setting conflict with other LAWs, for example: skmm fffa00000. The size of l2sram is 1M Bytes. You can use command reginfo checking LAWs setting in u-boot prompt. If you do not give any argument or give more than one argument, it will use the default address 0xfffa00000.

Deploy SKMM-Host Kernel Driver on Host Machine

- **Boot Host Kernel Image**

```
setenv bootargs "root=/dev/ram rw ramdisk_size=800000 console=ttyS0,115200 $otherbootargs"
setenv linuxfile <tftpboot-dir>/uImage-p4080ds.bin
setenv fdtfile <tftpboot-dir>/uImage-p4080ds.dtb
setenv ramdiskfile <tftpboot-dir>/fsl-image-core-p4080ds.ext2.gz.u-boot

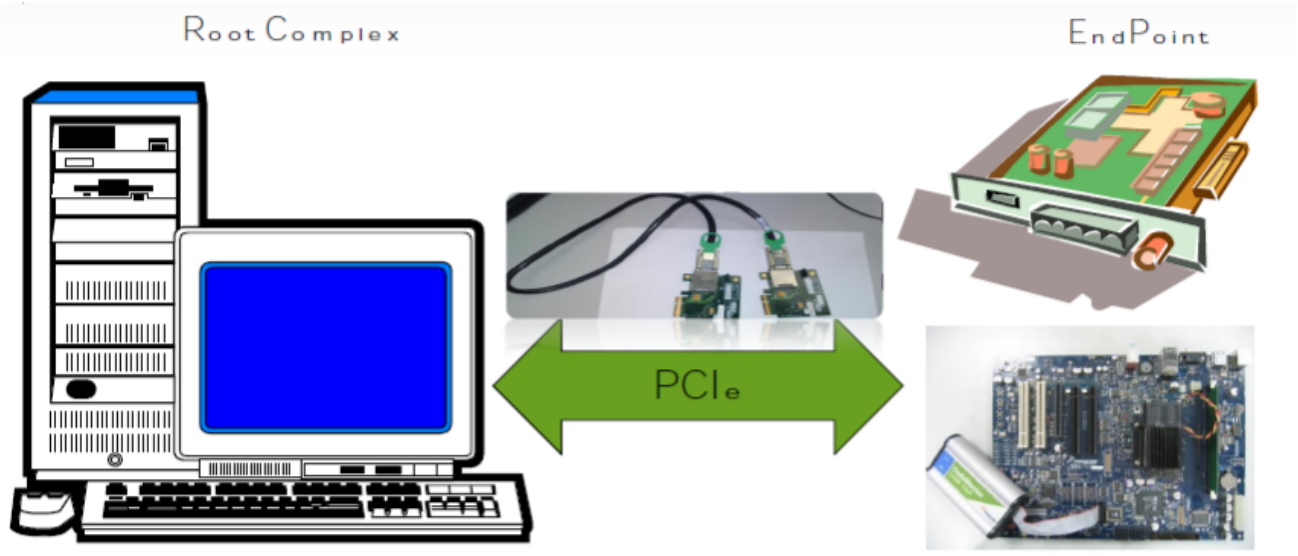
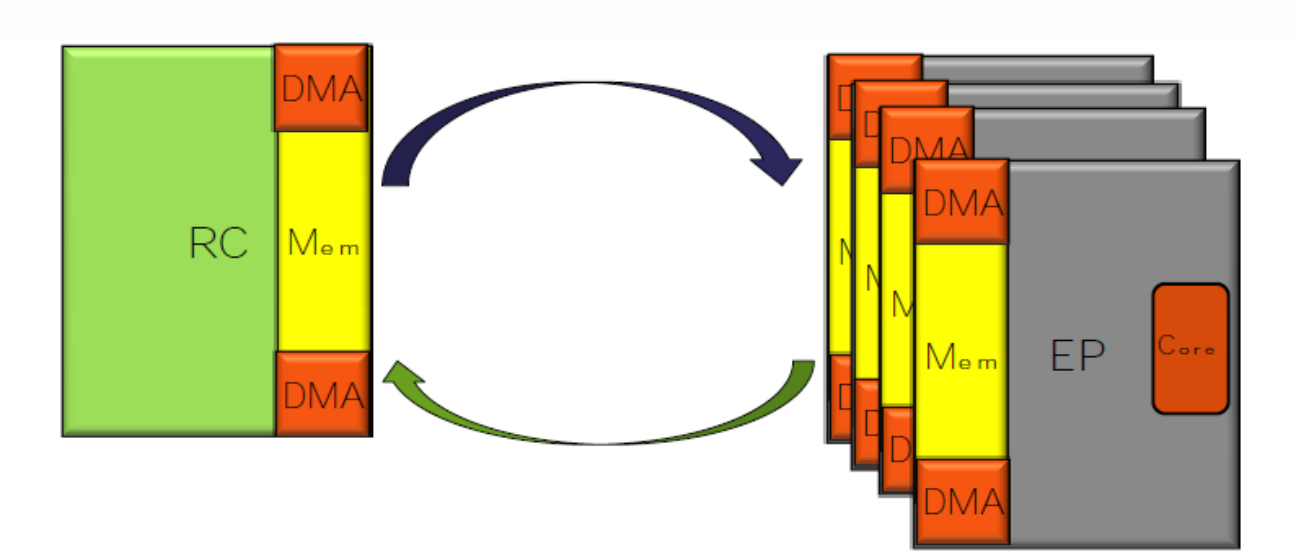
setenv linuxaddr 1000000
setenv fdtaddr c00000
setenv ramdiskaddr 2000000

setenv bootcmd 'tftp $linuxaddr $linuxfile && tftp $fdtaddr $fdtfile && tftp $ramdiskaddr
$ramdiskfile && bootm $linuxaddr $ramdiskaddr $fdtaddr'
saveenv

boot
```

Login with user "root" and no password.

Insert SKMM Driver

When inserting the SKMM driver a configuration file called "crypto.cfg" can be provided that is located at <sdk-dir>/sources/skmm-host. It has a pseudo-xml structure and can configure the firmware path and the ring pairs. Each ring contains the following details:

depth	Ring size.
affinity	Sec engine affinity.
priority	Relative priority of rings.
order	Whether the ring processing is ordered or not.

Users can create a maximum of 6 rings. If the ring configuration needs to be changed, the driver must be unloaded and reloaded for the new configuration to take effect. The command ring is disabled when HIGH_PERF_MODE flag is set in Makefile.

After configuration step insert the skmm and cryptodev driver:

```
insmod /lib/modules/$(uname -r)/extra/fsl_skmm_crypto_offload_drv.ko dev_config_file=/etc/skmm/
skmm_crypto.cfg
```

After all the operations have been completed, the following prints can be seen in kernel logs

```
$ dmesg |tail
[FSL-CRYPTO-OFFLOAD-DRV] DevId:1 DEVICE IS UP
cryptodev: driver 2.0 loaded.
```

Driver is up now, and it can be used to send crypto operations.

Deploy SKMM-Host Kernel Driver on x86 Host Machine

For x86 machines the kernel has already been booted. The only thing left to do is to just insert the SKMM-Host kernel driver. Go to the skmm-host directory, where it has been built, and run the following commands:

```
$ cd <sdk-dir>/sources/skmm-host
$ insmod fsl_skmm_crypto_offload_drv.ko dev_config_file=./crypto.cfg
```

Known Issues

- the skmm driver will not work if hyperthreading is disabled on the host kernel (for example by means of using boot argument smt-enabled=off)

8.5.1.5 Function Test

OpenSSL

Standard openSSL does not support SKMM offload in cryptodev engine. A version of OpenSSL tweaked for SKMM has been included and maintained in the build for demonstration purposes.

Before performing these tests first you have to build cryptodev and openSSL. Please refer to chapter "Build Procedure", sub-chapter "Building OpenSSL". After this step you are required to insert the cryptodev module. For PPC systems the cryptodev modules comes bundled with the kernel images so inserting the kernel is done via modprobe:

```
$ modprobe cryptodev.ko
```

For x86 systems you have go to the cryptodev directory, where it has been built, and to manually insert the module:

```
$cd <sdk-dir>/sources/cryptodev.ko
$sudo insmod cryptodev.ko
```

Supported tests

SKMM supports RSA 1k 2k 4k and DSA 1k 2k 4k crypto operations. Only the sequence of commands presented bellow has been tested.

Run the following commands on the host side for function test:

```
# echo "1234567890" > a
```

RSA 1k:

```
# openssl genrsa -out rsa.key_1024 1024 -engine cryptodev
# openssl rsa -in rsa.key_1024 -pubout -out rsa.pub_1024
# openssl rsautl -encrypt -in a -pubin -inkey rsa.pub_1024 -out e_a -engine cryptodev
# openssl rsautl -decrypt -in e_a -inkey rsa.key_1024 -out aa -engine cryptodev
# diff a aa
```

RSA 2k:

```
# openssl genrsa -out rsa.key_2048 2048 -engine cryptodev
# openssl rsa -in rsa.key_2048 -pubout -out rsa.pub_2048
# openssl rsautl -encrypt -in a -pubin -inkey rsa.pub_2048 -out e_a -engine cryptodev
# openssl rsautl -decrypt -in e_a -inkey rsa.key_2048 -out aa -engine cryptodev
# diff a aa
```

RSA 4k:

```
# openssl genrsa -out rsa.key_4096 4096 -engine cryptodev
# openssl rsa -in rsa.key_4096 -pubout -out rsa.pub_4096
# openssl rsautl -encrypt -in a -pubin -inkey rsa.pub_4096 -out e_a -engine cryptodev
# openssl rsautl -decrypt -in e_a -inkey rsa.key_4096 -out aa -engine cryptodev
# diff a aa
```

DSA 1k:

```
# openssl dsaparam -out dsa.param_1024 1024
# openssl gendsa -out dsa.key_1024 -engine cryptodev dsa.param_1024
# openssl pkey -in dsa.key_1024 -pubout -out dsa.pub_1024
# openssl pkeyutl -sign -in a -inkey dsa.key_1024 -out s_a -engine cryptodev
# openssl pkeyutl -verify -in a -sigfile s_a -pubin -inkey dsa.pub_1024 -engine cryptodev
```

DSA 2k:

```
# openssl dsaparam -out dsa.param_2048 2048
# openssl gendsa -out dsa.key_2048 -engine cryptodev dsa.param_2048
# openssl pkey -in dsa.key_2048 -pubout -out dsa.pub_2048
# openssl pkeyutl -sign -in a -inkey dsa.key_2048 -out s_a -engine cryptodev
# openssl pkeyutl -verify -in a -sigfile s_a -pubin -inkey dsa.pub_2048 -engine cryptodev
```

DSA 4k:

```
# openssl dsaparam -out dsa.param_4096 4096
# openssl gendsa -out dsa.key_4096 -engine cryptodev dsa.param_4096
# openssl pkey -in dsa.key_4096 -pubout -out dsa.pub_4096
# openssl pkeyutl -sign -in a -inkey dsa.key_4096 -out s_a -engine cryptodev
# openssl pkeyutl -verify -in a -sigfile s_a -pubin -inkey dsa.pub_4096 -engine cryptodev
```

8.5.1.6 Performance Test

Introduction

SKMM supports kernel space performance test.

Before any performance test the private keys in the blob mtd bloc 3 on c293pcie board must be removed:

```
c293# flash_erase /dev/mtd3 0 0
```

SKMM test script "c29x_skmm_perf_profile.sh" is used to do performance test in kernel space. The test script can be found at:

```
skmm-host/perf
```

The test script may be run either by invoking the test script from its original directory or by copying it to a directory visible by the operating system:

```
sudo cp skmm-host/perf/c29x_skmm_perf_profile.sh /usr/bin
```

In the examples below we will asume the ladder strategy.

Supported Tests

SKMM supports RSA 1K, 2K, 4K both public and private operations and DSA 1K, 2K, 4K sign and verify operations. Please note that DSA signature test must be run before DSA verification test. This is how the test are run:

- RSA public key 1K, 2K, 4K:

```
HOST# sh c29x_skmm_perf_profile.sh RSA_PUB_OP_1K -m 0xe0 -t 2 -s 3
HOST# sh c29x_skmm_perf_profile.sh RSA_PUB_OP_2K -m 0xe0 -t 2 -s 3
HOST# sh c29x_skmm_perf_profile.sh RSA_PUB_OP_4K -m 0xe0 -t 2 -s 3
```

- RSA private key 1K, 2K, 4k:

```
HOST# sh c29x_skmm_perf_profile.sh RSA_PRV_OP_1K -m 0xe0 -t 2 -s 3
HOST# sh c29x_skmm_perf_profile.sh RSA_PRV_OP_2K -m 0xe0 -t 2 -s 3
HOST# sh c29x_skmm_perf_profile.sh RSA_PRV_OP_4K -m 0xe0 -t 2 -s 3
```

- DSA signature 1K, 2K, 4K:

```
HOST# sh c29x_skmm_perf_profile.sh DSA_SIGN_TEST_1K -m 0xe0 -t 2 -s 3
HOST# sh c29x_skmm_perf_profile.sh DSA_SIGN_TEST_2K -m 0xe0 -t 2 -s 3
HOST# sh c29x_skmm_perf_profile.sh DSA_SIGN_TEST_4K -m 0xe0 -t 2 -s 3
```

- DSA verify 1K, 2K, 4K:

```
HOST# sh c29x_skmm_perf_profile.sh DSA_VERIFY_TEST_1K -m 0xe0 -t 2 -s 3
HOST# sh c29x_skmm_perf_profile.sh DSA_VERIFY_TEST_2K -m 0xe0 -t 2 -s 3
HOST# sh c29x_skmm_perf_profile.sh DSA_VERIFY_TEST_4K -m 0xe0 -t 2 -s 3
```

Usage

The "c29x_skmm_perf_profile.sh" accepts the following configuration parameters:

Argument	Value	Description
-m	cpu mask	Test threads will be created on the CPUs masked by the value in above option.
-t	thread per cpu	Number of threads per cpu.
-s	Test duration in seconds.	The length of the test specified in seconds
-r	Test enqueue-dequeue count.	The length of the test specified in the actual number of enqueued-dequeued tests

Table continues on the next page...

Notes:

- The test can be stopped at any time by pressing CTRL+C
- If both configuration parameters '-s' and '-r' are given the 's' has precedence.
- For a full list of configuration parameters run:

```
$ c29x_skmm_perf_profile.sh --help
```

- On success the test script outputs something akin to:

```
******** Result ***********
Test Name:
Host CPU Frequency:
# job finished successfully:
Per job in us:
Total jobs in 1 sec:
```

8.5.2 PK Calculator User Guide

8.5.2.1 Hardware setup

C293PCIe Switch Settings

This section provides details applicable to the NXP C293 PCIe development platform. These switch settings may not be valid for other version of cards. For PKC mode, the switches are set as following:

```
SW4: 01011000
SW5: 11110000
SW6: 00001111
SW7: 00000111
SW8: 00000001

1:OFF 0:ON
```

In particular, the following switches affect the behaviour of the board. See "C29x PCIe Card Quick Start Guide" for different configurations.

```
SW7[1]   - ON (0) - hold-off boot
SW7[2-4] - ON ON ON  (000) for PCIe-x4 5GHz or
           ON ON OFF (001) for PCIe-x4 2.5GHz
```

Device Detection

Run "sudo lspci" or "sudo lspci –vvv" to check the presence of the C293 card. Depending on the actual lspci binary beeing used (busybox or true lspci), you can find more details about the installed board:

```
01:00.0 Power PC: NXP Semiconductors Inc Device 0800 (rev 10) (prog-if 01)
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR-
FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR-
<PERR- INTx-
        Latency: 0, Cache Line Size: 64 bytes
        Interrupt: pin A routed to IRQ 47
        Region 0: Memory at f7c00000 (32-bit, non-prefetchable) [size=1M]
        Region 1: Memory at f0000000 (32-bit, prefetchable) [size=1M]
        Capabilities: [44] Power Management version 3
                Flags: PMEClk- DSI- D1+ D2+ AuxCurrent=0mA PME(D0+,D1+,D2+,D3hot+,D3cold-)
                Status: D0 NoSoftRst- PME-Enable- DSel=0 DScale=0 PME-
        Capabilities: [4c] Express (v2) Endpoint, MSI 00
                DevCap: MaxPayload 256 bytes, PhantFunc 0, Latency L0s <64ns, L1 <1us
                        ExtTag- AttnBtn- AttnInd- PwrInd- RBE+ FLReset-
                DevCtl: Report errors: Correctable- Non-Fatal- Fatal- Unsupported-
                        RlxdOrd- ExtTag- PhantFunc- AuxPwr- NoSnoop+
                        MaxPayload 256 bytes, MaxReadReq 512 bytes
                DevSta: CorrErr+ UncorrErr- FatalErr- UnsuppReq+ AuxPwr- TransPend-
                LnkCap: Port #0, Speed 5GT/s, Width x4, ASPM L0s, Exit Latency L0s <2us, L1
unlimited
                        ClockPM- Surprise- LLActRep- BwNot-
                LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- CommClk-
                        ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
                LnkSta: Speed 5GT/s, Width x4, TrErr- Train- SlotClk- DLActive- BWMgmt- ABWMgmt-
                DevCap2: Completion Timeout: Range ABC, TimeoutDis+, LTR-, OBFF Not Supported
                DevCtl2: Completion Timeout: 50us to 50ms, TimeoutDis-, LTR-, OBFF Disabled
                LnkCtl2: Target Link Speed: 2.5GT/s, EnterCompliance- SpeedDis-
                         Transmit Margin: Normal Operating Range, EnterModifiedCompliance-
ComplianceSOS-
                         Compliance De-emphasis: -6dB
                LnkSta2: Current De-emphasis Level: -6dB, EqualizationComplete-,
EqualizationPhase1-
                         EqualizationPhase2-, EqualizationPhase3-, LinkEqualizationRequest-
        Capabilities: [88] MSI: Enable+ Count=1/16 Maskable- 64bit+
                Address: 00000000fee0f00c  Data: 41a2
        Capabilities: [100 v1] Advanced Error Reporting
                UESta:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC-
UnsupReq- ACSViol-
                UEMsk:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC-
UnsupReq- ACSViol-
                UESvrt: DLP+ SDES- TLP- FCP+ CmpltTO- CmpltAbrt- UnxCmplt- RxOF+ MalfTLP+ ECRC-
UnsupReq- ACSViol-
                CESta:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr+
                CEMsk:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr+
                AERCap: First Error Pointer: 00, GenCap+ CGenEn- ChkCap+ ChkEn-
        Kernel driver in use: FSL-Crypto-Driver
```

8.5.2.2 Build Procedure

PKC host driver Build Configuration

These are the build time parameters that can be changed in the Makefile.

Variable name	Default Value	Description
DEBUG_PRINT	N	Defines the print level. If set to 'y', debug level prints will be enabled.
INFO_PRINT	N	Defines the print level. If set to 'y', info level prints will be enabled.
DEBUG_DESC	N	Enable/disable descriptors debug-print
ENHANCE_KERNEL_TEST	N	Enhance PKC kernel test performance by disabling kernel test schedule

Prerequisites

- NXP SDK
- For x86 targets, two kernel patches must be applied before building the PKC driver. These are header patches provided with the PKC driver sources. For powerpc or arm , no kernel patching is necessary.

Build Driver for PowerPC or ARM

The binaries for pkc-host, cryptodev and openssl packages are delivered with fsl-image-core. No additional steps are necessary.

Build Driver for x86

This section details the manual build procedure for x86 targets. It is assumed that the build machine is also the target for the purpose of testing. Cross-building for x86 machines is out of the scope of this manual. After installing the SDK, sources of pkc-host, cryptodev and openssl will be available in:

```
<sdk_dir>/sources/
```

Before building the pkc-host module, as root, apply PKC patches to kernel headers. Kernel rebuild is not necessary after patching:

```
# cd /lib/modules/$(uname -r)/build/include/linux/
# patch -p0 < path_to/pkc-host/crypto-patches/crypto-add-pkc-support.patch
# cd /lib/modules/$(uname -r)/build/include/crypto/
# patch -p0 < path_to/pkc-host/crypto-patches/algapi-add-pkc-support.patch
```

Build pkc-host driver

```
$ cd path_to/pkc-host
$ make
$ sudo make modules_install
$ sudo install -D images/pkc-firmware.bin /etc/crypto/pkc-firmware.bin
$ sudo install perf/c29x_driver_perf_profile.sh /usr/bin/c29x_driver_perf_profile.sh
```

Build cryptodev and openssl for user-space testing:

```
$ cd path_to/cryptodev-module
$ make
$ sudo make modules_install
$ sudo install -D crypto/cryptodev.h /usr/include/crypto/cryptodev.h
```

```
$ cd path_to/openssl
$ make clean
$ ./config --prefix=/usr/local --openssldir=/usr/local/openssl shared -DHAVE_CRYPTODEV
```

```
$ make
$ sudo make install
```

```
To refresh the linker cache with newly added openssl library:
$ sudo ldconfig
```

8.5.2.3 Driver Configuration

This list shows the parameters can be provided to the driver:

Parameter	Default value	Description
napi_poll_count	1	Specifies how many times the worker threads should be polling for new jobs after the stream of jobs stopped. Workers stay on a stream until it dries and then either return or poll for new jobs as specified by this parameter

Examples:

```
modprobe fsl_crypto_offload_drv.ko
```

```
modprobe fsl_crypto_offload_drv.ko napi_poll_count=2
```

8.5.2.4 Performance Test

The PKC driver is delivered together with a kernel performance test. This test allows a quick check of the driver functionality and raw performance. For user-space testing, NXP SDK has PKC enabled versions of openssl and cryptodev. The upstream versions of openssl and cryptodev do not have PKC offloading support and can't be used for PKC driver testing.

Kernel test

The test consist of a user shell script and a driver block that communicates with this script. This user-space shell script sends commands to the kernel driver though /sysfs. The script must be run as root and the driver must be loaded before running the script. Without any parameters, the script will print a help screen:

```
$ c29x_driver_perf_profile.sh
or
$ c29x_driver_perf_profile.sh --help
```

Usage:

```
$ sudo c29x_driver_perf_profile.sh <test_name> [option]...
```

Option	Value	Description
-m	cpu mask	Test threads will be created on the CPUs identified by the mask
-t	threads per cpu	Number of threads per cpu (defaults to one)
-s	seconds	Test execution time
-r	job count.	The test will execute the specified number of jobs

NOTE

Tests can be stopped at any time with CTRL-C. If both -s and -r are specified, then -s is selected by the test script. If the driver was built with ENHANCE_KERNEL_TEST=y the load on the CPU will increase and the test script may miss CTRL-C signals. In this case, always run the script with -s option to stop it after a certain time.

NOTE

The test script accepts options only in the order given by the help screen. Options in arbitrary order are not supported.

Example:

```
$ sudo modprobe fsl_crypto_offload_drv.ko

$ sudo c29x_driver_perf_profile.sh RSA_PUB_OP_2K
$ sudo c29x_driver_perf_profile.sh RSA_PUB_OP_1K -m 0xf -t 1 -s 10
$ sudo c29x_driver_perf_profile.sh RSA_PUB_OP_1K -m 0xf -t 1 -r 100000
```

See the following list for available tests:

```
1. RSA_PUB_OP_1K
2. RSA_PUB_OP_2K
3. RSA_PUB_OP_4K
4. RSA_PRV_OP_1K
5. RSA_PRV_OP_2K
6. RSA_PRV_OP_4K
7. DSA_SIGN_TEST_1K
8. DSA_SIGN_TEST_2K
9. DSA_SIGN_TEST_4K
10. DSA_VERIFY_TEST_1K
11. DSA_VERIFY_TEST_2K
12. DSA_VERIFY_TEST_4K
13. DSA_SIGN_VERIFY_TEST
14. DSA_KEYGEN_TEST
15. ECDSA_KEYGEN_TEST
16. DH_KEYGEN_TEST
17. ECDH_TEST
18. ECDSA_VERIFY_TEST
19. ECDSA_SIGN_TEST
20. ECP_SIGN_TEST_256
21. ECP_VERIFY_TEST_256
22. ECP_SIGN_TEST_384
23. ECP_VERIFY_TEST_384
24. ECP_SIGN_TEST_521
25. ECP_VERIFY_TEST_521
26. ECPBN_SIGN_TEST_283
27. ECPBN_VERIFY_TEST_283
28. ECPBN_SIGN_TEST_409
29. ECPBN_VERIFY_TEST_409
30. ECPBN_SIGN_TEST_571
31. ECPBN_VERIFY_TEST_571
32. DH_TEST_1K
33. DH_TEST_2K
34. DH_TEST_4K
35. ECDH_KEYGEN_P256
36. ECDH_KEYGEN_P384
37. ECDH_KEYGEN_P521
38. ECDH_KEYGEN_B283
```

```
39. ECDH_KEYGEN_B409
40. ECDH_KEYGEN_B571
```

Openssl

For user-space tests with openssl, both pkc and cryptodev drivers must be loaded. Only RSA and DSA tests are supported with OpenSSL

```
$ sudo modprobe fsl_crypto_offload_drv.ko
$ sudo modprobe cryptodev.ko

$ openssl speed -multi 10 -elapsed rsa -engine cryptodev
```

8.6 PCIe DMA Test User Manual

8.6.1 Introduction to PCI DMA Test

Introduction

When NXP PCIE controller working in EP mode, it can be connected to a host such as x86 computer or PowerPC board, and works as a PCI device. The PCI DMA test application is for the scenario and is used to test PCI memory write performance. It contains two parts. One is PCI DMA EP application running on EP side, the other is PCI DMA host driver module running on host side.

Test Environment



Figure 188. Test environment

As the above figure shown, we can directly insert the PCI card into host or use PCI card and cable to connect host and EP board.

Host side supports x86 and PowerPC boards.

EP side supports P5020DS, T4240QDS.

Data processing



Figure 189. Data processing

As the above figure shown, EP may include multiple functions, each function is looked as a PCI device. EP application will initialize all the functions, allocate memory to store the data from host, and can write data to host. In host side, the driver module also allocates memory to store the data from EP, and can write data to the specified EP.

8.6.2 PCI DMA EP Application

Introduction

PCI DMA EP application is a user space application running on EP side. It initializes all the PCI functions of the specified PCI controller. For T4240QDS, the first PCI controller contains two physical functions and each physical function supports 64 virtual functions. Each function is looked as a PCI DMA device indexed by PF index and VF index, and is created three or four inbound windows.

1. Inbound window 0 is associated with BAR0 mapped to CCSR. Only enabled for Physical function.
2. Inbound window 1 is associated with BAR1, which is used for MSIX.
3. Inbound window 2 is associated with BAR2, which is used to store configuration information including RC command, EP status and test result.
4. Inbound window 3 is associated with BAR4, which is used to receive data from host.

The application can create a test thread on a specified PCI DMA device. Test thread will wait for host command and start performance test.

EP Kernel Image

PCI DMA EP application depends on DMA UIO driver and PCI EP VFIO driver. please make sure the two drivers are selected when compiling the EP kernel image.

```
 -> Device Drivers
     ->VFIO Non-Privileged userspace driver framework
         <*>   VFIO support for NXP PCI Endpoint devices

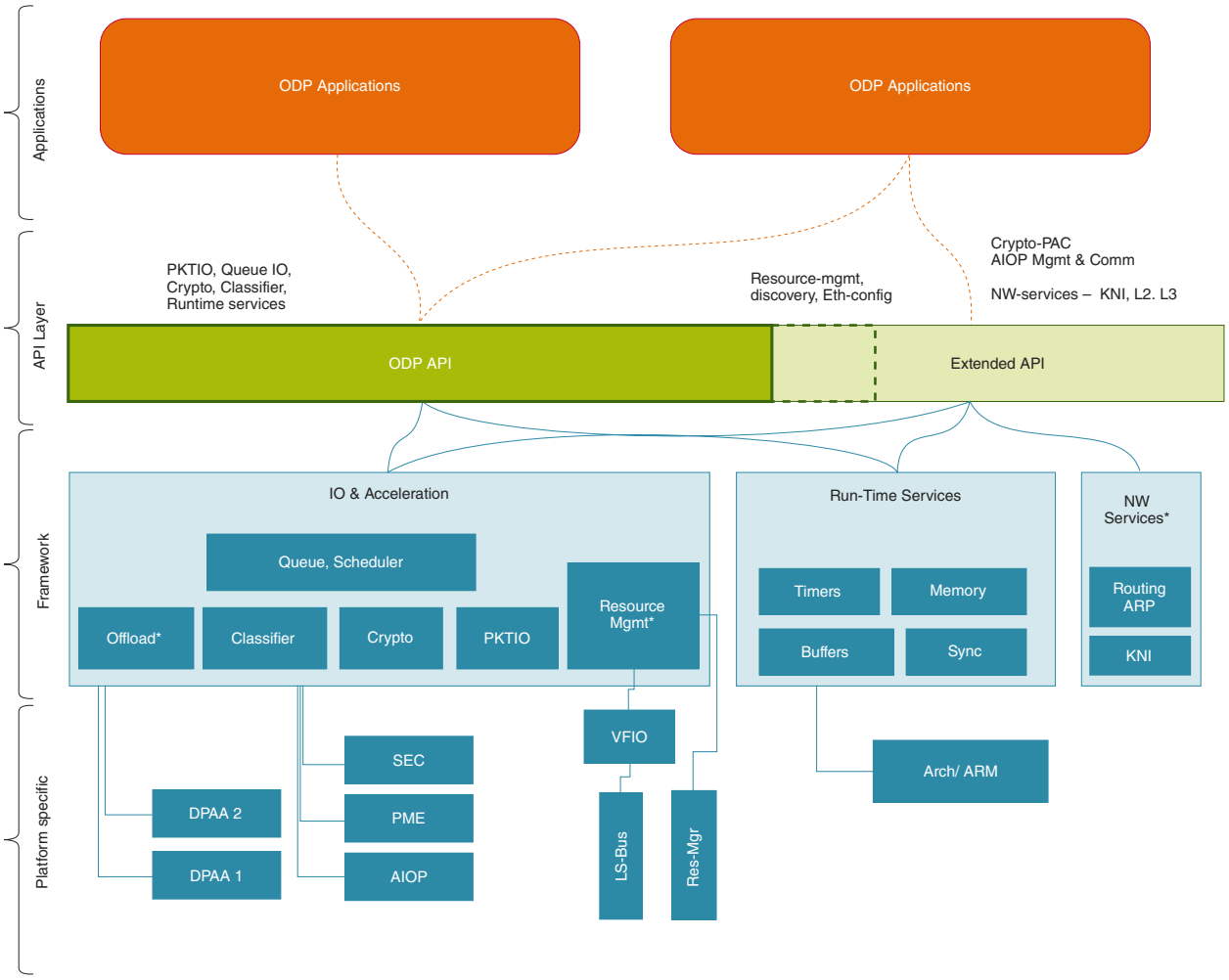
     -> Userspace I/O drivers
<*>   NXP DMA support
```

Compile EP Application

1. Use yacto to compile EP application.

   The detailed information can refer to SDK documents.

```
#> bitbake fsl-image-core
```

2. Directly Compile EP application

   a. Get the code and move them to sdk folder.

```
git clone git://git.am.freescale.net/gitolite/sdk/skmm-ep.git
git checkout remotes/origin/pciedma_multi_chan_perf -b pciedma_multi_chan_perf
```

   b. Configure the compile environment

```
lmh@lmh:~/work/sdk-devel-gerrit/skmm-ep$ source standalone-env -m t4240qds
```

   c. Compile code

```
lmh@lmh:~/work/sdk-devel-gerrit/skmm-ep$ make
 [CC] common.c          (lib:skmm_common)
 [AR] libskmm_common.a
 [CC] skmm_pci.c        (lib:skmm_pci)
 [AR] libskmm_pci.a
 [CC] skmm_sec.c        (lib:skmm_sec)
 [AR] libskmm_sec.a
 [CC] skmm_ddr.c        (lib:skmm_mem)
 [AR] libskmm_mem.a
 [CC] skmm_sec_blob.c   (lib:skmm_sec_blob)
 [AR] libskmm_sec_blob.a
 [CC] skmm_memmgr.c     (lib:skmm_memmgr)
 [AR] libskmm_memmgr.a
 [CC] skmm_uio.c        (lib:skmm_uio)
 [AR] libskmm_uio.a
 [AR] libskmm_cache.a
 [CC] process.c         (lib:skmm_process)
 [AR] libskmm_process.a
 [CC] setup.c           (lib:skmm_dma_mem)
 [CC] allocator.c       (lib:skmm_dma_mem)
 [CC] map.c             (lib:skmm_dma_mem)
 [AR] libskmm_dma_mem.a
 [CC] of.c              (lib:skmm_of)
 [AR] libskmm_of.a
 [CC] pci.c             (lib:skmm_usdpaa_pci)
 [AR] libskmm_usdpaa_pci.a
 [CC] pci_ep_vfio.c     (lib:skmm_pci_ep_vfio)
 [AR] libskmm_pci_ep_vfio.a
 [CC] dma_driver.c      (lib:skmm_dma)
 [AR] libskmm_dma.a
 [CC] trigger_msi_ep2rc.c (bin:trigger_msi_ep2rc)
 [LD] trigger_msi_ep2rc
 [CC] skmm.c            (bin:skmm)
 [CC] abstract_req.c    (bin:skmm)
 [CC] rsa.c             (bin:skmm)
 [CC] dsa.c             (bin:skmm)
 [CC] ecdsa.c           (bin:skmm)
 [CC] dh.c              (bin:skmm)
 [CC] ecdh.c            (bin:skmm)
```

```
  [LD] skmm
  [CC] pciep_dma.c      (bin:pciep_dma)
  [LD] pciep_dma
```

Run EP Application

The EP application is named pciep_dma and locates skmm-ep/bin folder. We can directly run this application on EP side, for example:

```
root@p5020ds:~# ./pciep_dma  0
Initialized pci0-pf0
pcidma>
```

The following commands are provided to perfom the PCI DMA performance test.

1. help: list the available command

   ```
   pcidma> help
   Available commands:
   list
   rm
   add
   dump
   info
   help
   ```

2. add : create a test thread on the specified PF/VF waiting the RC command to start the performance test .

   The command format:

   ```
   add [pf idx] [vf idx]
   ```

   For example:

   ```
   pcidma> add 0 0
   pcidma> Starting a pci0-pf0's test thread on cpu1
   ```

3. rm : remove the test thread

   The command format:

   ```
   rm [pf idx] [vf idx]
   ```

   For example:

   ```
   pcidma> rm 0 0
   Leaving pci0-pf0's test thread on cpu1
   ```

4. list : list all the running test thread

   The command format:

   ```
   list
   ```

   For example:

   ```
   pcidma> list
   pci0-pf0's test thread is runing on cpu0
   ```

5. info : display the inbound/outbound/Registers/configuration windows information.

   The command format:

```
 info [PF index][VF index]
```

   For example:

```
  pcidma> info 0 0
PCI EP device pci0-pf0 info:
 type: PF

Outbound windows:
Win0: cpu_addr:0x0 pci_addr:0x0 size:0x1000000000 attr:0x80044023  vfio_off:0x10000000000
Win1: cpu_addr:0xc00000000 pci_addr:0x100f1400000 size:0x400000 attr:0x80044015  vfio_off:
0x11000000000
Win2: cpu_addr:0xff8000000 pci_addr:0x0 size:0x10000 attr:0x8008800f  vfio_off:0x12000000000
Win3: cpu_addr:0x0 pci_addr:0x0 size:0x0 attr:0x0  vfio_off:0x13000000000
Win4: cpu_addr:0x0 pci_addr:0x0 size:0x1000000000 attr:0x44023  vfio_off:0x14000000000

Inbound windows:
Win0: cpu_addr:0xffe000000 pci_addr:0xe0000000 size:0x1000000  attr:0x80e44017  vfio_off:0x0
Win1: cpu_addr:0xe0000000 pci_addr:0xe1000000 size:0x2000  attr:0xa0f5500c  vfio_off:
0x1000000000
Win2: cpu_addr:0xe0002000 pci_addr:0xe1002000 size:0x1000  attr:0xa0f5500b  vfio_off:
0x2000000000
Win3: cpu_addr:0xe0400000 pci_addr:0xe1400000 size:0x400000  attr:0xa0f55015  vfio_off:
0x3000000000

Registers window:
cpu_addr:0xffe200000 size:0x1000 vfio_off:0x40000000000

 PCI configurations window:
cpu_addr:0x0 size:0x0 vfio_off:0x50000000000
pcidma>
```

6. dump : dump the Hexadecimal values of the registers, EP configuration local buffer and remote buffer on the function

   The command format:

```
dump [pf idx] [vf idx] [reg config msix local_buffer remote_buffer] [length]
```

   For example:

```
pcidma> dump 0 0 local_buffer
dump local_buffer:
00000000: 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5
00000004: 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5
00000008: 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5
0000000c: 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5
pcidma> dump 0 0 reg
dump reg:
00000000: 0x80000024 0x00000000 0x00000000 0x0013ffff
00000004: 0x0400ffff 0x00040028 0x00008000 0x00000000
00000008: 0x00000280 0x00000000 0x000cffff 0x00000000
0000000c: 0x00000000 0x00000000 0x00000000 0x00000000
pcidma>  dump 0 0 config
dump config:
00000000: 0x00000000 0x00000000 0x00000000 0x00000000
00000004: 0x00000000 0x00000000 0x00000000 0x00000000
```

```
00000008: 0x00000000 0x00000000 0x00000000 0x00000000
0000000c: 0x00000000 0x00000000 0x00000000 0x00000000
```

8.6.3 PCI DMA Host Driver Module

Introduction

PCI DMA host driver module is a PCI driver module running in the host side and is used to measure PCI memory write performance. It supports multiple PCI functions and SR-IOV, it an create a thread on the specified function to measure performance using host DMA or memcpy. It also can control EP device to start EP DMA and calculate the performance. The driver also provides some sysfs interface to change test settings such as packet length loop times and write direction (RC to EP or EP to RC).

Host Kernel Image

PCI DMA Host driver module depends on DMA driver and PCI IOV. Please make sure the two drivers are selected when compiling the host kernel image.

```
    -> Device Drivers
        -> DMA Engine support
           [*] Freescale Elo and Elo Plus DMA support (enable the option)
           [ ] Network: TCP receive copy offload (disable the option)

    ->Bus options                                                              [*]
PCI IOV support
```

Compile Host Module

1. Use yacto to compile EP application.

   The detailed information can refer to SDK documents.

   ```
   #> bitbake fsl-image-core
   ```

2. Directly Compile host module

   a. Get the code and move them to sdk folder.

   ```
   git clone git://git.am.freescale.net/gitolite/sdk/skmm-host.git
   git checkout remotes/origin/pciedma_multi_chan_perf -b pciedma_multi_chan_perf
   ```

   b. Compile code

   For x86:

   ```
   lmh@lmh:~/work/sdk-devel-gerrit/skmm-host$ make ARCH=x86
   make -C /lib/modules/3.8.0-31-generic/build SUBDIRS=`pwd` modules
   make[1]: Entering directory `/usr/src/linux-headers-3.8.0-31-generic'
     LD [M]  /home/lmh/work/sdk-devel-gerrit/skmm-host/"pci_dma_test".o
     Building modules, stage 2.
     MODPOST 1 modules
     LD [M]  /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test.ko
   make[1]: Leaving directory `/usr/src/linux-headers-3.8.0-31-generic'
   cc  -Wall perf/mini_calc/mini_calc.c -o mini_calc
   lmh@lmh:~/work/sdk-devel-gerrit/skmm-host$
   ```

For PowerPC:

```
lmh@lmh:~/work/sdk-devel-gerrit/skmm-host$ make KERNEL_DIR=<path_to_your_linux_kernel>
CROSS_COMPILE=<name_of_your_toolchain> ARCH=powerpc
make -C /home/lmh/work/linux SUBDIRS=`pwd` modules
make[1]: Entering directory `/home/work/linux'
  CC [M]  /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test/pci_dma_dev.o
  CC [M]  /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test/pci_dma_sys.o
  CC [M]  /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test/pci_dma_test.o
  LD [M]  /home/lmh/work/sdk-devel-gerrit/skmm-host/"pci_dma_test".o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: ".dma_find_channel" [/home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test.ko]
undefined!
  CC      /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test.mod.o
  LD [M]  /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test.ko
make[1]: Leaving directory `/home/work/linux'
cc  -Wall perf/mini_calc/mini_calc.c -o mini_calc
lmh@lmh:~/work/sdk-devel-gerrit/skmm-host$
```

Run Host Module

The host module is named pci_dma_test.ko. We can directly insert the module. For example:

```
root@t4240qds:~# insmod pci_dma_test.ko
FSL PCI DMA Test Driver.
root@t4240qds:~#
```

We can use 'num_vfs' to make the driver enable virtual function. If we set ‘num_vfs=4’, the module will create 4 virtual functions for PF0 and PF1 respectively. For example:

```
root@t4240qds:~# insmod pci_dma_test.ko num_vfs=4
FSL PCI DMA Test Driver.
root@t4240qds:~#
```

The driver module provides the following sysfs interface to configure test settings. We can use cat and echo to change the related value.

such as packet length loop times and write direction(RC to EP or EP to RC)

1. bars_info: show the device bar information

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat bars_info
PCI DMA BAR0:
cpu_addr:0x0000000c00000000 size:0x0000000001000000
PCI DMA BAR1:
cpu_addr:0x0000000c01000000 size:0x0000000000002000
PCI DMA BAR2:
cpu_addr:0x0000000c01002000 size:0x0000000000001000
PCI DMA BAR3:
cpu_addr:0x0000000000000000 size:0x0000000000000000
PCI DMA BAR4:
cpu_addr:0x0000000c01400000 size:0x0000000000400000
```

2. config_info : display PCI EP configuration information

   For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat config_info
status:0x00000001 commond:0x00000000
```

```
rx config: addr:0x100f1000000, size:0x400, loop:0x3e8
root@t4240qds:/sys/class/pcidma/pcidma0#
```

3. test_dma_enable : enable/disable DMA

   For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_dma_enable
test dma status: enable
root@t4240qds:/sys/class/pcidma/pcidma0# echo 0 > test_dma_enable
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_dma_enable
test dma status: disable
root@t4240qds:/sys/class/pcidma/pcidma0#
```

4. test_loop : test loop times

   For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_loop
test loop: 500
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1000 > test_loop
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_loop
test loop: 1000
```

5. test_lens: test packet length

   For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_lens
test length: 64B 256B 1024B 4096B 1048576B 2097152B
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1024 > test_lens
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_lens
test length: 1024B
```

6. test_rc2ep: test direction RC writes to EP.

   For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_rc2ep
test rc2ep: true
root@t4240qds:/sys/class/pcidma/pcidma0# echo 0 > test_rc2ep
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_rc2ep
test rc2ep: false
root@t4240qds:/sys/class/pcidma/pcidma0#
```

7. test_ep2rc: test direction EP writes to RC

   For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_ep2rc
test ep2rc: false
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_ep2rc
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_ep2rc
test ep2rc: true
```

8. test_start: start to test, when finished the results will be printed.

   For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_start
test starting
```

```
root@t4240qds:/sys/class/pcidma/pcidma0#
test info:
test0 packet length:1024B loop:1000times
EP->RC throughput:2570Mbps
```

9. test_info: display the current test results.

   For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_info
test info:
test0 packet length:1024B loop:1000times
EP->RC throughput:2570Mbps
root@t4240qds:/sys/class/pcidma/pcidma0#
```

8.6.4 Test Procedure

Start EP Application

1. Configure PCI controller in EP mode via changing RCW
2. Start EP board and login using EP kernel image
3. Run PCI DMA EP application to initialize EP device.

```
root@p5020ds:~# ./pciep_dma  0
Initialized pci0-pf0
pcidma>
```

4. Start test thread on the specified PCI functions.

```
pcidma> add 0 0
pcidma> Starting a pci0-pf0's test thread on cpu1
```

Start Host driver module

1. Start host computer or board and login
2. Run PCI DMA host module.

```
root@t4240qds:~# insmod pci_dma_test.ko
FSL PCI DMA Test Driver.
FSL-PCIDMA-Driver 0001:01:00.0: BAR:0  addr:0xc20000000 len:0x16777216
FSL-PCIDMA-Driver 0001:01:00.0: BAR:1  addr:0xc21000000 len:0x8192
FSL-PCIDMA-Driver 0001:01:00.0: BAR:2  addr:0xc21002000 len:0x4096
FSL-PCIDMA-Driver 0001:01:00.0: BAR:3  addr:0x0 len:0x0
FSL-PCIDMA-Driver 0001:01:00.0: BAR:4  addr:0xc21400000 len:0x4194304
```

Test performance

1. Test RC to EP DMA performance

```
root@t4240qds:/sys/class# cd pcidma/pcidma0/
root@t4240qds:/sys/class/pcidma/pcidma0# ls
bars_info  config_infodevicesubsystem  test_dma_enable  test_ep2rctest_info  test_lens
test_loop  test_rc2ep  test_start  uevent
```

```
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_start
test starting
root@t4240qds:/sys/class/pcidma/pcidma0#
test info:
test0 packet length:64B loop:500times
RC->EP throughput:49Mbps
test1 packet length:256B loop:500times
RC->EP throughput:164Mbps
test2 packet length:1024B loop:500times
RC->EP throughput:658Mbps
test3 packet length:4096B loop:500times
RC->EP throughput:1594Mbps
test4 packet length:1048576B loop:500times
RC->EP throughput:11216Mbps
test5 packet length:2097152B loop:500times
RC->EP throughput:11337Mbps
```

2. Test RC to EP memcpy performance

```
root@t4240qds:/sys/class/pcidma/pcidma0# echo 0 > test_dma_enable
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_start
test starting
root@t4240qds:/sys/class/pcidma/pcidma0#
test info:
test0 packet length:64B loop:500times
RC->EP throughput:767Mbps
test1 packet length:256B loop:500times
RC->EP throughput:720Mbps
test2 packet length:1024B loop:500times
RC->EP throughput:767Mbps
test3 packet length:4096B loop:500times
RC->EP throughput:763Mbps
test4 packet length:1048576B loop:500times
RC->EP throughput:762Mbps
test5 packet length:2097152B loop:500times
RC->EP throughput:372Mbps
```

3. Test EP to RC DMA performance

```
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_ep2rc
root@t4240qds:/sys/class/pcidma/pcidma0# echo 0 > test_rc2ep
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_start
test starting
root@t4240qds:/sys/class/pcidma/pcidma0#
test info:
test0 packet length:64B loop:500times
EP->RC throughput:353Mbps
test1 packet length:256B loop:500times
EP->RC throughput:1270Mbps
test2 packet length:1024B loop:500times
EP->RC throughput:3893Mbps
test3 packet length:4096B loop:500times
EP->RC throughput:7710Mbps
test4 packet length:1048576B loop:500times
EP->RC throughput:12148Mbps
test5 packet length:2097152B loop:500times
EP->RC throughput:12162Mbps
```

Note: PCI DMA host module does not support KVM.

8.7 ARMv8 AArch32 User Manual

8.7.1 ARMv8 AArch32 User Manual

Linux SDK for QorIQ Processors

Some of QorIQ Processors are compatible with ARMv8 architecture, such as LayerScape platforms. The LayerScape platforms include LS1043A, LS1046A and so on.

The ARM architecture v8, ARMv8 supports two Execution states,

- A 64-bit Execution state, AArch64.
- A 32-bit Execution state, AArch32, which is compatible with previous versions of the ARM architecture.

LayerScape platforms can support AArch32 Execution state.

U-Boot and Kernel configurations and builds

For AArch32 support, the specific U-Boot and Kernel images are used.

To set up a cross compile environment and perform builds using Yocto Project, xxx-32b is used as the <machine>.

$ . ./fsl-setup-env -m <machine>

For example:

```
$ . ./fsl-setup-env -m ls1043ardb-32b
```

The corresponding images built by Yocto can be found in the following directory:

```
<sdk-install-dir>/build_<machine>/tmp/deploy/images/<machine>/
For example:
<sdk-install-dir>/build_ls1043ardb-32b/tmp/deploy/images/ls1043ardb-32b
```

Deploy and Boot Process

Deploy and boot process are the same with AArch64 platform, please refer to each board's reference manual.

Chapter 9
Linux User Space

9.1 QorIQ OpenDataPlane (ODP) User Manual

9.1.1 Introduction

This document provides information about ODP Sample Applications built on ODP API v1.11 implementation. User can experience the main functionalities of OpenDataPlane (ODP) with these applications and can also learn how to use the ODP API from source code of these applications.

The document explores the following target applications:

- ODP generator sample application
- ODP pktio sample application
- ODP ipsec transport and tunnel sample applications
- ODP packet classify sample application
- ODP timer sample application
- ODP traffic manager sample application
- ODP LPM IP Forwarding sample application
- ODP OpenFastPath Applications (FPM & FPM_BURSTMODE)

More information about ODP introduction, architecture and user guide are available on Linaro web site:

OpenDataPlane API Reference Manual for linux-generic

OpenDataPlane (ODP) User Guide

OpenDataPlane (ODP) Implementers Guide

9.1.1.1 Intended audience

This document is intended for software developers and architects who want to develop ODP applications on QorIQ based platforms. The document assumes that users are familiar with Linux-based software development, and also with the ODP concepts and APIs.

9.1.1.2 Definitions and acronyms

SDK	Software Development Kit
RDB	Reference Design Board
DUT	Device Under Test
FMan	Frame Manager
PBL	Pre Boot Loader

Table continues on the next page...

Table continued from the previous page...

RCW	Reset Configuration Word
UDP	User Datagram Protocol
SIP DIP	Source Internet Protocol and Destination Internal Protocol

9.1.1.3 Supported platforms

Current release is supported on the following QorIQ processing platforms:

- LS2088A

9.1.1.4 Unsupported ODP API's

Following ODP API's are not supported in this release

```
odp_packet_add_data
odp_packet_rem_data
odp_packet_extend_head
odp_packet_trunc_head
odp_packet_extend_tail
odp_packet_trunc_tail
odp_packet_copy_from_pkt
odp_packet_concat
odp_packet_split
odp_packet_copy_part
odp_packet_align
odp_packet_move_data
odp_packet_copy_data
odp_packet_has_ts
odp_packet_has_ts_clr
odp_packet_ts_set
odp_packet_ts
odp_cos_drop_set
odp_cls_cos_pool_set
odp_pktio_skip_set
odp_cls_cos_pool
odp_cos_drop
```

9.1.1.5 ODP Limitations and Known Issues

The following are the limitations and known issues for this release.

Limitations

- Route option with subnet is not supported in odp_ipsec and odp_ipsec_offload application.
- CRYPTO – DPAA hardware is an asynchronous hardware. Hence only asynchronous mode is supported for crypto operations.
- CRYPTO – Only ESP sessions are supported (crypto algorithm supported : 3DES, AES, authentication algorithm : SHA1, MD5, SHA256)
- CRYPTO – New buffer/Out of place crypto mode is not supported.

- CRYPTO – Sessions with requirement of authentication before cipher are not supported.
- Per CoS pool is not supported.
- Refer to the Unsupported ODP API's on page 1282 chapter for a list of the API's that are not supported in this release.

Known Issues

- **ODP-879** - FPM: application cleanup causes segmentation fault and can't be restarted.
- **ODP-894** - FPM burst mode is not working with single Tx Queue.
- **ODP-867** - Ordered queues are supported with only 500 buffers configured for all interfaces.
- **ODP-819** - Adding new SA cause the existing traffic to halt for some time.
- **ODP-837** - Ordered queues are not supported if there are more than one Tx queue per DPNI.
- **ODP-882** - Shaper with rate above 1Gbps is not supported.
- **ODP-416** - Freed buffer pool memory can't be reclaimed.
- **ODP-806** - IPsec offload application fails to re-run in ordered mode.
- **ODP-694** - Software queues can't be re-attached to schedular.

9.1.2 Product Description

9.1.2.1 Product diagram

The figure below identifies how ODP application runs in a typical network processor software architecture.



Figure 190. ODP on QorIQ

9.1.3 Build ODP Through Yocto

9.1.3.1 How to Install and Build SDK

For detailed information on ISO installation, see “Essential Build Instructions” under “Getting Started” within the QorIQ SDK documentation.

9.1.3.2 Building ODP applications only

ODP applications can also be compiled alone using Yocto once you have ran "*bitbake fsl-image-kernelitb*". The steps are as follows:

1. cd build_<board-name>/
2. source SOURCE_THIS
3. Clean ODP: bitbake -f -c cleansstate odp
4. Make ODP: bitbake -f -c compile odp

NOTE

The above step will only compile the odp applications. For adding them to rootfs and creating a Kernelitb, use the command given below.

5. bitbake fsl-image-kernelitb

9.1.4 Using ODP Applications

Execute the below commands to allocate resources for ODP applications.

9.1.4.1 LS2088ARDB/LS2085ARDB Board Preparation and Bring-up

Board Bring-up

The LS2088A documentation contains Software Deployment Guides for all the LS2088ARDB reference board. Refer to the LS2088ARDB Software Deployment Guide for instructions on how to deploy U-Boot, Linux kernel and Management Complex and root file system to the board.

Interface Naming Conventions

Throughout the ODP User Manual, ODP application commands are listed. These commands use intf-1, intf-2, …, intf-x as generic interface names. Use dpni.<x> in placeof intf-<x> when running ODP commands on LS2088ARDB/LS2085ARDB.

Example command listed in user manual:

```
$ ./odp_pktio -c 2 -m 2 -i <intf-1>,<intf-2>
```

The table below provides the mapping between the generic interface names and the names to be used in the ODP commands for LS2088ARDB/LS2085ARDB:.

Generic Interface Names	LS2088ARDB/LS2085ARDB Ports	LS2088ARDB/LS2085ARDB Interface Names
intf-1	eth4	dpmac.1(dpni.x*)
intf-2	eth5	dpmac.2(dpni.x*)
intf-3	eth6	dpmac.3(dpni.x*)
intf-4	eth7	dpmac.4(dpni.x*)
intf-5	eth0	dpmac.5(dpni.x*)
intf-6	eth1	dpmac.6(dpni.x*)
intf-7	eth2	dpmac.7(dpni.x*)
intf-8	eth3	dpmac.8(dpni.x*)

NOTE

Mapping between dpmac to dpni is auto-generated by the resource allocation "`dynamic_dpl.sh`" script. See the steps below for details.

Resource Allocation Commands

Execute the following commands to allocate the resources required to run the ODP applications.

```
$ cd /usr/odp/scripts
$ source ./dynamic_dpl.sh dpmac.1 dpmac.2  (Reference command)
```

NOTE

In order to use multiple interfaces, user can run the following command:

```
source ./dynamic_dpl.sh dpmac.1 dpmac.2 …… dpmac.8
```

The output log of the reference command is shown below:

```
###################### Container  dprc.2  is created ####################

Container dprc.2 have following resources :=>

 * 4 DPBP
 * 5 DPCON
 * 1 DPSECI
 * 3 DPNI
 * 10 DPIO
 * 10 DPCI


######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                dpmac.1               00:00:00:00:0:1
dpni.2                dpmac.2               00:00:00:00:0:2
dpni.3                dpni.3                00:00:00:11:11:11
```

NOTE

`dynamic_dpl.sh` script allocates default number of resources like buffer pools (DPBP), I/O thread context (DPIO), S/W queues (DPCI) etc. as shown in output above.

To increase the number of resources required by an application the environment variables below shall be exported to overwrite the default configuration before executing `dynamic_dpl.sh` script.

```
DPIO_COUNT : Defines the number of I/O contexts that are available to the application. Current
implementation allocates one DPIO for each timer thread spawned internally; in addition to
allocating a DPIO for a worker thread. Example - If an application initiates 8 worker threads
which
internally may spawn 8 timer threads; DPIO count for that case shall be at least 16. If
application
is not spawning timer threads then the DPIO count would be equal to the number of worker threads.

Usage : export DPIO_COUNT=<n>
    eg. "export DPIO_COUNT=10"

DPBP_COUNT : Defines number of buffer pools that can be allocated in an application.

Usage : export DPBP_COUNT=<n>
    e.g "export DPBP_COUNT=4"
```

```
DPCI_COUNT : Defines the number of queues that can be created in an application.

Usage : export DPCI_COUNT=<n>
    e.g "export DPCI_COUNT=10"

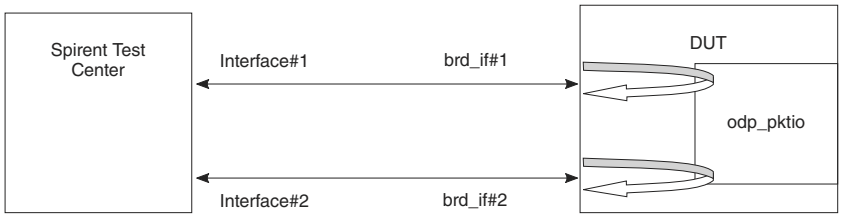
DPCONC_COUNT : Defines number of scheduler groups that can be created in an application. 3 DPIO's
are created by default for all applications to create Control, Worker and 'All' scheduler group.

Usage : export DPCONC_COUNT=<n>
    eg. "export DPCONC_COUNT=6"
```

Configuration for ODP OpenFastPath applications(fpm & fpm_burstmode)

```
DPCI_COUNT : Defines the number of queues that can be created in an application.
        Set the DPCI_COUNT=128 for OpenFastPath applications, using below command:

Usage : export DPCI_COUNT =<n>
e.g "export DPCI_COUNT =128"
```

Execute the command below to specify the memory requirement for ODP application:

```
export APPL_MEM_SIZE=32
```

The command above reservers 32MB of memory for the ODP application.

Note: If the command above is not executed, then first ODP application will consume all memory and no other ODP application can be run.

9.1.4.2 odp_pktio application

9.1.4.2.1 Overview

The odp_pktio application is a sample packet input/output application which receives packets from external traffic source and reflects back the packets after swapping the IP addresses and MAC addresses.

The application will be configured using its available command line options:

```
       Usage: %s OPTIONS
             E.g. %s -i eth1,eth2,eth3 -m 0

           OpenDataPlane example application.

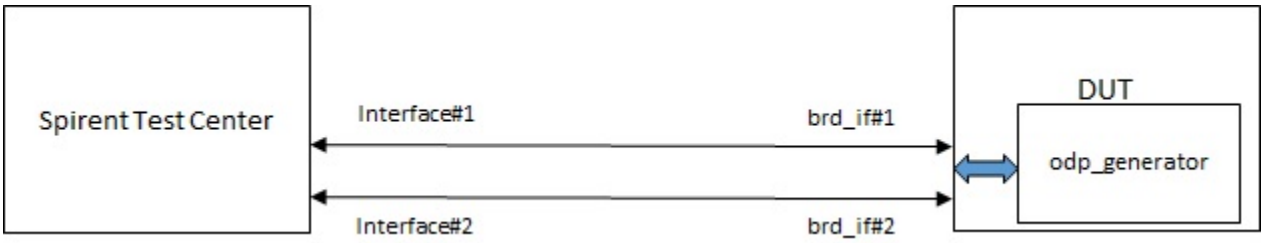
           Mandatory OPTIONS:
             -i, --interface Eth interfaces (comma-separated, no spaces)

           Optional OPTIONS
             -c, --count <number> CPU count.
             -m, --mode      0: Receive and send directly (no queues)
                             1: Receive and send via queues.
                             2: Receive via scheduler, send via queues.
             -h, --help           Display help and exit.
            environment variables: ODP_PKTIO_DISABLE_SOCKET_MMAP
                                   ODP_PKTIO_DISABLE_SOCKET_MMSG
                                   ODP_PKTIO_DISABLE_SOCKET_BASIC
            can be used to advanced pkt I/O selection for linux-generic
```

9.1.4.2.2 Test setup



Figure 191. Test setup (odp-pktio)

9.1.4.2.3 Running odp_pktio on DUT

Execute below commands on DUT to run odp_pktio:

```
$ ./odp_pktio -c 2 -m 2 -i <intf-1>,<intf-2>
```

NOTE

Interfaces intf-1 can be dpni.<index>. See LS2088ARDB/LS2085ARDB Board Preparation and Bring-up on page 1285 for interface details.

For additional details on options, see the command line options mentioned above.

9.1.4.2.4 Test description

The application will receive traffic from the traffic generator – and this will be seen in the application console, on one interface or on multiple interfaces where the application is running. Once a packet is received, its destination Ethernet addresses will be swapped (and so the IP addresses) and the packet will be sent back on the port it came from, to the packet originator.

As a reference, packet scheduler mode is chosen.

A number of 100000 packets will be sent on one port, from the Spirent test center/traffic generator.

Packet format

Eth source	Eth destination	Ip source	Ip destination	Proto
Any	Any	Any	Any	Ipv4/UDP

The application will send back 100000 packets to the Spirent test center, each packet having its Ethernet and IP addresses swapped.

9.1.4.3 odp_generator application

9.1.4.3.1 Overview

The odp_generator application is a sample application which demonstrate termination use cases. It supports the following functionalities:

1. Receive packets from external traffic source.
2. Transmit self generated UDP packets to external network.
3. Initiate a self generated PING request and expects corresponding response from the external network source.

The application will be configured using its available command line options:

```
Usage: %s OPTIONS
             E.g. %s -I eth1 -r

           OpenDataPlane example application.

             Work mode:
               1.send udp packets
               2.receive udp packets
               3.works like ping

           Mandatory OPTIONS:
             -I, --interface Eth interfaces (comma-separated, no spaces)
             -a, --srcmac src mac address
             -b, --dstmac dst mac address
             -c, --srcip src ip address
             -d, --dstip dst ip address
             -s, --packetsize payload length of the packets
             -m, --mode work mode: send udp(u), receive(r), send icmp(p)
             -n, --count the number of packets to be send
             -t, --timeout only for ping mode, wait ICMP reply timeout seconds
             -i, --interval wait interval ms between sending each packet
                           default is 1000ms. 0 for flood mode

           Optional OPTIONS
             -h, --help       Display help and exit.
            environment variables: ODP_PKTIO_DISABLE_SOCKET_MMAP
                                   ODP_PKTIO_DISABLE_SOCKET_MMSG
                                   ODP_PKTIO_DISABLE_SOCKET_BASIC
            can be used to advanced pkt I/O selection for linux-generic
```

9.1.4.3.2 Test setup

Figure 192. Test setup (odp_generator)



9.1.4.3.3 Running odp_generator on DUT

Execute below commands on DUT to run odp generator application:

Receive mode:

```
$ ./odp_generator -I <intf-1> -m r -w 1
```

UDP mode:

```
$ ./odp_generator -I <intf-1>--srcmac 00:00:00:00:00:0X --dstmac
00:00:10:00:94:02 --srcip 192.85.2.2 --dstip 192.85.1.2 --packetsize 64 -m u
```

Ping mode:

```
$ ./odp_generator -I <intf-1> --srcmac 00:00:00:00:00:0X --dstmac
00:10:94:00:00:02 --srcip 192.85.2.2 --dstip 192.85.1.2 --packetsize 64   -m
p
```

NOTE

Interfaces intf-1 can be dpni.<index> . See LS2088ARDB/LS2085ARDB Board Preparation and Bring-up on page 1285 for interface details.

For additional details on options, see the command line options mentioned above.

9.1.4.4 ODP ipsec application (odp_ipsec, odp_ipsec_offload)

9.1.4.4.1 Overview

The “odp_ipsec” application functions as a simple L3 IPv4 router with support for IPsec 3DES cipher in both transmit and receive directions. Note that both IPsec "transport" and “tunnel” modes are supported. These applications authenticate and encrypts/decrypts traffic from external source and forwards the encrypted/decrypted packets.

The applications will be configured using its available command line options:

```
Usage: %s OPTIONS
             E.g. %s -i eth1,eth2,eth3 -m 0

           OpenDataPlane example application.

           Mandatory OPTIONS:
            -i, --interface Eth interfaces (comma-separated, no spaces)
            -m, --mode   0: SYNC
                         1: ASYNC_IN_PLACE
                         2: ASYNC_NEW_BUFFER
                    Default: 0: SYNC api mode

           Routing / IPSec OPTIONS:
            -r, --route SubNet:Intf:NextHopMAC
            -p, --policy SrcSubNet:DstSubNet:(in|out):(ah|esp|both)
            -e, --esp SrcIP:DstIP:(3des|aes|null):SPI:Key192
            -a, --ah SrcIP:DstIP:(md5|sha1|null):SPI:Key128

             Where: NextHopMAC is raw hex/dot notation, i.e. 03.BA.44.9A.CE.02
                    IP is decimal/dot notation, i.e. 192.168.1.1
                    SubNet is decimal/dot/slash notation, i.e 192.168.0.0/16
                    SPI is raw hex, 32 bits
                    KeyXXX is raw hex, XXX bits long

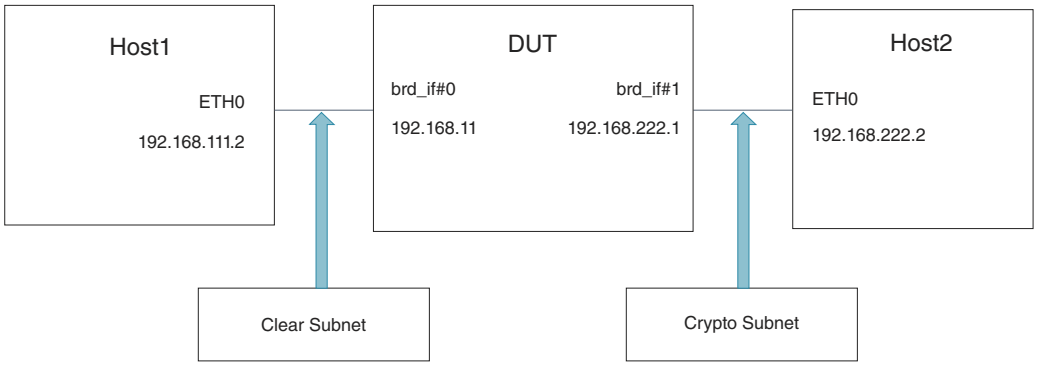
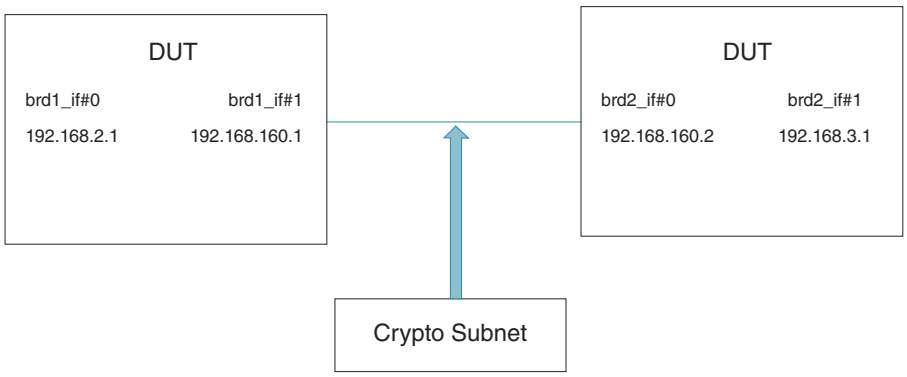
             Examples:
                -r 192.168.222.0/24:p8p1:08.00.27.F5.8B.DB
                -p 192.168.111.0/24:192.168.222.0/24:out:esp
                -e 192.168.111.2:192.168.222.2:3des:
201:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224
                -a 192.168.111.2:192.168.222.2:sha1:201:2122232425262728292a2b2c2d2e2f3031323334
```

In addition, the odp_ipsec_offload application demonstrates IPSec ESP protocol offload advantages over standard algorithm-oriented ipsec API.

The applications will be configured using its available command line options:

```
Usage: %s OPTIONS
             E.g. %s -i eth1,eth2,eth3

           OpenDataPlane example application.

           Mandatory OPTIONS:
            -i, --interface Eth interfaces (comma-separated, no spaces)

           Routing / IPSec OPTIONS:
            -r, --route SubNet:Intf:NextHopMAC
            -p, --policy SrcSubNet:DstSubNet:(in|out):(ah|esp|both)
            -e, --esp SrcIP:DstIP:(3des|aes|null):SPI:Key192
            -a, --ah SrcIP:DstIP:(md5|sha1|null):SPI:Key128

             Where: NextHopMAC is raw hex/dot notation, i.e. 03.BA.44.9A.CE.02
                    IP is decimal/dot notation, i.e. 192.168.1.1
                    SubNet is decimal/dot/slash notation, i.e 192.168.0.0/16
                    SPI is raw hex, 32 bits
                    KeyXXX is raw hex, XXX bits long
       Modes Options:
       -q,   specify the queue type
               0: ODP_SCHED_SYNC_PARALLEL
               1: ODP_SCHED_SYNC_ATOMIC (default)
               2: ODP_SCHED_SYNC_ORDERED

             Examples:
                -r 192.168.222.0/24:p8p1:08.00.27.F5.8B.DB
                -p 192.168.111.0/24:192.168.222.0/24:out:esp
                -e 192.168.111.2:192.168.222.2:3des:
201:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224
                -a 192.168.111.2:192.168.222.2:sha1:201:21222324252627282a2b2c2d2e2f3031323334
```

9.1.4.4.2 Test setup

This test uses the hardware setup described in below figures:



Figure 193. Transport mode setup



Figure 194. Tunnel mode setup

9.1.4.4.3 Running ODP ipsec applications on DUT

Transport mode (odp_ipsec)

Execute below commands on DUT for odp_ipsec application in transport mode.

```
$ ./odp_ipsec -i brd_if#0, brd_if#1 -c <Num of cores> -m 1 -
r 192.168.111.2/32: brd_if#0:00.11.00.00.00.01 -r 192.168.222.2/32: brd_if#1:00.22.00.00.00.02 -p
192.168.111.0/24:192.168.222.0/24:out:esp -e
192.168.111.2:192.168.222.2:3des:201:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224
-p 192.168.222.0/24:192.168.111.0/24:in:esp -e
192.168.222.2:192.168.111.2:3des:301:c966199f24d095f3990a320d749056401e82b26570320292
```

Execute below commands on Host1:

- ifconfig eth0 192.168.111.2 up
- ifconfig eth0 hw ether 00:11:00:00:00:01
- ip route add 192.168.222.0/24 via 192.168.111.1
- arp -s 192.168.111.1 00:00:00:00:00:01

Execute below commands on Host2 for odp_ipsec application in transport mode

- ifconfig eth0 192.168.222.2 up
- ifconfig eth0 hw ether 00:22:00:00:00:02
- ip route add 192.168.111.0/24 via 192.168.222.1
- arp -s 192.168.222.1 00:00:00:00:00:02

Host2 has the following setkey configuration file applied. Use reference setkey script as below:

```
#!/usr/sbin/setkey -f
# Flush the SAD and SPD
flush;
spdflush;

add 192.168.111.2 192.168.222.2 esp 0x201 -E 3des-cbc
0x656c8523255ccc23a66c1917aa0cf30991fce83532a4b224;

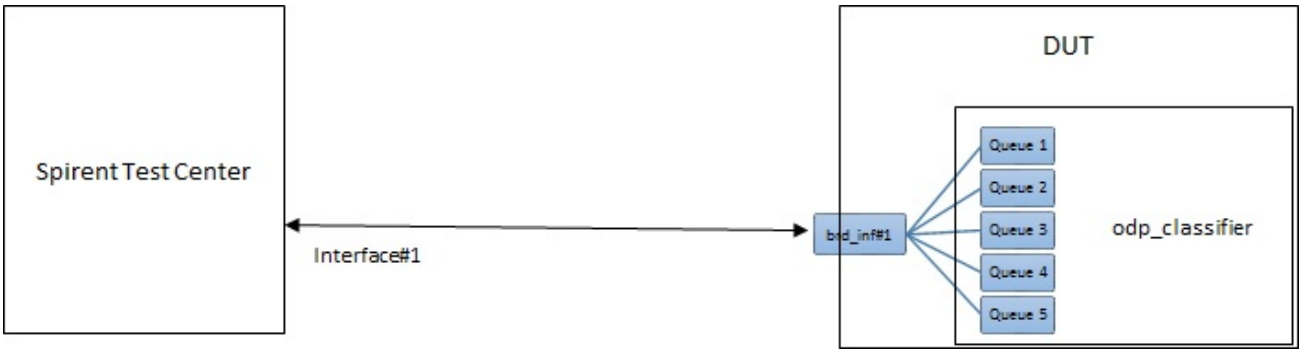
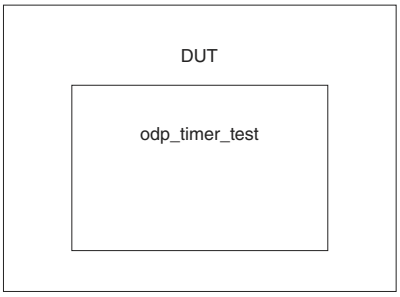
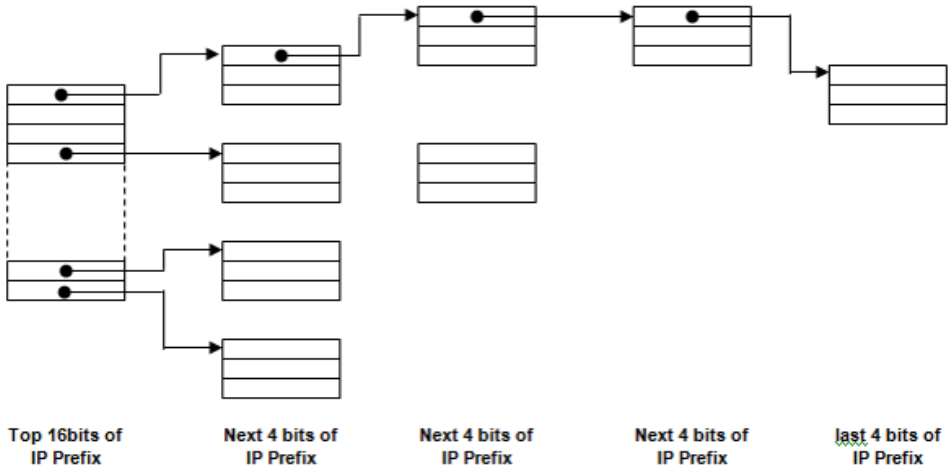
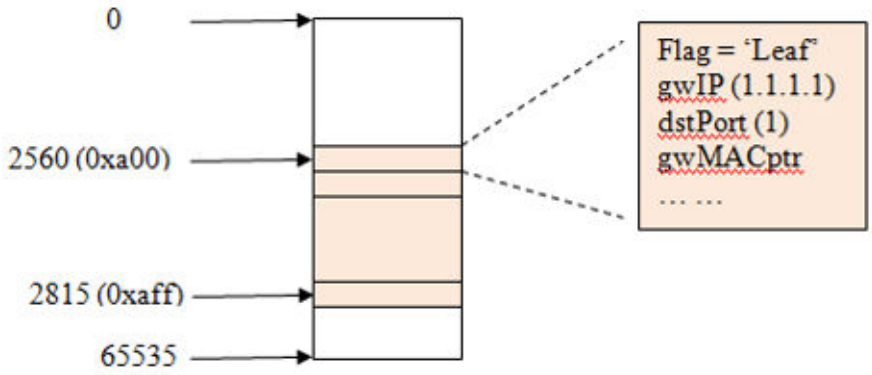
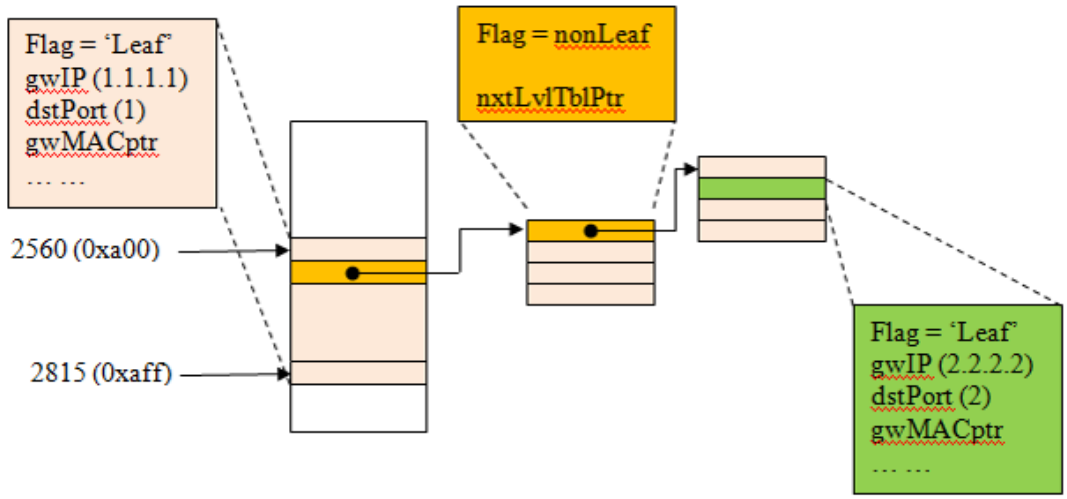
add 192.168.222.2 192.168.111.2 esp 0x301 -E 3des-cbc
0xc966199f24d095f3990a320d749056401e82b26570320292;
```

```
        spdadd 192.168.111.2 192.168.222.2 any -P in ipsec
                  esp/transport//require;

        spdadd 192.168.222.2 192.168.111.2 any -P out ipsec
                  esp/transport//require;
```

Tunnel mode (odp_ipsec_offload and odp_ipsec)

Execute the below commands for tunnel mode in odp_ipsec_offload application

```
On Left board(DUT):

$ ./odp_ipsec_offload -i brd1_if#0,brd1_if#1 -r 192.168.2.2/32:brd1_if#0:02.00.c0.a8.3c.01 -r
192.168.3.2/32:brd1_if#1:<next hop (mac addr of brd2_if#0)> -p
192.168.2.0/24:192.168.3.0/24:out:both -e
192.168.2.2:192.168.3.2:aes:1:0102030405060708090a0b0c0d0e0f10 -a
192.168.2.2:192.168.3.2:sha1:1:21222324252627282922b2c2d2e2f3031323334 -t
192.168.2.2:192.168.3.2:192.168.160.1:192.168.160.2 -p 192.168.3.0/24:192.168.2.0/24:in:both -e
192.168.3.2:192.168.2.2:aes:2:0102030405060708090a0b0c0d0e0f10 -a
192.168.3.2:192.168.2.2:sha1:2:21222324252627282922b2c2d2e2f3031323334 -t
192.168.3.2:192.168.2.2:192.168.160.2:192.168.160.1 -c 1


On Right board(Host2):

$ ./odp_ipsec_offload -i brd2_if#0,brd2_if#1 -r 192.168.3.2/32:brd2_if#1:02.00.c0.a8.3c.02 -r
192.168.2.2/32:brd2_if#0:< next hop (mac addr of brd1_if#1> -p
192.168.2.0/24:192.168.3.0/24:in:both -e
192.168.2.2:192.168.3.2:aes:1:0102030405060708090a0b0c0d0e0f10 -a
192.168.2.2:192.168.3.2:sha1:1:21222324252627282922b2c2d2e2f3031323334 -t
192.168.2.2:192.168.3.2:192.168.160.1:192.168.160.2 -p 192.168.3.0/24:192.168.2.0/24:out:both -e
192.168.3.2:192.168.2.2:aes:2:0102030405060708090a0b0c0d0e0f10 -a
192.168.3.2:192.168.2.2:sha1:2:21222324252627282922b2c2d2e2f3031323334 -t
192.168.3.2:192.168.2.2:192.168.160.2:192.168.160.1 -c 1
```

Execute the below commands for tunnel mode in odp_ipsec application

On Left board(DUT):

```
$ ./odp_ipsec -i brd1_if#0,brd1_if#1 -r 192.168.2.2/32:brd1_if#0:02.00.c0.a8.3c.01 -r
192.168.3.2/32:brd1_if#1:<next hop (mac addr of brd2_if#0)> -p
192.168.2.0/24:192.168.3.0/24:out:both -e
192.168.2.2:192.168.3.2:3des:1:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224 -a
192.168.2.2:192.168.3.2:md5:1:a731649644c5dee92cbd9c2e7e188ee6 -t
192.168.2.2:192.168.3.2:192.168.160.1:192.168.160.2 -p 192.168.3.0/24:192.168.2.0/24:in:both -e
192.168.3.2:192.168.2.2:3des:2:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224 -a
192.168.3.2:192.168.2.2:md5:2:a731649644c5dee92cbd9c2e7e188ee6 -t
192.168.3.2:192.168.2.2:192.168.160.2:192.168.160.1 -c 1 -m 1
```

On Right board(Host2):

```
$ ./odp_ipsec -i brd2_if#0,brd2_if#1 -r 192.168.3.2/32:brd2_if#1:02.00.c0.a8.3c.02 -r
192.168.2.2/32:brd2_if#0:< next hop (mac addr of brd1_if#1> -p
192.168.2.0/24:192.168.3.0/24:in:both -e
192.168.2.2:192.168.3.2:3des:1:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224 -a
```

```
192.168.2.2:192.168.3.2:md5:1:a731649644c5dee92cbd9c2e7e188ee6 -t
192.168.2.2:192.168.3.2:192.168.160.1:192.168.160.2 -p 192.168.3.0/24:192.168.2.0/24:out:both -e
192.168.3.2:192.168.2.2:3des:2:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224 -a
192.168.3.2:192.168.2.2:md5:2:a731649644c5dee92cbd9c2e7e188ee6 -t
192.168.3.2:192.168.2.2:192.168.160.2:192.168.160.1 -c 1 -m 1
```

For LS208xARDB:

```
brd1_if#0 dpni.<DPNI_INDEX_1>
brd2_if#0 dpni.<DPNI_INDEX 2>
brd2_if#1 dpni.<DPNI_INDEX_1>
brd1_if#1 dpni.<DPNI_INDEX_2>
```

For additional details on options, see the command line options mentioned above.

9.1.4.5 odp_classifier application

9.1.4.5.1 Overview

odp_classifier is a sample application to demonstrate classification among the queues based on configured packet matching rules along with the rules for non-matching(default queue) as well as error'ed packets. The current implementation demonstrates classification based on following parameters:

1. IP Source address
2. IP QoS
3. VLAN Priority

The following are functional behavior of the application:

- Application configures multiple packet matching rules specifying the queue as a target place.
- Traffic, matching with the configured rules, will be distributed among the target queues.
- Remaining traffic will be forwarded to default queue.

Application is supported for two modes as given below:

- **Drop mode**: All the received frames are dropped at application.
- **Reply mode:** Received frame are reflected back onto the same interface after swapping its Ethernet SMAC & DMAC and SIP & DIP addresses.

The application will be configured using its available command line options:

```
Usage: %s OPTIONS

           Mandatory OPTIONS:
             -i, --interface Eth interfaces (comma-separated, no spaces)
   -p, --policy <odp_pmr_term_e>:<value>:<mask bits>:<queue name>
odp_pmr_term_e: Type of packet matching rule
value: Value of rule to be matched
mask bits: Masking bits for matching rules
queue name: Target Queue where packet is to be received
   -l, --l2_policy <VLAN Priority Value>:<queue name>
VLAN Priority Value: L2 VLAN priority value
:Queue Name: Target Queue where packet is to be received
   -q, --l3_policy <L3 QoS Value>:<queue name>
L3 QoS Value: L3 QoS or IP TOS (One byte) value
Queue Name: Target Queue where packet is to be received
           Optional OPTIONS
  -c, --count <number> CPU count.
```

```
  -m, --mode Mode of application
0: Packet Drop mode. Received packets will be dropped
1: Packet Reply mode. Received packets will be sent back
Default: Packet Drop mode.
   -t, --timeout Time to run the classifier
1: Time for which the classifier will be run in seconds
0: Runs in infinite loop
Default: Runs in infinite loop.
   -a, --l3_policy_precedence Flag to flip precedence between L2 and L3 Policy.
1: L3 policy rule will take precedence over L2 Policy rules
0: L2 policy rule will take precedence over L3 policy rules
Default: 0
             -h, --help       Display help and exit
```

9.1.4.5.2 Test setup



Figure 195. Test setup (odp_classifier)

9.1.4.5.3 Running odp_classifier on DUT

Execute below commands on DUT to odp classifier application:

```
Reply mode:
$./odp_classifier –i <intf-1> -p ODP_PMR_SIP_ADDR:192.85.1.1:FFFFFFFF:queue1 -l 1:queue2 -q
40:queue3 -m 1
Drop Mode:
$./odp_classifier –i <intf-1> -p ODP_PMR_SIP_ADDR:192.85.1.1:FFFFFFFF:queue1 -l 1:queue2 -q
40:queue3 -m 0
```

NOTE

Interface intf-1 can be dpni.<index> or fm<X>-mac<Y>. See LS2088ARDB/LS2085ARDB Board Preparation and Bring-up on page 1285 for interface details as per board in use.

NOTE

For LS2085ARDB/LS2088ARDB, DPBP objects are required to be increaesed to 16 for configuring multiple rules for classifciation. Refer section LS2088ARDB/LS2085ARDB Board Preparation and Bring-up on page 1285 to increase object count in resource container.

For additional details on options, see the command line options mentioned above.

9.1.4.6 odp_timer application

9.1.4.6.1 Overview

odp_timer is a sample application that will enqueue an event after a certain time period, when the timer expires and dequeues that event and validates it.

The application will create number of threads, given by –c option and allocate a timer for each thread. Each timer will be added by the application with ticks given by –p option.

Feasibility of ticks will be checked by min and max values given by options –m and –x.

When timer expires, a timeout buffer will be enqueued into a queue that will be received by application using odp_schedule() API. After that, buffer will be validated and timer will reset if more timeouts remain given by –t option.

The application will be configured using its available command line options:

```
Usage: %s OPTIONS
  -c, --count <number>    CPU count);
  -r, --resolution <us>   timeout resolution in usec
  -m, --min <us>          minimum timeout in usec
  -x, --max <us>          maximum timeout in usec
  -p, --period <us>       timeout period in usec
  -t, --timeouts <count>  timeout repeat count
  -h, --help                    this help
```

9.1.4.6.2 Test setup



Figure 196. Test setup (odp_timer)

9.1.4.6.3 Running odp_timer on DUT

Execute below commands on DUT to run the odp timer application

```
$./odp_timer_test -c 2 -t 2
```

For additional details on options, see the command line options mentioned above.

9.1.4.7 odp_lpmfwd application

9.1.4.7.1 Overview

odp_lpmfwd application demonstrates the IPv4 packets forwarding based upon Longest Prefix Match methodology. The LPM based IPv4 forwarding application is a multi-threaded application that routes IPv4 packets from one Ethernet interface to another. The packets that reach the LPM application can be forwarded to destination IP address using LPM route algorithm. The LPM routing algorithm uses the prefix for destination IP address to do the route look up.

Longest Prefix Match Algorithm:

Instead of using traditional Radix-Trie algorithm, here we choose to use one simpler LPM algorithm. It uses 5 level tables: First level table – 65536 entries array, indexed by the top 16 bits of IP address. Second level table – 32 entries array, indexed by bit12~15 of IP address. Third level table – 32 entries array, indexed by bit8~11 of IP address. Fourth level table – 32 entries array, indexed by bit4~7 of IP address. Fifth level table – 32 entries array, indexed by bit0~3 of IP address. The 2nd level to 5th level tables are only created when its first level table entry has valid value. See below figure:



At init time, only the first level (i.e. top16 bits) array is created which contains 65536 null entries. While adding route entries to the FIB table, the 2nd level to 5th level arrays will be created accordingly. This is a typical ASIC design algorithm of LPM which is fast and simple to search while costs far more memory. The worst case is to index and compare 5 times when searching an IP address, but it's still fast enough.

Examples:

1) Add one Class A route 10.0.0.0/8 to the route table. (Gateway is 1.1.1.1, destination port is 1) The first 16bit of 10.0.0.0 (0x0a000000) is 0x0a00 (2560). And the mask is 8 bit which is smaller than the 1st level bit-length (16b), so below entries (from 0x0a00 to 0x0aff) will be created in the FIB table:



From entry No.2560 to No.2815 (total 255 entries) are filled with same content (flag, gwIP, dstPort, ptr …). Now if a packet with DIP of 10.1.1.1 comes in, its first 16 bit value is 0x0a01 (2561). So, the No.2561 entry of the 1st level table will be checked. If it's a 'leaf' node (now it is), then the best-match is found. And the packet will be forwarded to the 'dstPort' after replacing the SMAC and DMAC. And, any DIP of 10.x.x.x will all be forwarded to port 1 with gwIP 1.1.1.1 based on above table.

2) Now, a new route 10.1.1.0/24 is added to the FIB table. The table will be like this:



The new route will overwrite the No.2561 (0x0a01) entry with 'flag' of 'nonLeaf' and a pointer to 'next-level-table'. Now a 16-entry-block memory will be allocated as the 'next-level-table' because the 2nd level is 4bit indexed. And the base address of this new block will be set to the 'nxtLvlTblPtr' of No.2561. Because the next 4 bit of the new route is 0 (bit16 to bit19 of 0x0a010100), so the 1st entry in the 2nd level table is used as another 'nonLeaf' entry. While all the other entries in the 2nd level table should be filled with the same value of its 'parent' route (10.0.0.0/8). The netmask is 24bit which is larger than 16+4, so the 3rd level table should also be allocated (16 entries). And the next 4bit of the new route is 1 (bit20 to bit23 of 0x0a010100), so the 2nd entry will be used for the new route. And because the netmask (24bit) is no-larger-than 16+4+4, so this entry will be the 'Leaf' entry of this new route (see above figure in green). And the according values (gwIP, dstPort, etc.) will be filled in that entry. Now a frame with DIP of 10.1.1.100 comes. There will have 3 lookups to get the final result:

- Index with first 16bit of DIP, whose value is 0x0a01. 'Non-leaf' means to continue the next-level lookup.
- Index with the next 4bit of DIP, whose value is 0. Then 'Non-leaf' again.
- Index with the next 4bit of DIP, whose value is 1. Then the 'leaf' node is found and the lookup reaches an end.

Now a frame with DIP of 10.1.192.10 comes. You can see it will find the 'leaf' node in the 2nd level table and get the route of net-address 10.0.0.0/8. And a frame with DIP of 10.1.10.10 will find its 'leaf' node in the 3rd level table and also get the route of net-address 10.0.0.0/8 as we expected. The multi-branch trie algorithm provides a very fast way of route-lookup but a relatively complicated way of route-add/deletion.

Application Usage:

There are two binaries for LPM based IPv4 packets forwarding:

odp_lpmfwd : Main binary which will do forwarding.

odp_lpmfwd_config : Helper binary to configure the routes, interfaces and ARPs.

Both binaries will communicate via Linux message queues.

odp_lpmfwd usage:

./odp_lpmfwd -h

```
OpenDataPlane LPM forwarding application.

Usage: odp_lpmfwd OPTIONS
  E.g. odp_lpmfwd -i eth1,eth2 -m 0 -c 1
 In the above example,
 eth1 and eth2 are the interfaces from which pkts will be forwarded
```

```
 depends upon the routes

Mandatory OPTIONS:
  -i, --interface eth interfaces (comma-separated, no spaces)

Optional OPTIONS
  -m, --mode      0: Burst send & receive packets (no queues)
                  1: Send & receive packets through ODP Schedular.
                        Default: Packet burst mode.
  -c, --count <number> CPU count.
  -h, --help           Display help and exit.
```

odp_lpmfwd_config usage:

./odp_lpmfwd_config --help

```
Usage:
        odp_lpmfwd_config[OPTIONS…]
    -B, --routeadd=TYPE        adding a route
    -E, --showintf=TYPE        show interfaces
       -F, --intfconf=TYPE        change intf config
    -G, --arpadd=TYPE          adding a arp entry
    -P, --PID=TYPE             pid to hook with (Mandatory param)
        -?, --help                Give this help list
        --usage                   Give a short usage message
        -V, --version             Print program version
```

Help for show all enabled interfaces:

./odp_lpmfwd_config –P <pid of odp_lpmfwd> -E –help

```
Usage: -E [OPTION...]
        -a, --a=ALL      All interfaces
          -?, --help        Give this help list
          --usage           Give a short usage message
          -V, --version     Print program version

Example:
                ./odp_lpmfwd_config -P 2234 -E -a true
```

Help for assign IP addresses to interfaces:

./odp_lpmfwd_config –P <pid of odp_lpmfwd> -F –help

```
Usage: -F [OPTION...]
 -a, --a=IPADDR             IP Address
-i, --i=IFNUM               If Number
-?, --help                  Give this help list
--usage                 Give a short usage message
-V, --version               Print program version

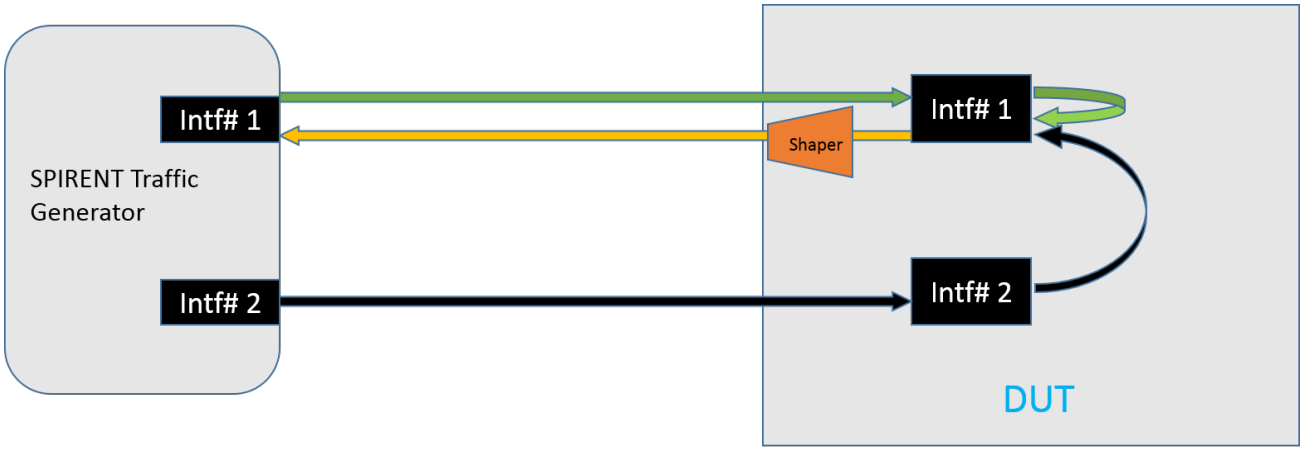
Example:       ./odp_lpmfwd_config -P 2234 -F -a
        192.168.222.1 -i 1Where i is the index of the interface.

e.g.: For following command:
./odp_lpmfwd –i intf-2,intf-1
In odp_lpmfwd_config to assign the IP addresses “i” should be 1 for intf-2 and 2 for intf-1
```

Help for ARP entry addition:

./odp_lpmfwd_config –P <pid of odp_lpmfwd> -G –help

```
Usage: -G [OPTION...]
-m, --m=MACADDR      MAC Address
-r, --r=Replace      Replace Exiting Entry - true/ false {Default:false}
-s, --s=IPADDR       IP Address
-?, --help           Give this help list
--usage                 Give a short usage message
-V, --version               Print program version

Example:       ./odp_lpmfwd_config -P 2234 -G -s 192.168.111.2 -m 00:00:00:00:00:11 -r true
```

Help for route entry addition:

./odp_lpmfwd_config –P <pid of odp_lpmfwd> -B –help

```
Usage: -B [OPTION...]

  -c, --c=FIBCNT             Number of FIB entries
  -d, --d=DESTIP             Destination IP
  -g, --g=GWIP               Gateway IP
  -n, --n=NETMASK            netmask length to be used by LPM
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

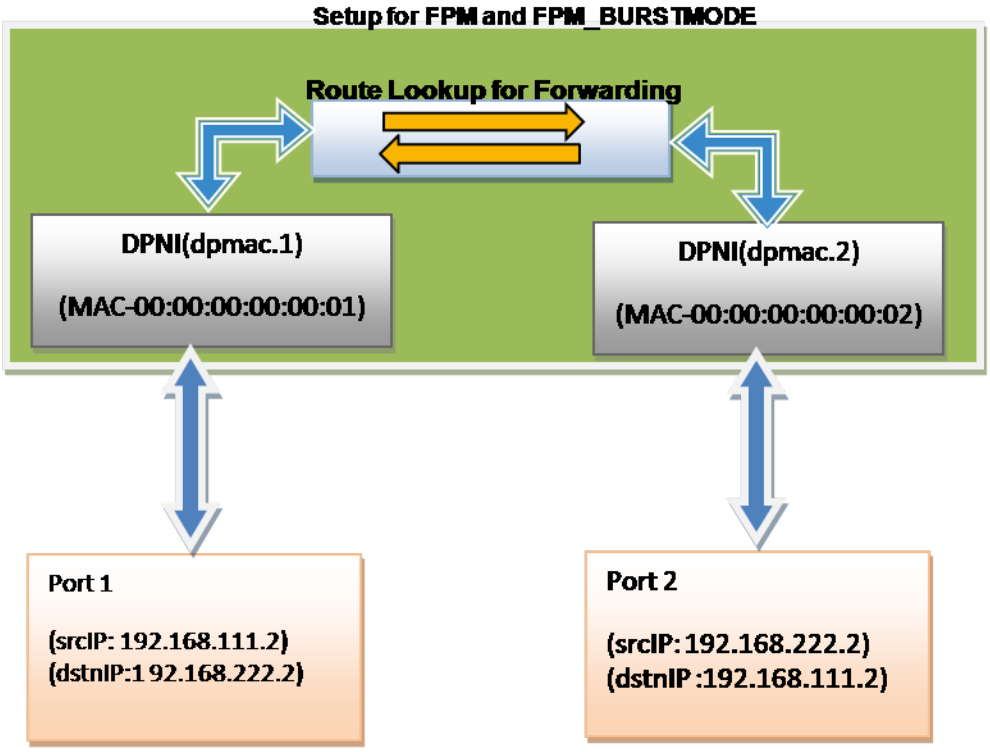
Example:
    ./odp_lpmfwd_config -P 2234 -B -d 192.168.111.0 -g 192.168.222.2 -n 24 -c 256

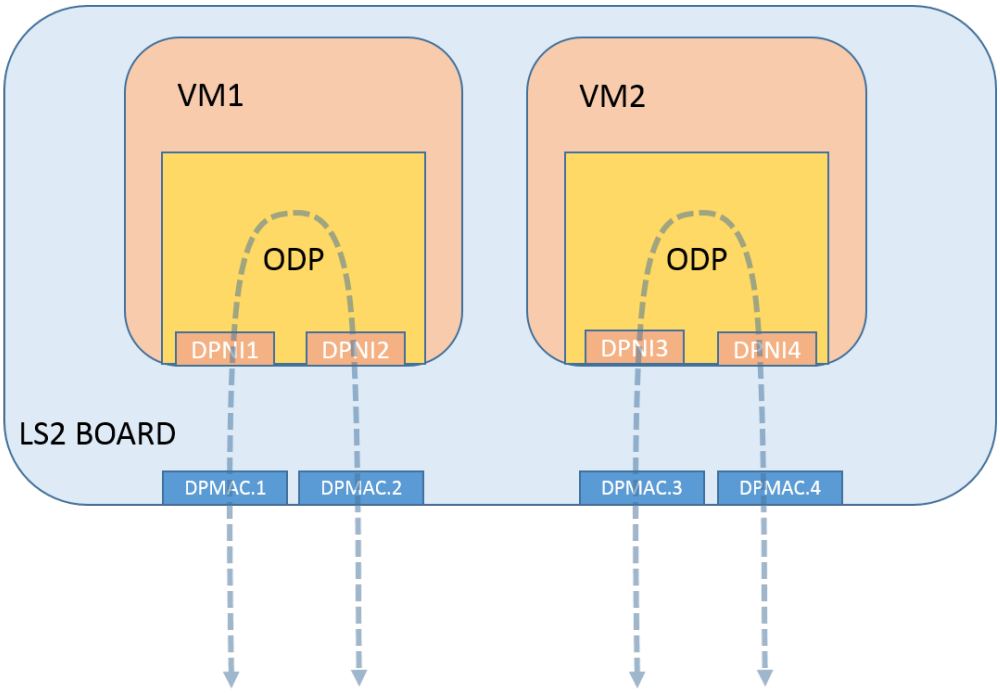
Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

9.1.4.7.2 Running odp_lpmfwd on DUT

Execute below commands on DUT to run odp_lpmfwd:

```
$ ./odp_lpmfwd –i <intf-1>,<intf-2> &
```

NOTE

Interfaces intf-1 and intf-2 can be `fm<X>-mac<Y>`. See LS2088ARDB/LS2085ARDB Board Preparation and Bring-up on page 1285 for interface details as per board in use.

To configure the interfaces, routes and ARPs, run the following commands:

```
$ ./odp_lpmfwd_config -P $1 -E -a true
$./odp_lpmfwd_config -P $1 -F -a 192.168.222.1 -i 1
$./odp_lpmfwd_config -P $1 -F -a 192.168.111.1 -i 2
$./odp_lpmfwd_config -P $1 -G -s 192.168.222.2 -m 00:00:00:00:00:10 -r true
$./odp_lpmfwd_config -P $1 -G -s 192.168.111.2 -m 00:00:00:00:00:11 -r true
$./odp_lpmfwd_config -P $1 -B -d 192.168.111.0 -g 192.168.222.2 -n 24 -c 256
$./odp_lpmfwd_config -P $1 -B -d 192.168.222.0 -g 192.168.111.2 -n 24 -c 256
```

Where $1 is PID of the process odp_lpmfwd.

9.1.4.7.3 Test description

When a packet is received on one of the input interfaces, it will be routed and forwarded according to the LPM algorithm. The packet will be changed - destination Ethernet address will be updated according to received command line options, the TTL

(Hop Limit) will be decremented and the checksum will be recalculated. Finally the packet will be transmitted on the outgoing interface.

Packet format

Stream 1 on interface#1: IPv4 stream, DIP 192.168.222.4

Stream 2 on interface#2: IPv4 stream, DIP 192.168.111.4

9.1.4.8 odp_tm application

9.1.4.8.1 Overview

odp_tm application demonstrate general Traffic management mechanism on egress side that accepts packets from input queues and applies scheduling and/or bandwidth controls to decide which input packet should be chosen as the next output packet and when this output packet can be sent to external network.

odp_tm supports upto 8 tm_queues (0 to 7) on egress side. Number of tm_queues are configurable (discussed in later sections). Once a packet is received then tm_queue selection is done based on below criteria:

If packet contains a VLAN header then VLAN priority signifies the output tm_queue i.e. VLAN prio 0 to tm_queue0, VLAN prio 1 to tm_queue1 and so on. If configured tm_queues are lesser than VLAN prio value then non-matching VLAN priority packet will always be enqueued to last tm_queue. If packet does not contains any VLAN header then packet will always be enqueued to last tm_queue. PRIO=0 having the highest priority and PRIO=7 having the lowest priority.

Supported scheduling algorithm:

- Strict Priority Scheduling upto 8 priority tm_queues.
- Weighted Fair Queue Scheduling. If weights on each are same then weighted round robin scheduling will be in effect.

Along with Scheduling, single rate bandwidth shaper can also applied on outgoing traffic.

Output interface will be selected based on the configured route using option "-d" (--Destination SubNet:Intf:NextHopMAC)

```
Usage: odp_tm OPTIONS
 E.g. odp_tm -i eth1,eth2,eth3 -m 0
OpenDataPlane example application.
Mandatory OPTIONS:
    -i, --interface Eth interfaces(comma-separated, no spaces)
Optional OPTIONS
     -c, --count <number> CPU count.
     -d, --Destination SubNet:Intf:NextHopMACSubNet: IPaddress with mask bits
              i.e. aa:bb:cc:dd/maskbitsIntf: Interface name i.e. dpni-0NextHopMAC: Destination Mac
Address for next hop.
              Bytes are dot(.) separated i.e. 00.00.00.00.08.01
     -m, --mode 0: Scheduling profile as Strict Priority
                1: Scheduling profile as Weighted scheduling.
                Default Scheduling profile as Strict Priority
    -s, --shaping <boolean> Flag to enable/disable shaping.
                0: To disable
                1: To enable.
                Default shaper is disabled
     -r, --rate <number> Shaping rate in Mbps. Valid only if shaping enabled Values must be in range
(1, 10000)
     -b, --burst-size <number> maximum burst size in KB.Valid only if shaping enabled Values must
be in range (1, 64)
     -n, --num-prio <number> Number of priority queues configured on egress side.
                            Only valid if scheduling profile is strict prio Default all supported
priority queues will be configured.
```

```
     -w, --weight <queue_name>:<weight> Weight value corresponding to each queue.
                        multiple queues configuration will be comma-separated and no spaces. Values
must be in range (1, 255).
               E.g. odp_tm -w queue1:10,queue2:20 -h, --help Display help and exit.
```

NOTE

Input configuration used in command will be applied on all the network interfaces that are being used in the command.

9.1.4.8.2 Running odp_tm on DUT

Execute below commands on DUT to run odp_tm application:

```
./odp_tm -i <intf-1>,<intf-2> -c 8 -d 192.168.111.0/24:dpni.2:00.00.00.00.08.02 -d
192.168.222.0/24:dpni.1:00.00.00.00.08.01 -s 1 -r 500 -b 32
```

Note: The following environment variables needs to be set " export MAX_TCS=8" "export OPR_ENABLE=0" "export FS_ENTRIES=1" before running dynamic_dpl.sh script. Interfaces intf-1 can be dpni.<index>. See LS2088ARDB/LS2085ARDB Board Preparation and Bring-up on page 1285 for interface details.

9.1.4.8.3 Test Setup



9.1.4.8.4 Test Description

Validation of scheduling without shaper:

Configure traffic stream on both the interfaces, having VLAN priority 0 to 7 and destination IP address targeting to a common output interface. Inject traffic to both the input interfaces.

- In case of strict priority scheduling, highest priority traffic (VLAN prio 0) should be received on output interface if input traffic rate is greater than output rate(10Gbps) otherwise mixed traffic will be received.
- In case of weighted priority scheduling, mixed traffic will be received depending upon the queue weights on output interface with 10Gbps output rate. If configured weights are equal, then mixed traffic will be in equal percentage.

Validation of scheduling with shaper:

Configure traffic stream on both the interfaces, having VLAN priority 0 to 7 and destination IP address targeting to a common output interface. Inject traffic to both the input interfaces.

- In case of strict priority scheduling, highest priority traffic (VLAN prio 0) should be received on output interface if input traffic rate is greater than output rate(Configured shaping rate) otherwise mixed traffic will be received.

- In case of weighted priority scheduling, mixed traffic will be received depending upon the queue weights on output interface with configured shaping rate. If configured weights are equal, then mixed traffic will be in equal percentage.

9.1.4.9 OpenFastPath applications

9.1.4.9.1 Overview

OpenFastPath is an open source implementation of a high performance TCP/IP stack that provides features that network application developers need to cope with today's fast-paced network. To enable this packet processing OFP uses OpenDataPlane.

Following application have been integrated in the ODP for the use on Layerscape architecture:

1.FPM (Forwarding Plane Manager)

```
./fpm -h
Usage: fpm OPTIONS
 E.g. fpm -i eth1,eth2,eth3
ODPFastpath application.

Mandatory OPTIONS: -i, --interface Eth interfaces (comma-separated, no spaces)

Optional OPTIONS
 -c, --count <number> Core count.
 -p, --performance Performance Statistics
 -h, --help Display help and exit.
```

2. FPM_BURSTMODE (Forwarding Plane Manager in burstmode)

```
./fpm_burstmode -h
Usage: fpm_burstmode OPTIONS
 E.g. fpm_burstmode -i eth1,eth2,eth3
ODPFastpath application.

Mandatory OPTIONS: -i, --interface Eth interfaces (comma-separated, no spaces)

Optional OPTIONS -c, --count <number> Core count.
-h, --help Display help and exit.
```

.

9.1.4.9.2 Test Setup OpenFastPath (fpm & fpm_burstmode)



9.1.4.9.3 Running fpm and fpm_burstmode applications

Execute below commands on DUT to run "fpm" and "fpm_burstmode" applications on the LS1088ardb boards:

```
$ cd /usr/ofp/bin
<Execute the application>
$ ./fpm -i <intf-1>,<intf-2> -c 8 (For running fpm application)
or
$ ./fpm_burstmode -i <intf-1>,<intf-2> -c 8  (For running fpm_burstmode application)

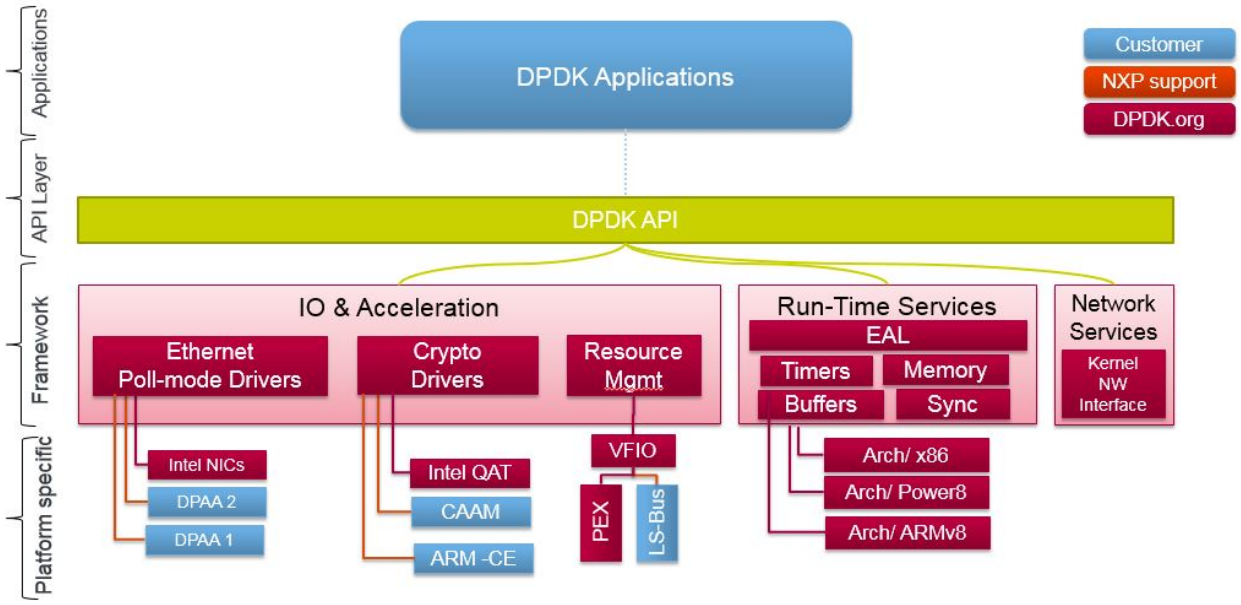
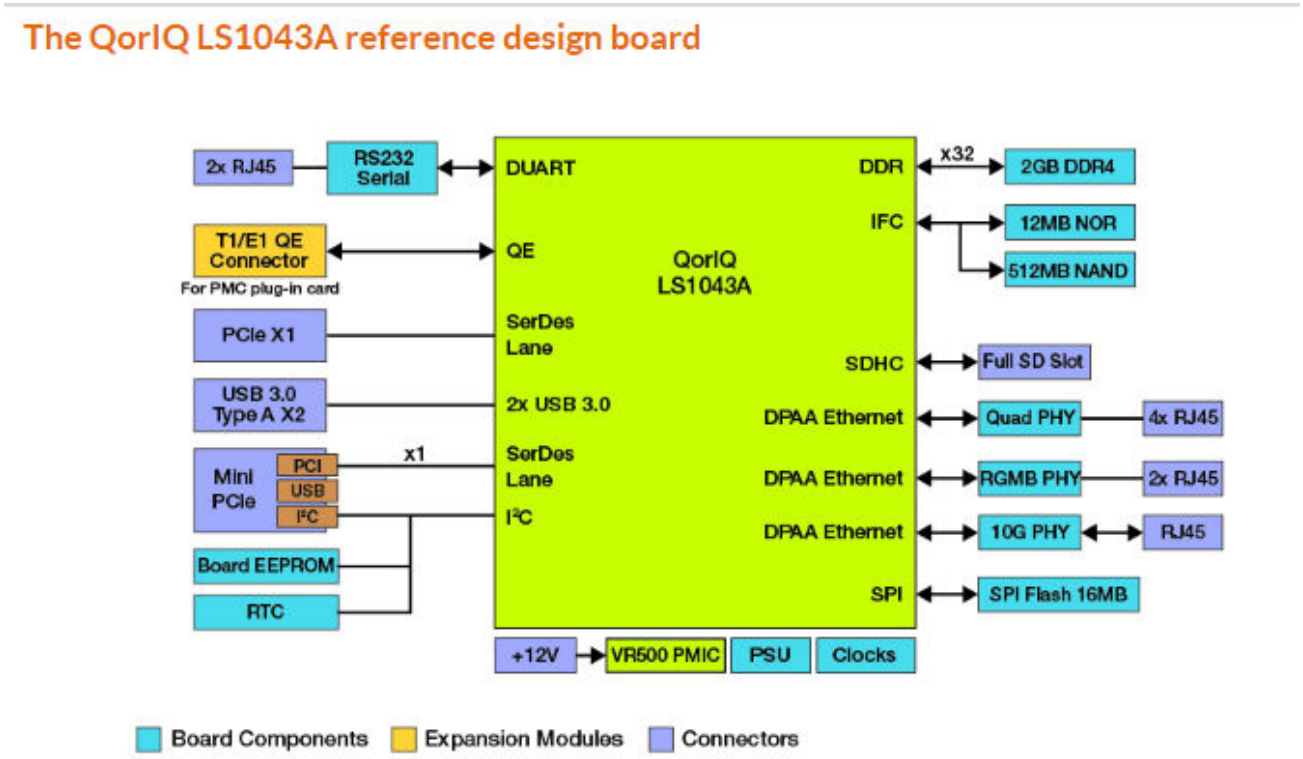
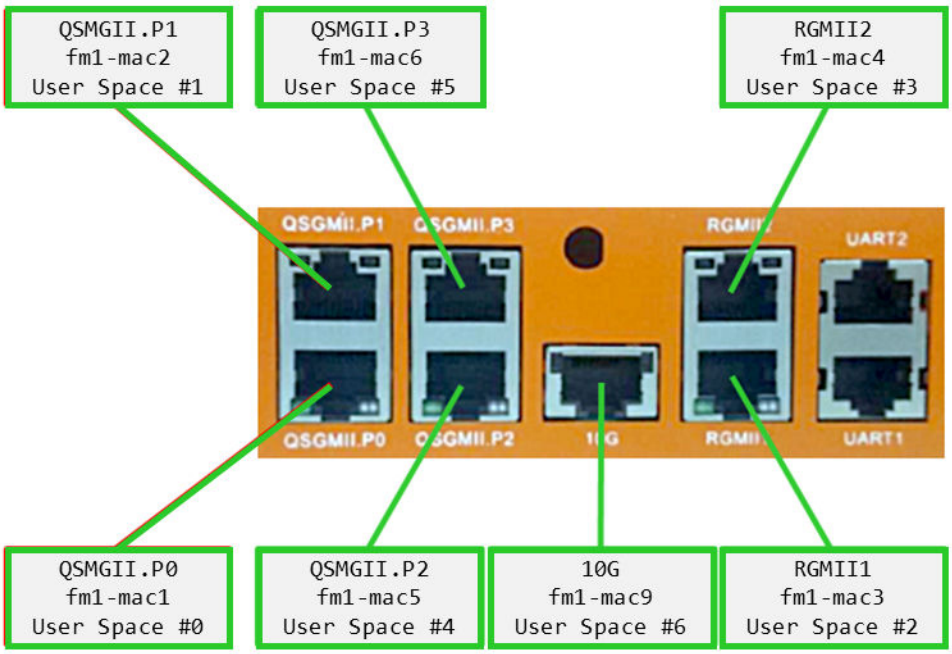
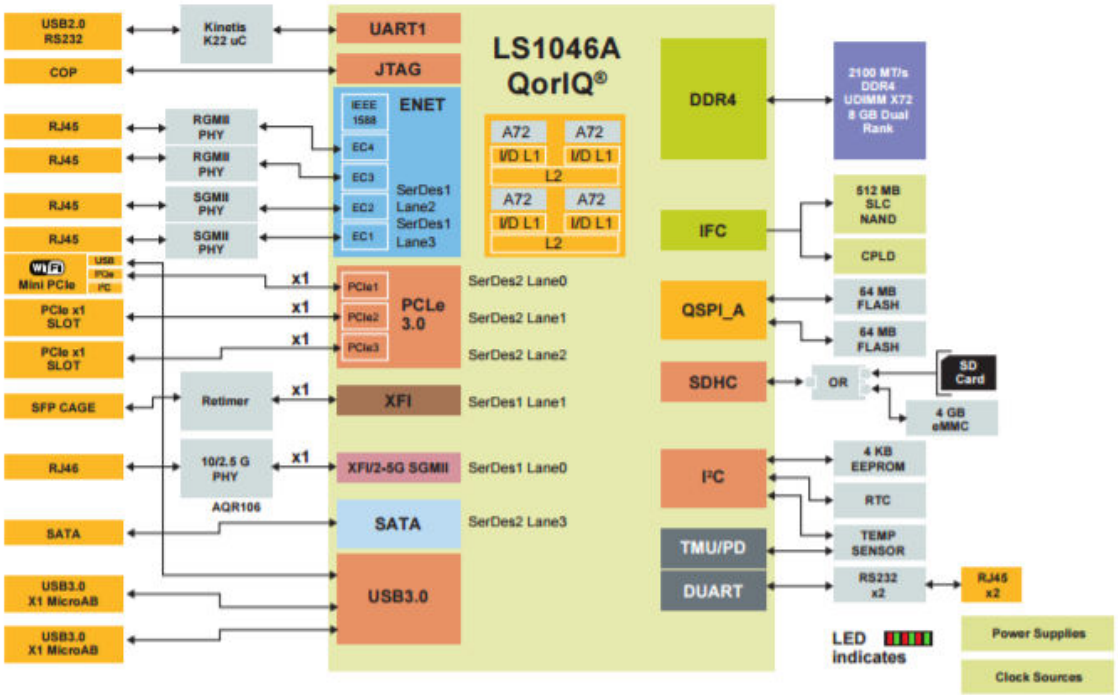
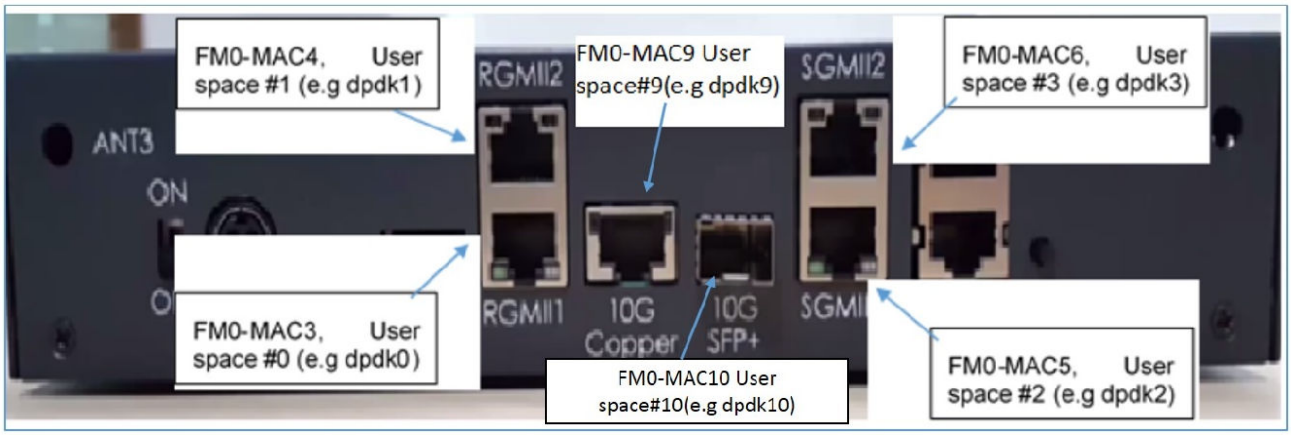
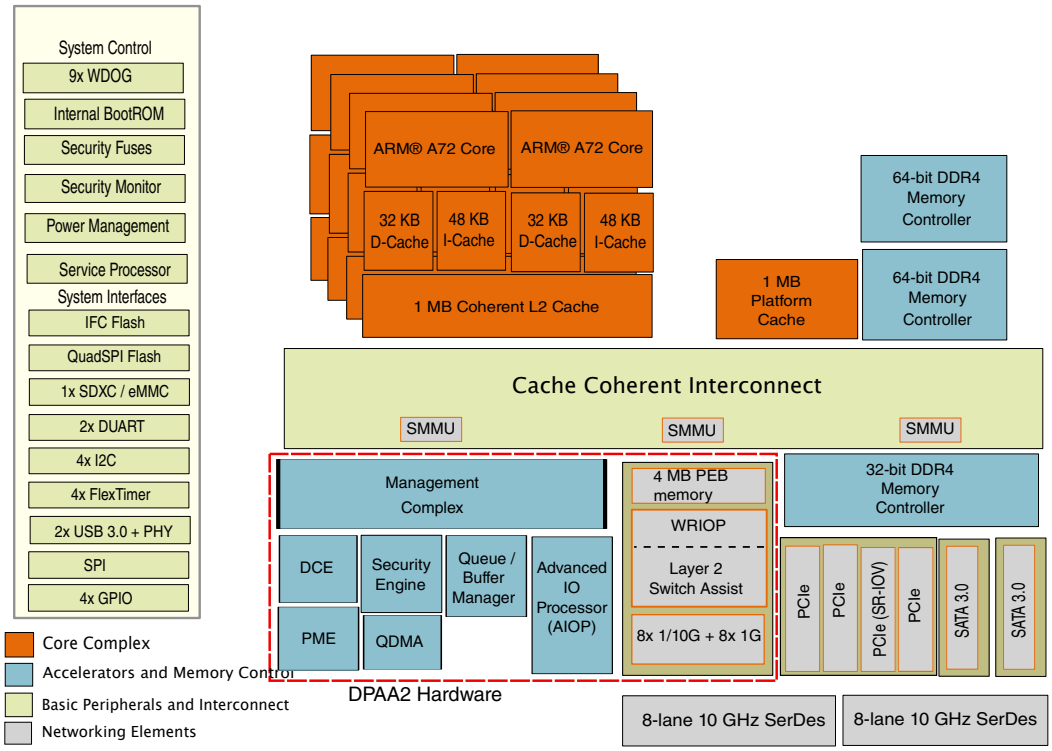
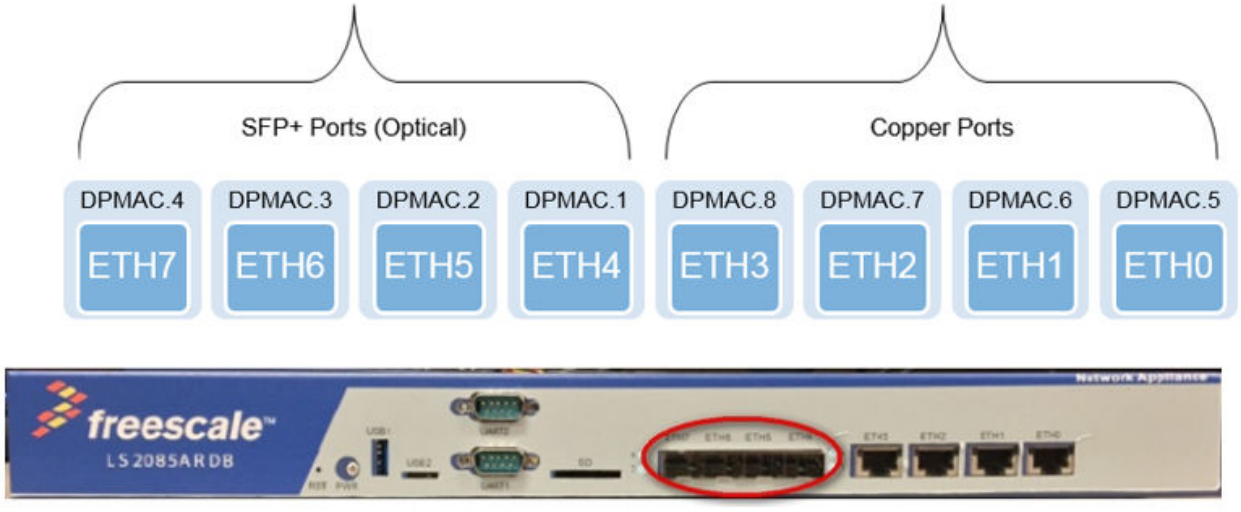
<Assign IP Address to the TAP devices created for each interface used>
$ ifconfig fp0 192.168.111.1/24 up
$ ifconfig fp1 192.168.222.1/24 up
$ echo 1 > /proc/sys/net/ipv4/ip_forward
```

NOTE

DPCI_COUNT must be increased to 128 to run these applications. This can be done by setting the environment variable using "export DPCI_COUNT=128" command before running the dynamic_dpl.sh. Additionally for fpm_burstmode following environment variables needs to be set " export MAX_TCS=8" "export OPR_ENABLE=0" "export FS_ENTRIES=1 ". Interfaces intf-1 can be dpni.<index>. See LS2088ARDB/LS2085ARDB Board Preparation and Bring-up for interface details.

9.1.4.9.4 Test description-ODP OpenFastPath (fpm & fpm_burstmode)

When a packet is received on one of the input interfaces, it will be routed and forwarded according to the forwarding algorithm. The packet will be changed - destination Ethernet address will be updated according to received command line options, the TTL (Hop Limit) will be decremented and the checksum will be recalculated. Finally the packet will be transmitted on the outgoing interface.

Packet format

Stream 1 on interface#1: IPv4 stream, SIP 192.168.111.2 DIP 192.168.222.2

Stream 2 on interface#2: IPv4 stream, SIP 192.168.222.2 DIP 192.168.111.2

Note: User can create desired routes for different Source and Destination IPs using following commands on the board:

```
$ route add -net <subnet/subnet mask> gw <gw address>
for example, Route for destination Subnet 11.11.11.0 to go out through the DPMAC.2(Refer Setup
diagram) should look like:
 route add -net 11.11.11.0/24 gw 192.168.222.2
```

9.1.5 ODP over Virtual Machine (LS2080A)

On LS2080A, ODP is supported over Virtual Machine (VM) with DPAA2 Ethernet Interfaces directly assigned to the Virtual Machine.

The following ODP applications are supported:

- odp_pktio
- odp_l3fwd

ODP_on_VM Setup:



9.1.5.1 Building Images for Virtual Machine

Refer Build ODP Through Yocto on page 1284 section for installing SDK ISO for ODP.

Execute the commands below to build images for ODP on VM in <INSTALL_PATH_FOR_ISO>/QorIQ-SDK-V2.0-20160527-yocto:

1. `. ./fsl-setup-env -m ls2080ardb`

2. `echo 'ROOTFS_IMAGE = "fsl-image-virt"' >> conf/local.conf`
3. bitbake -c cleansstate odp
4. bitbake -c patch odp
5. Uncomment the below line in file *"<INSTALL_PATH_FOR_ISO>/build_ls2080ardb/tmp/work/ls2080ardb-fsl-linux/odp/git-r0/git/platform/linux-dpaa2/Makefile.inc*"
   a. *# DPAA2_CFLAGS += -DODP_USE_PHYS_IOVA*
6. `bitbake fsl-image-kernelitb`

After compiling successfully, the images will be generated at `'<INSTALL_PATH_FOR_ISO>/build_ls2080ardb/tmp/deploy/images/ls2080ardb/'`.

The following images are used for LS2080ARDB and Virtual Machine Bring-up:

- *u-boot-ls2080ardb.bin*
- *PBL_0x2a_0x41_1867_533_1867.bin (rcw)*
- *mc_9.0.5_ls2080a_20160414.itb*
- *dpc-0x2a41.dtb*
- *dpl-odp.0x2A_0x41.dtb*
- *kernel-fsl-ls2080a-rdb.itb*
- *Image*
- *fsl-image-virt-ls2080ardb.ext2.gz.u-boot*

9.1.5.2 Running Virtual Machine

The virtual machine runs inside the host Linux system and it is launched by an application called QEMU. In order to run a virtual machine instance we have to do some preparation steps:

1. Download the Guest Kernel and rootfs Image from Image server.

2. Mount Huge TLBFS on host so that QEMU can use huge pages.

3. Bind Guest's DPRC to the VFIO driver on host.

Please note that console logs for the Guest Linux do not appear on the host Linux console (i.e UART). The guest logs are exposed through telnet, and they can be accessed by doing telnet on the host boards IP Address (IP_ADDR_BRD) and GUEST_CONSOLE_TELNET_PORT. Each Virtual machine that is run on a single host is allocated a different GUEST_CONSOLE_TELNET_PORT, and this port number is specified by user running Virtual machine through QEMU command line.

Steps for running Virtual machine

1. Log-in into to the host Linux.
2. Assign IP_ADDR_BRD to network interface connected to Image server and to the machine from which guest console is accessed.

```
# ifconfig eth0 <IP_ADDR_BRD>
```

3. Mount Huge TLB FS

   Execute the commands below to initialize the resources required by VM to be run:

```
# echo "hugetlbfs /hugetlbfs hugetlbfs defaults,mode=0777 0 0" >> /etc/fstab
```

```
# mkdir /hugetlbfs
# mount /hugetlbfs
```

4. Create DPRC Containers for VM

   a. Create container for VM kernel

   ```
   # ./vm_dpl_setup.sh vm_dpl_setup_parent.conf
   ```

   b. Create container for VM user space

   ```
   # ./vm_dpl_setup.sh vm_dpl_setup_child.conf <DPMAC1> <DPMAC2>
   ```

   Please note that DPMAC1 and DPMAC2 have to be replaced with the DPMACs which have to be given to DPDK running in VM.

   The configuration files for DPL setup i.e. vm_dpl_setup_child.conf and vm_dpl_setup_parent.conf provide the sample configuration for single VM running on 8 cores. For other configurations, the environment variables in configuration file have to be modified accordingly.

   In case of running two virtual machines, step a) and step b) have to be run twice. Each run will create the container for one virtual machine. In the second run the PARENT_DPRC env variable in vm_setup_child.conf has to be updated with the DPRC created in second run of step a).

5. Check the new IOMMU groups created for child dprc container of Guest VM root container. dprc.1 is the system DPRC root container which always belongs to Host Linux. In this example dprc.2 is the child of dprc.1 and this is going to be the root container of VM.

```
# ls sys/devices/platform/80c000000.fsl-mc/dprc.1/dprc.2/
    dpbp.1     dpcon.1   dpio.1   dpmcp.23   dpmcp.27   dprc.3   power
    dpbp.2     dpcon.2   dpio.2   dpmcp.24   dpmcp.28   driver   rescan
    dpbp.3     dpcon.3   dpio.3   dpmcp.25   dpmcp.29   driver_override  subsystem
    dpbp.4     dpcon.4   dpio.4   dpmcp.26   dpni.1     iommu_group  uevent

 DPRC.3 is the only child container of DPRC.2, Check the iommu-group of dprc.3
# readlink /sys/devices/platform/80c000000.fsl-mc/dprc.1/dprc.2/dprc.3/iommu_group

Output
../../../../../../kernel/iommu_groups/7 IOMMU group “6” and “7” should be available in /dev/
vfio/

# ls /dev/vfio/
6 7  vfio
```

NOTE

:- dprc.2 and dprc.3 are resource container that contain pre-allocated resources like ethernet interfaces to be used by virtual machines.

   The child’s IOMMU group has to be given to VM while launching. In above example child’s IOMMU group 7 has to be given to VM during launch. Child’s IOMMU group is referred as CHILD_IOMMU_GROUP in further steps.

6. Go to a directory that has space to copy files. Copy Guest linux kernel and Guest root file system to the directory:

```
#  cd /tmp/
#  scp <USER>@<IP_ADDR_IMAGE_SERVER>:<TFTP_BASE_DIR>/Image   .
#  scp <USER>@<IP_ADDR_IMAGE_SERVER>:<TFTP_BASE_DIR>/ fsl-image-core-ls2085ardb.ext2.gz .
```

7. Launch the Virtual Machine

```
 #   qemu-system-aarch64 -m 4096 -nographic -mem-path /hugetlbfs -cpu host-machine type=virt -
kernel ./Image -enable-kvm -initrd ./fsl-image-ls2085ardb.ext2.gz
        -append 'root=/dev/ram0 rw console=ttyAMA0,115200 rootwait earlyprintk
ramdisk_size=1000000' -serial tcp::4446,server,telnet
        -device vfio-fsl-mc,host=dprc.2 -device vfio-smmu,x-group=<CHILD_IOMMU_GROUP> -smp 8  -S
```

NOTE

- Please note that GUEST_CONSOLE_TELNET_PORT is given as 4446 in above example, if you are running two virtual machines you can give 4446, 4447 to the two virtual machines respectively/
- Value of CHILD_IOMMU_GROUP is 7 in the example that is taken in this document.

The following logs appear on Host UART console upon launching the Virtual Machine

```
QEMU 2.4.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server
qemu-system-aarch64: -device vfio-fsl-mc,host=dprc.2: Failed to create KVM VFIO device: No such
device
qemu-system-aarch64: -device vfio-fsl-mc,host=dprc.2: Failed to create KVM VFIO device: No such
device
Only One Root DPRC can exists
```

Please note that QEMU logs come completely only when you log-in to the guest machine using telnet as specified in section "Accessing virtual machine console".Also note that The log saying "Failed to create KVM VFIO device: No such device" is currently a known issue, it can be ignored all functionality works even with this log.The –S option mentioned in the qemu command stops the Virtual Machine bootup after initial setup. The QEMU threads needs to be affined to cores sequentially (this is needed because of few H/W limitations). Run 'info cpus' command on QEMU CLI interface to see the QEMU threads.

Console To access the console for launched VM, telnet to the Host Linux ip address with telnet port as specified in the QEMU command while launching the VM

```
$ telnet <IP_ADDR_BRD> 4446
Trying 192.168.1.102...
Connected to 192.168.1.102.
Escape character is '^]'.
```

Console print on UART where VM was launched After connecting to the VM through telnet, qemu command prompt will appear:

```
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server
qemu-system-aarch64: -device vfio-fsl-mc,host=dprc.2: Failed to create KVM VFIO device: No such
device
qemu-system-aarch64: -device vfio-fsl-mc,host=dprc.2: Failed to create KVM VFIO device: No such
device
Only One Root DPRC can exists
[Press Enter to drop to Qemu shell]
(qemu)
```

Affine VM core threads to Physical CPU

The Virtual Machine core threads needs to be affined to physical cores sequentially to avoid VM core migrating on physical cores.

Run ‘`info cpus command on QEMU CLI interface to get the VM threads ids`

```
 cid="dWNBT">(qemu) info cpus
* CPU #0: thread_id=<tid1>
  CPU #1: (halted) thread_id=<tid2>
  CPU #2: (halted) thread_id=<tid3>
  CPU #3: (halted) thread_id=<tid4>
  CPU #4: (halted) thread_id=<tid5>
  CPU #5: (halted) thread_id=<tid6>
  CPU #6: (halted) thread_id=<tid7>
  CPU #7: (halted) thread_id=<tid8>
```

Open a new SSH session to the host Linux to affine VM threads to physical cores ---------- Affine core for VM on physical cores 0,1,2,3,4,5,6,7 ----------

```
~# taskset -p 0x1 <tid1>
~# taskset -p 0x2 <tid2>
~# taskset -p 0x4 <tid3>
~# taskset -p 0x8 <tid4>
~# taskset -p 0x10 <tid5>
~# taskset -p 0x20 <tid6>
~# taskset -p 0x40 <tid7>
~# taskset -p 0x80 <tid8>
```

*******Continue launching and booting the Virtual Machine********

On the console where qemu was launched, run

```
(qemu) c
```

Once the VM is launched, prompt will come on telnet console

```
---------Booting Linux on physical CPU 0x0-----------
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 4.1.8+gda8455a (b27504@b27504-OptiPlex-790) (gcc version 4.9.3
20150311 (prerelease) (Linaro GCC 4.9-2015.03) ) #1 SMP PREEMPT Fri May 20 16:59:34 IST 2016
[    0.000000] CPU: AArch64 Processor [411fd071] revision 1
[    0.000000] Detected PIPT I-cache on CPU0
[    0.000000] alternatives: enabling workaround for ARM erratum 832075
[    0.000000] alternatives: enabling workaround for ARM erratum 834220
[    0.000000] efi: Getting EFI parameters from FDT:
……
…….
……
…….
Starting kernel log daemon...0
Starting internet superserver: xinetd.

QorIQ SDK (FSL Reference Distro) 2.0 ls2080ardb /dev/ttyAMA0

ls2080ardb login:
```

8. Launching two Virtual machines

   In case of two virtual machines, please take care of following:

   - Memory assigned to each virtual machine should not exceed the total number of huge pages assigned on system. In above example (step 8) giving 2048 Mega bytes to each virtual machine works fine.

- Console telnet port of both virtual machine must be different, for example 4446 could be console of VM1 and 4447 could be console for VM2.

Following are the example launch commands for two VMs:

VM1:

```
qemu-system-aarch64 -m 2048 -nographic -mem-path /hugetlbfs -cpu host -machine type=virt -
kernel ./Image -enable-kvm -initrd ./fsl-image-ls2085ardb.ext2.gz -append 'root=/dev/ram0 rw
console=ttyAMA0,115200 rootwait earlyprintk ramdisk_size=1000000' -serial tcp::
4446,server,telnet -device vfio-fsl-mc,host=dprc.2 -device vfio-smmu,x-
group=<CHILD_IOMMU_GROUP> -smp 8 –S
```

VM2:

```
qemu-system-aarch64 -m 2048 -nographic -mem-path /hugetlbfs -cpu host -machine type=virt -
kernel ./Image -enable-kvm -initrd ./fsl-image-ls2085ardb.ext2.gz -append 'root=/dev/ram0 rw
console=ttyAMA0,115200 rootwait earlyprintk ramdisk_size=1000000' -serial tcp::
4447,server,telnet -device vfio-fsl-mc,host=dprc.2 -device vfio-smmu,x-
group=<CHILD_IOMMU_GROUP> -smp 8 -S
```

ConsoleTo access the console for launched VM, telnet to the Host Linux ip address with telnet port as specified in the QEMU command while launching the VM machine.

9.1.5.3 Running ODP Application in VM

Resources Allocated to VM

Physical interfaces are assigned to VM through the resource container (dprc.x) assigned to the VM while executing the QEMU command

Data**P**ath**L**ayout (dpl) file used to boot up the system contains two resource containers dprc.2 and dprc.4. Below table lists the interfaces allocated to these two resource containers and below mentioned interface names shall be used while executing the ODP applications.

Host - dprc.2 **(dprc.3 – vm dprc)**			**Host dprc.4** **(dprc.5 – vm dprc)**		
Physical Port	*Port Name to be used in ODP*	*Mac Address*	*Physical Port*	*Port Name to be used in ODP*	*Mac Address*
eth4	dpni-1	02:00:c0:a8:47:01	eth6	dpni-3	02:00:c0:a8:47:03
eth5	dpni-2	02:00:c0:a8:47:02	eth7	dpni-4	02:00:c0:a8:47:04

Execute the commands below on the Virtual Machine before running the ODP application:

```
# echo 512 > /proc/sys/vm/nr_hugepages
# cd /usr/odp/scripts/
# export DPRC=<dprc.x>
# ./bind_dprc.sh <dprc.x>
# export HOST_START_CPU=0
```

NOTE

For <*dprc.x*> use:

- *'dprc.3' for VM launched with dprc.2*
- *'dprc.5' for VM launched with dprc.4*

For *HOST_START_CPU :*

- *First phyical core which is assigned to VM. Default value is 0*

See Using ODP Applications for instructions on how to run ODP Applications.

9.1.6 Troubleshooting

- Please ensure to use correct bootargs:

```
bootargs console=ttyS1,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0600,115200
ramdisk_size=0x2000000 default_hugepagesz=2m hugepagesz=2m hugepages=256
```

- In case application is not able to run please ensure dpni.<index> is used correctly in the command line.

In case packets are not flowing:

- Ensure that traffic generator to board connectivity is proper.
- Ensure that the stream is configured properly having the destination MAC address of that interface where the stream is destined.
- Ensure that there should be no space when interface are specified in application command.

If multiple instances of ODP applications need to be run:

Make sure to set APPL_MEM_SIZE environment variable before running the application to specify the memory to be used by application. The size is specified in MegaBytes e.g in case application plans to use 32MB of memory, then following command can do reservation:

```
export APPL_MEM_SIZE=32
```

9.1.7 Using Debug Tool to Get Hardware Statistics for DPAA2 Platforms

Using DEBUG TOOL to get hardware stats for DPAA2 platforms:

plat_debug_tool is used to display stats of hardware blocks like QBMAN, Crypto, etc.

How to use plat_debug_tool

NOTE

The plat_debug_tool is only applicable to DPAA2 platforms.

1. Enabling Debug Server in ODP applications

By default debug server is disabled in ODP applications. To enable debug server, execute following commands

```
$ export PLAT_DEBUG_THREAD=1
$ export PLAT_DEBUG_PORT=<x>
```

- *<x> : UDP port number on which debug server will listen upon.*
- *Valid range: 1024 to 65535.*

- *Default port is 10000.*

2. Running plat_debug_tool client

Before running plat_debug_tool, execute following command:

```
$ export PLAT_DEBUG_PORT=<x>
```

- *<x> : UDP port number on which client will bind to.*
- *Default port is 10000.*

NOTE

UDP port number should be same as given for server.

For running plat_debug_tool, execute following command.

```
$ ./plat_debug_tool -d dpni.1 -o 0 -c 0 -i 192.168.10.10 (Reference Command)
```

- Usage: ./plat_debug_tool OPTIONS
  - Mandatory OPTIONS:
    - -d, --device Device name like dpni.1, dpbp.1

      NOTE

      Execute restool command to identify the HW devices used by ODP application

      Supported devices:
      - dpni.x - for hardware interfaces
      - dpbp.x - for buffer pools
      - dpseci.x - for ipsec

      NOTE

      option not required for object ID 20 (will be ignored if given)

    - -o, --obj_id all: Display all the stats for a given device.

      Stats for dpni.x
      - 0: Dpni Stats
      - 1: Dpni Attributes
      - 2: Dpni Link State
      - 3: Dpni Max Frame Length
      - 4: Dpni MTU
      - 5: L3 chksum hardware offload (enable/disable)
      - 6: L4 chksum hardware offload (enable/disable)
      - 7: Dpni Primary Mac Addr
      - 8: Congestion Group Id for FQs
      - 9: Scheduling Priority for FQs
      - 10: Tail Drop Threashold for FQs
      - 11: FQ Context
      - 12: FQ State

Stats for dpbp.x

- 13: Qbman frame count
- 14: Qbman byte count
- 15: Qbman has free buffers or not
- 16: Qbman buffer pool is depleted or not
- 17: Number of free buffers in qbman

Stats for dpsec.x

- 18: DPseci Attributes
- 19: DPseci counters
- 20: Per SA stats (only for odp_ipsec and odp_ipsec_proto apps)

NOTE

To use per SA stats option, It is required to enable the ipsec-debugs at compile time. Following are the steps to enable the ipsec-debugs in yocto:

1. Go to the following path:
   a. cd sources/meta-fsl-dataplane/recipes-extended/odp
2. Add a following line in the file "odp_git.bb"
   a. For board ls2080ardb:
      i. EXTRA_OECONF_append_ls2080ardb += "--enable-ipsec-debug=yes"
   b. For board ls2088ardb:
      i. EXTRA_OECONF_append_ls2088ardb += "--enable-ipsec-debug=yes"
3. Go back to the build directory and run the following commands to recompile the ODP:
   a. bitbake -f -c clean odp
   b. bitbake -f -c compile odp

- -c, --command
  - 0: get
  - 1: reset
- -i, --dest_ip
  - IP address of debug server
  - Default: local host (127.0.0.1)
- -h, --help
  - Display help and exit.

9.2 DPDK User Manual

9.2.1 Introduction

Scope

This document contains instructions for installing and configuring the user space based Data Plane Development Kit (DPDK) software. It is designed to get customers up and running quickly. The document describes how to compile and run a DPDK application in a Linux application (linuxapp) environment, without going deeply into detail.

Intended audience

This document is intended for software developers and architects who want to use or develop DPDK API complaint user space applications on QorIQ based platforms. The document assumes that users are familiar with Linux-based software development, and also with the DPDK concepts and APIs.

9.2.1.1 Supported Platforms

- LS1043ARDB
- LS1046ARDB
- LS2088ARDB

9.2.1.2 Definitions and Acronyms

Definition and acronyms

Acronym/Term	Description
DPDK	Data Plane Development Kit
RCW	Reset and control word
Host	The term ‘Host’ is used for Linux running on Layerscape board directly
Guest/VM	The term ‘Guest’ is used for Linux running inside the virtual machine(s) that are in turn running over Host Linux operating system.VM and Guest have been used interchangeably in this guide
IP_ADDR_BRD	This term is used for IP address of LS1 and LS2 RDB boards
IP_ADDR_IMAGE_SERVER	This term is used for IP address of the machine on which all the software images are kept
TFTP_BASE_DIR	Base directory of TFTP server where all the images are kept
GUEST_CONSOLE_TELNET_PORT	Telnet port for accessing guest console
DPAA2	Data Path Acceleration Architecture Gen 2. Steps/text marked with this marker are applicable for DPAA2 platforms
DPAA	Steps/Test marked with this marker are applicable for DPAA platforms

9.2.1.3 References

Document	Link	Version	Description
LS1043ARDB	LS1043A-RDB: QorIQ LS1043A Reference Design Board		
LS1046ARDB	LS1046A-RDB: QorIQ LS1046A Reference Design Board		
LS2088ARDB	LS2088A-RDB: QorIQ LS2088A Reference Design Board		
L2fwd	L2fwd usage	NA	L2fwd usage and setup
L2fwd-Crypto	L2fwd-crypto	NA	L2fwd-crypto usage and setup
L3fwd	L3fwd usage	NA	L3fwd usage and setup
Ipsec-secgw	Ipsec-secgw usage	NA	Ipsec-secgw usage and setup

9.2.2 DPDK Overview

The main goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. Users may use the code to understand some of the techniques employed, to build upon for prototyping or to add their own protocol stacks.

The framework creates a set of libraries for specific environments through the creation of an Environment Abstraction Layer (EAL), which may be specific to a mode of the specific platform. These environments are created through the use of make files and configuration files. Once the EAL library is created, the user may link with the library to create their own applications. Other libraries, outside of EAL, including the Hash, Longest Prefix Match (LPM) and rings libraries are also provided. Sample applications are provided to help show the user how to use various features of the DPDK.

The DPDK implements a run to completion model for packet processing, where all resources must be allocated prior to calling Data Plane applications, running as execution units on logical processing cores. The model at present does not support a scheduler and all devices are accessed by polling. The primary reason for polling is the performance overhead imposed by interrupt processing. On the flip side, it causes the the core to be 100% utilized always even on low traffic scenario.

In addition to the run-to-completion model, a pipeline model may also be used by passing packets or messages between cores via the rings. This allows work to be performed in stages and may allow more efficient use of code on cores.

More information on general working of DPDK can be found at: http://dpdk.readthedocs.io/en/v16.07/prog_guide/index.html

9.2.2.1 DPDK DPAA Platform Support

The NXP Data Path Acceleration Architecture comprises a set of hardware components which are integrated via a hardware queue manager and use a common hardware buffer manager. Software accesses the DPAA via hardware components called "*Software Portals or DPIO*". These directly provide queue and buffer manager operations such as enqueues, dequeues, buffer allocations, and buffer releases and indirectly provide access to all of the other DPAA hardware components via the queue manager.



Figure 197. DPDK Architecture with NXP DPAA Components

NXP DPAA architecture based *PMD (Poll Mode Drivers)* has been added to DPDK infrastructure to support seamless working on NXP platform. With the addition of these drivers, DPDK framework on NXP platforms permits Linux user space applications to be build using standard DPDK APIs in a portable fashion. The drivers directly access the DPAA queue and buffer manager software portals in a high performance manner and the internal details remains hidden from higher level DPDK framework. The *PMD*s has also been added for security co-processor as `dpaa2_caam`.

Some of these changes to support NXP DPAA architecture in DPDK are not yet part of Open Source DPDK Repository. Efforts are underway to publish the complete *PMD* code in the upstream repository.

Multithread environment

DPDK was originally designed for Intel architectures, however efforts are underway to make it multiple architecture friendly. Still, there are some restrictions, which should be taken care, when used on NXP platforms.

1. Multiple pthreads

   DPDK usually pins one pthread per core to avoid the overhead of task switching. This allows for significant performance gains, but lacks flexibility and is not always efficient. DPDK is comprised of several libraries. Some of the functions in these libraries can be safely called from multiple threads simultaneously, while others cannot. This section allows the developer to take these issues into account when building their own application.

   The run-time environment of the DPDK is typically a single thread per logical core. Typically, it is best to avoid sharing data structures between threads and/or processes where possible. Where this is not possible, the execution blocks must access the data in a thread-safe manner. Mechanisms such as atomics or locking can be used to allow execution blocks to operate serially. However, this can have an effect on the performance of the application.

2. Fast-path APIs

   Applications operating in the data plane are performance sensitive but certain functions within those libraries may not be safe to call from multiple threads simultaneously.

   The Hash, LPM, Mempool libraries and RX/TX in the PMD are examples of such multithread unsafe functions. The RX/TX of the *PMD* are the most critical aspects of a DPDK application and it is recommended that no locking be used with these paths as it will impact performance. Note, however, that these functions can safely be used from multiple threads when each thread is performing I/O on a different NIC queue. If multiple threads are to use the same hardware queue on the same NIC port, then locking or some other form of mutual exclusion is necessary. In the NXP implementation, each thread has to use a software portal (DPIO) instance to access the underlying DPAA hardware. Thus, it is recommended that only one thread per logical core should be created for RX/TX and other I/O access to DPAA hardware.

Since this guide contains support for both DPAA2 and DPAA platforms, the following markers are used throughout the guide:

- DPAA2 – This marker marks the steps/text applicable only for DPAA2 platforms. e.g. LS2088
- DPAA – This marker marks the steps/text applicable only for DPAA platforms. e.g. LS1043

All other steps which don't have any marker are applicable for both platforms.

9.2.3 Build DPDK

For detailed information on ISO installation, see "Essential Build Instructions" under "Getting Started" within the QorIQ SDK documentation.

9.2.3.1 Standalone compilation of DPDK framework and example binaries

A limited set of DPDK binaries are already part of the Root filesystem used for booting up the LSXXXXA RDB. Steps mentioned in below sections are for illustrating standalone DPDK framework and application compilation.

DPDK Sources can be obtained as per the Release Note. Following steps assume that DPDK sources are available from notified location.

Compilation Steps

1. Common commands to be run for both DPAA2 and DPAA platforms:

```
# export KERNEL_PATH=<kernel path>
# export CROSS_COMPILE=<toolchain path>
```

2. DPAA2 and DPAA specific commands

   a. DPAA2

```
#source standalone-dpaa2
(Change the standalone script as per your environment)
# make T=arm64-dpaa2-linuxapp-gcc install
```

   b. DPAA

```
#source standalone-dpaa
(Change the standalone script as per your environment)
# make T=arm64-dpaa-linuxapp-gcc install
```

NOTE

Depending on the platform for which build is targeted (DPAA2 or DPAA) either 2.a or 2.b is required, both steps should not be used in same compilation iteration.

3. DPDK sample application compilation (same steps for both DPAA and DPAA2 platforms)

   For compiling the example applications, following steps can be used: (only steps for l2fwd and l3fwd applications are shown below, but similar steps would be application for other applications available as part of DPDK framework):

```
# make -C examples/l3fwd
# make -C examples/l2fwd
# testpmd and test applications get compiled along with DPDK framework
```

Location of binaries

```
examples/l2fwd/build/l2fwd
examples/l3fwd/build/l3fwd
arm64-dpaa2-linuxapp-gcc/build/app/test-pmd/testpmd
```

The ARMCE driver compilation is enabled by default, and it needs openssl libraries for compilation. Follow the steps to compile DPDK with ARMCE:

The code base of openSSL can be downloaded from OpenSSL Git. Use tag “OpenSSL_1_0_2h” for checkout the code and build openSSL using the following commands:

```
# ./configure linux-aarch64 --prefix=<OpenSSL library path> shared
# make
# make install# export OPENSSL_PATH=<OpenSSL library path>
```

Modify `config/defconfig_arm64-dpaa2-linuxapp-gc`c to `enable/disable`ARMCE:

```
CONFIG_RTE_LIBRTE_PMD_ARMCE=n
```

Recompile DPDK after any change in the config files.

If you are not using standard rootfs, you may need the openssl libraries (libcrypto.so.1.0.0 and libssl.so.1.0.0) to be copied at location `/lib` and `/usr/lib` respectively on the board.

After compilation of DPDK (with ARMCE), compile the l2fwd-crypto and test application as shown below.

L2fwd-crypto

```
# make -C examples/l2fwd-crypto
```

L2fwd-crypto binary is built in `examples/l2fwd-crypto/build/`

ipsec-secgw

```
# make -C examples/ipsec-secgw
```

Ipsec-secgw binary is built in `examples/ipsec-secgw/build/`

Test

```
# make -C app/test
```

test binary is built in `app/test/build/app/`

9.2.3.2 Yocto based Build

DPDK applications can also be compiled alone using Yocto once you have ran "*bitbake fsl-image-kernelitb*". The steps are as follows:

1. cd build_<board-name>/
2. source SOURCE_THIS
3. Clean DPDK: bitbake -f -c cleansstate dpdk
4. Make DPDK: bitbake -f -c compile dpdk

   Note: the step above will only compile the dpdk library and applications. To add them to rootfs and to create a Kernelitb, use the command given below.
5. bitbake fsl-image-kernelitb

Note: For dpdk virtual kernel .itb generation use below command before starting “bitbake fsl-image-kernelitb”

echo 'ROOTFS_IMAGE = "fsl-image-virt"' >> conf/local.conf

9.2.3.3 Pktgen

Compiling Pktgen

Get pktgen-dpdk sources:

Pktgen source has to be cloned from upstream repository and following steps have to be followed to build pktgen:

```
# git clone git://dpdk.org/apps/pktgen-dpdk => do the make install in DPDK.
# make sure the CROSS_COMPILE path is set.
# export RTE_SDK=<DPDK Source DIR>
```

For DPAA2 platforms set `RTE_TARGET=arm64-dpaa-linuxapp-gcc`:

```
# export RTE_TARGET=arm64-dpaa2-linuxapp-gcc
```

For DPAA platforms set `RTE_TARGET=arm64-dpaa-linuxapp-gcc`:

```
# export RTE_TARGET=arm64-dpaa-linuxapp-gcc
```

To build:

```
# make
```

Copy "`Pktgen.lua`" and "`pktgen`" (available at: `app/app/arm64-dpaa2-linuxapp-gcc/pktgen`) to the board.

NOTE

Pktgen is not yet integrated with Yocto based build, so it has to be built standalone as described in this section.

9.2.4 Software Images and Overview

The list of images below are used to run DPDK and related applications on the target platform.

NOTE

The tables given in the following subsections contain:

*MACRO/name is used for the image (“*_IMG”) - this name is used in this document wherever the image should be referenced.

*Non-specific platform suffix is given as ‘X’ - exact name of image for a platform is defined in platform subsection for each platform in

The images which are common for all platforms i.e images without any “X” suffix are mentioned in this chapter only with their correct names applicable for all platforms, thus they are not mentioned in

9.2.4.1 Platform Images

Images common for both DPAA and DPAA2 platform:

S/No	File/Image Name	Macro/Name used for this image within the guide	Description
1	u-boot-<X>.bin	PLAT_UBOOT_IMG	u-boot binary
2	rcw_1600.bin or PBL_0x2a_0x41_1867_533_1867.bin	PLAT_RCW_IMG	RCW Binary
3	kernel-fsl-<X>.itb	PLAT_KRNL_IMG	ITB binary containing Linux kernel binary and Root file system.

Images applicable only to DPAA platforms:

S/No	File/Image Name	Macro/Name used for this image within the guide	Description
1	fsl_fman_ucode_<X>.bin	DPAA_FMAN_UCODE_IMG	FMAN Binary
2	ppa.itb	DPAA_PPA_IMG	PPA Binary

Images applicable only to DPAA2 platforms:

S/No	File/Image Name	MACRO/NAME used for image within the guide	Description
1	mc_<version>_<X>.itb	DPAA2_MC_IMG	Management complex binary
2	dpc-0x2a41.dtb	DPAA2_DPC_IMG	DPC Image
3	dpl-eth.0x2A_0x41.dtb	DPAA2_DPL_IMG	DPL Image

9.2.4.2 Virtual machine (VM or guest) images

S/No	File/Image name [with absolute path in target platform rootfs]	MACRO/NAME used for image throughout the guide	Description
1	/boot/Image	VM_KRNL_IMG	Virtual machine's Kernel Image
2	/boot/fsl-image-core-<X>.ext2.gz	VM_ROOTFS_IMG	Virtual machine's Root file system image

9.2.4.3 DPDK and application images

S/No	File/Image name [absolute path in target platform rootfs]	Description

Table continues on the next page...

Table continued from the previous page...

1	`/usr/bin/dpdk-example/l2fwd`
	`/usr/bin/dpdk-example/l3fwd`
	`/usr/bin/dpdk-example/l2fwd-crypto`
	`/usr/bin/dpdk-example/testpmd`
		DPDK Example applications

2	`/usr/bin/dpdk-example/extras/usdpaa_config_ls<PLAT>.xml`
	`/usr/bin/dpdk-example/extras/usdpaa_policy_hash_ipv4_1queue.xml`
	`/usr/bin/dpdk-example/extras/usdpaa_policy_hash_ipv4_2queue.xml`
	`/usr/bin/dpdk-example/extras/usdpaa_policy_hash_ipv4_4queue.xml`
		DPAA
		FMC Configurations and Policy files. These are applicable only for DPAA platforms. Plat is platform name for DPAA platform for example ls1043 or ls1046

3	`/usr/bin/dpdk-example/extras/dynamic_dpl.sh`	DPAA2 Dynamic DPL container creation script. This is applicable only for DPAA2 platforms
4	`/usr/sbin/dpdk_nic_bind`	DPDK NIC binding utility. This is only applicable for executing DPDK applications in VM.
5	`/usr/bin/ovs-dpdk/ovsdb-tool`	OVS database tool image
6	`/usr/bin/ovs-dpdk/ovsdb-server`	OVS data base server
7	`/usr/bin/ovs-dpdk/ovs-vsctl`	OVS switch control tool
8	`/usr/bin/ovs-dpdk/ovs-vswitchd`	OVS switch daemon
9	`/usr/bin/ovs-dpdk/ovs-ofctl`	OVS flow control tool

NOTE

DPAA1

For DPAA1, OVS images (`/usr/bin/ovs-dpdk`) are only included in the kernel itb with virtualization support i.e. the kernel itb is created by changing rootfs using the following command:

```
$ echo 'ROOTFS_IMAGE = "fsl-image-virt"'>>conf/local.conf
```

9.2.5 Supported Platforms and Platform-specific Details

NOTE

DPDK in this SDK release supports DPAA platforms (LS1043A and LS1046A), and DPAA2 platform LS2088A.

9.2.5.1 LS1043A Reference Design Board (RDB)

LS1043A is a DPAA-based platform. Follow all the optional steps marked for DPAA platforms in this document. For more information on LS1043ARDB, see www.nxp.com/LS1043ARDB

Hardware specifications



Figure 198. QorIQ LS1043A Reference Design



Figure 199. LS1043A Port Layout

Software image names

S/No	File name for this platform	MACRO/name used for image throughout this guide	Description
1	`u-boot-ls1043ardb.bin`	PLAT_UBOOT_IMG	U-Boot Image
2	`rcw_1600.bin`	PLAT_RCW_IMG	RCW Image
3	`kernel-fsl-ls1043a-rdb-usdpaa.itb`	PLAT_KRNL_IMG	Host kernel Image
4	`fsl_fman_ucode_ls1043_r1.1_106_4_18.bin`	DPAA_FMAN_UCODE_IMG	FMAN microcode image
5	`fsl-image-core-ls1043ardb.ext2.gz`	VM_ROOTFS_IMG	Virtual machine's Root FS image.

9.2.5.2 LS1046A Reference Design Board (RDB)

LS1046A is a DPAA-based platform. Follow all the optional steps marked for DPAA platforms in this guide. For more information on QorIQ LS1046A, see www.nxp.com/LS1046ARDB.

Hardware specifications



Figure 200. QorIQ LS1046A Reference Design



Figure 201. LS1046A Port Layout

Software image names

S/No	File name for this platform	MACRO/name used throughout this section	Description
1	`u-boot-qspi.bin`	PLAT_UBOOT_IMG	U-Boot Image
2	`rcw_1600_qspiboot.bin` or `rcw_1800_qspiboot.bin`	PLAT_RCW_IMG	RCW Image
3	`kernel-fsl-ls1046a-rdb-usdpaa.itb`	PLAT_KRNL_IMG	Host kernel Image
4	`fsl_fman_ucode_ls1046_r1.0_106_4_18.bin`	DPAA_FMAN_UCODE_IMG	FMAN microcode Image
5	`fsl-image-core-ls1046ardb.ext2.gz`	VM_ROOTFS_IMG	Virtual machine's Root FS image.

9.2.5.3 LS2088A Reference Design Board (RDB)

Hardware specifications

LS2088A Reference Design Board



Figure 202. QorIQ LS2088A Architecture



Figure 203. LS208xARDB Connector Locations (Panel)

Software specifications

The LS2088ARDB and DPDK binary images are built using the Yocto Project build process. The root filesystem includes a minimal set of DPDK example and helper scripts. Refer the Yocto build process guide provided as part of the release to obtain the following images:

Table 209. List of LS2088ARDB Images from Yocto Project build process

S/No	File/Image Name	Description
1	u-boot-ls2088ardb.bin	u-boot binary
2	PBL_0x2a_0x41_1800_700_1866_1600.bin or PBL_0x2a_0x41_2000_800_2133_1600.bin(for rev. F boards only)	RCW Binary
3	mc_10.1.2_ls2088a.itb	MC Binary 10.1.x
4	dpc.0x2A_0x41.dtb	DPC Image
5	dpl-eth.0x2A_0x41.dtb	DPL Image
6	kernel-fsl-ls2088a-rdb.itb	ITB binary containing Linux kernel binary and Root file system.

9.2.6 Executing DPDK Applications on Host

This section describes how to execute DPDK and related applications in both Host and VM environments.

NOTE

IP_ADDR_BRD, IP_ADDR_IMAGE_SERVER, and TFTP_BASE_DIR are not U-Boot or Linux environment variables. They are used in this document to represent:

1. **IP_ADDR_BRD**: IP address of target board in test setup.
2. **IP_ADDR_IMAGE_SERVER**: IP address of the machine where all the software images are kept. These images are transferred to the board using either tftp or scp.
3. **TFTP_BASE_DIR**: TFTP base directory of TFTP server running on the machine where images are kept.

9.2.6.1 Flashing and booting up the target board

Follow the SDK documentation instructions to flash following images for the target board:

Common Images:

- U-boot
- RCW

DPAA specific images:

- FMAN micro code
- PPA image

DPAA2 specific images:

- Management complex Image
- DPC Image
- DPL Image

Ensure correct settings for IP_ADDR_BRD and IP_ADDR_IMAGE_SERVER, so that the board and server machine are reachable/ping-able.

9.2.6.2 DPDK examples and DPDK-based applications

This section describes the procedures once the target platform is booted up and logged into the Linux shell. This section is applicable to both DPAA and DPAA2 platforms and is organized as follows:

- *Generic Setup – DPAA* and *Generic Setup – DPAA2* contain common steps to be executed before executing any of DPDK example or DPDK-based applications. Only one of the sections must be followed depending on whether you are using a DPAA- or DPAA2-based platform.
- Application specific sections contain steps on how to execute the DPDK example and related applications.

9.2.6.2.1 Test Setup

Figure 204. Test Setup

9.2.6.2.2 Generic Setup - DPAA

This section contains steps which are common for all applications; these steps must be followed before executing any DPDK or related application on a DPAA-based platform.

DPAA-based platforms have DPAA hardware acceleration. The queues are hardware accelerated queues. They need to be configured in the DPAA-FMAN prior to being used. This can be done by choosing the appropriate policy configuration file. One can select either 1, 2 or 4 queue-based policy files. e.g. running a l3fwd application configuration may be 1 queue per port or 2 queues per port, so choose appropriate policy configuration file.

Any of the policy files below can be used while running the FMAN configure i.e. FMC script:

- `usdpaa_policy_hash_ipv4_1queue.xml`
- `usdpaa_policy_hash_ipv4_2queue.xml`
- `usdpaa_policy_hash_ipv4_4queue.xml`

All the DPDK binaries are part of the rootfs and can be found at:

```
# /usr/bin/dpdk-example
```

All the supporting DPDK XML files can be found at:

```
# /usr/bin/dpdk-example/extra
```

NOTE

Based on the policy files used, make sure that the DPDK applications runtime parameters are configured appropriately.

- *Configure the number of queues for distribution*

```
# export DPAA_NUM_RX_QUEUES=<Number of queues>
```

For example, if FMC policy file `usdpaa_policy_hash_ipv4_1queue.xml` is being used, set:

```
# export DPAA_NUM_RX_QUEUES=1
```

- *Configure FMAN (DPAA Ethernet block)*

  Run the FMC script as follows:

```
# fmc -c /usr/bin/dpdk-example/extras/usdpaa_config_ls1043.xml -p /usr/bin/dpdk-example/extras/
usdpaa_policy_hash_ipv4_1queue.xml -a
```

- *Setup huge pages*

  Using `mount | grep hugetlbfs` check to see if hugepages is already mounted. If the mount point does not exist, the command will not print any output.

```
# mkdir /mnt/hugepages
# mount -t hugetlbfs none /mnt/hugepages
```

9.2.6.2.3 Generic Setup - DPAA2

This section contains steps which are common for all applications; these steps must be followed before executing any DPDK or related application on a DPAA2-based platform.

These steps must be performed before running any of the DPDK application on host (including nxp-vhost application for running VM):

- Configure the DPAA2 resources with restool and define the resource container i.e. add the DPRC for application

```
# /usr/bin/dpdk-example/extras/dynamic_dpl.sh <DPMAC1_IDX> <DPMAC2_IDX>
# export DPRC=dprc.<DPRC_INDEX>
```

For example

```
/usr/bin/dpdk-example/extras/dynamic_dpl.sh  dpmac.1 dpmac.2

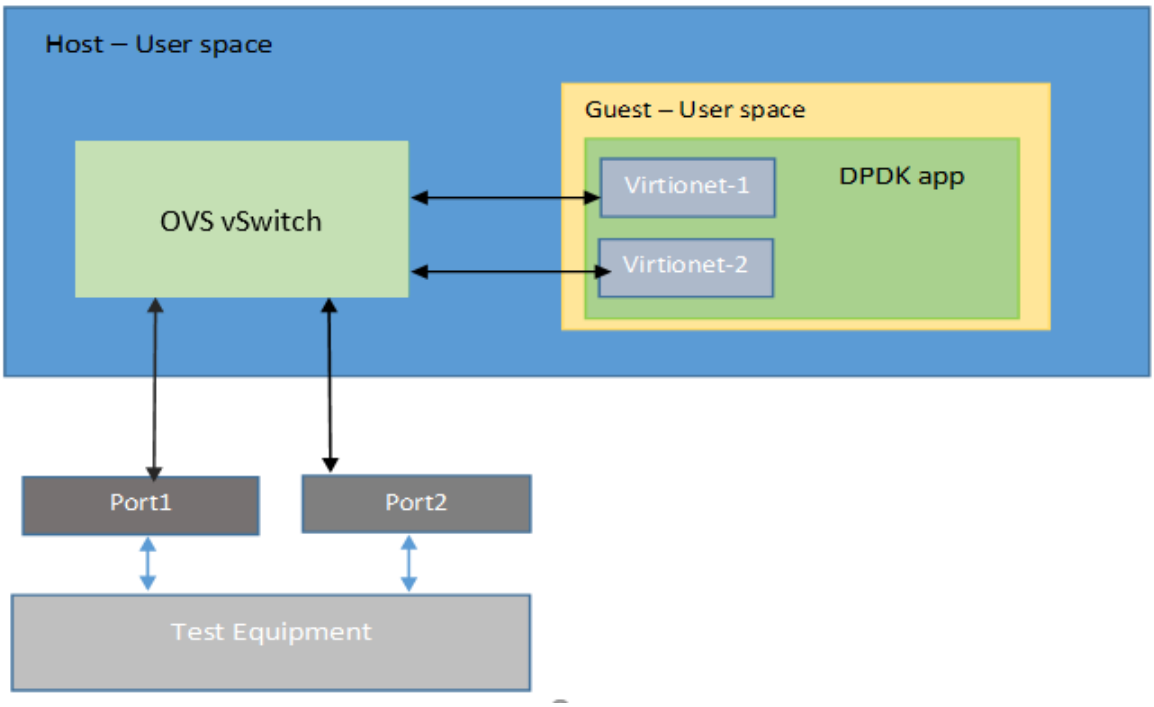
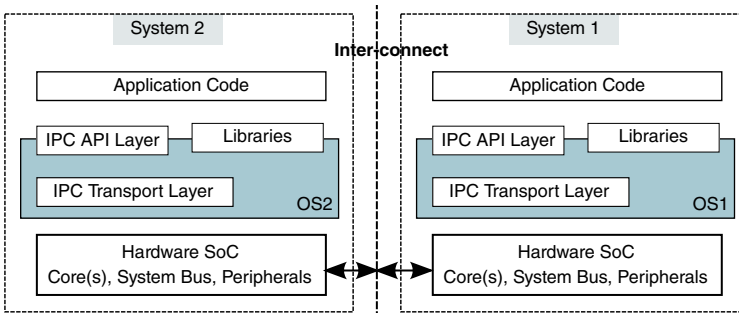
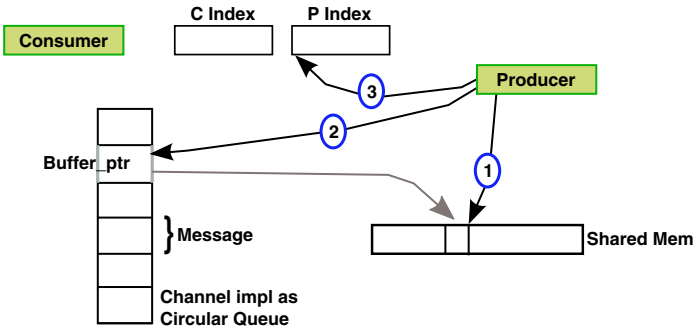
###################### Container  dprc.2  is created ####################

Container dprc.2 have following resources :=>

 * 16 DPBP
 * 5 DPCON
 * 4 DPSECI
 * 3 DPNI
 * 10 DPIO
 * 10 DPCI


######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                dpmac.3               00:00:00:00:0:3
dpni.2                dpmac.2               00:00:00:00:0:2
```

Once the steps above are performed, the environment will be set with appropriate container variable (DPRC) for DPDK Application execution. The script above also includes mounting of hugepages.

```
# export DPRC=dprc.2
```

9.2.6.2.4 DPDK example applications

L2fwd

For more information on L2fwd application, refer to L2 Forwarding Sample Application (in Real and Virtualized Environments).

```
# /usr/bin/dpdk-example/l2fwd -c 0x2 -n 1 -- -p 0x1 -q 1
```

L3fwd

For more information on L3fwd application, refer to L3 Forwarding Sample Application.

```
# /usr/bin/dpdk-example/l3fwd -c 0x6 -n 1 -- -p 0x3 --config="(0,0,1),(1,0,2)"
```

4 core - 2 Port, 2 queue per port

```
./l3fwd -c 0xF -n 4 -- -p 0x3 -P --
      config="(0,0,0),(0,1,1),(1,0,2),(1,1,3)"
```

4 core - 2 Port with dest mac

```
./l3fwd -c 0xF -n 4 -- -p 0x3 -P --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3)" --eth-
dest=0,11:11:11:11:11:11 --eth-dest=1,00:00:00:11:11:11
```

8 core - 2 Port with 4 queue per port

```
./l3fwd -c 0xFF -n 1 -- -p 0x3 --config="(0,0,0),(0,1,1),(0,2,2),(0,3,3),(1,0,4),(1,1,5),(1,2,6),
(1,3,7)"
```

L2fwd-Crypto - ARMCE

For more information on L2fwd-crypto application, refer to L2 Forwarding with Crypto Sample Application.

L2fwd-crypto Application favors physical devices with respect to virtual devices. So in order to run ARMCE, blacklist the physical devices by adding following command line parameters before ' -- ' in the commands shown below.

DPAA

```
-b 0002:10:10.0 -b 0002:10:11.0 -b 0002:10:12.0 -b 0002:10:13.0
```

DPAA2

```
-b 0000:00:00.3 -b 0000:00:01.3 -b 0000:00:02.3 -b 0000:00:03.3
```

- Cipher_only
  - 1 core:

    ```
    # ./l2fwd-crypto --vdev "cryptodev_armce_pmd"  -c 0x2 -n 1 -- -p 0x1 -q 1 -s --chain CIPHER_ONLY
    --cipher_algo AES_CBC --cipher_op ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:
    0e:0f:10
    ```

  - 2 core:

    ```
    # ./l2fwd-crypto --vdev "cryptodev_armce_pmd0" --vdev "cryptodev_armce_pmd1" -c 0x6 -n 1 -- -p
    0x3 -q 1  --chain CIPHER_ONLY  --cipher_algo AES_CBC --cipher_op ENCRYPT --cipher_key
    01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
    ```

- 4 core:

```
# /usr/bin/dpdk-example/l2fwd-crypto --vdev "cryptodev_armce_pmd0" --vdev
"cryptodev_armce_pmd1" --vdev "cryptodev_armce_pmd2" --vdev "cryptodev_armce_pmd3" -c 0xf -n 1
-- -p 0xf -q 1  --chain CIPHER_ONLY  --cipher_algo AES_CBC --cipher_op ENCRYPT --cipher_key
01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
```

- Cipher_hash
  - 1 core:

```
# /usr/bin/dpdk-example/l2fwd-crypto --vdev "cryptodev_armce_pmd" -c 0x2 -n 1 -- -p 0x1 -q 1 --
chain CIPHER_HASH  --cipher_algo AES_CBC --cipher_op ENCRYPT --cipher_key
01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --auth_algo SHA1_HMAC --auth_op GENERATE
```

NOTE

For multi core and multi port, add vdev and modify portmask (-p) and coremask (-c) options as described in cipher only use case.

- Hash_cipher
  - 1 core:

```
# /usr/bin/dpdk-example/l2fwd-crypto --vdev "cryptodev_armce_pmd" -c 0x2 -n 1 -- -p 0x1 -q 1 -s
--chain HASH_CIPHER --auth_algo SHA1_HMAC --auth_op GENERATE --cipher_algo AES_CBC --cipher_op
ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
```

NOTE

For multi core and multi port, add vdev and modify portmask (-p) and coremask (-c) options as described in cipher only use case.

- Hash_only
  - 1 core:

```
# /usr/bin/dpdk-example/l2fwd-crypto --vdev "cryptodev_armce_pmd" -c 0x2 -n 1 -- -p 0x1 -q 1 --
chain HASH_ONLY --auth_algo SHA1_HMAC --auth_op GENERATE
```

NOTE

For multi core and multi-port, add vdev and modify portmask (-p) and coremask (-c) options as described in cipher only use case.

L2fwd-Crypto – DPAA-SEC HW or DPAA2-SEC-HW

- Cipher only

```
# /usr/bin/dpdk-example/l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1  --chain CIPHER_ONLY  --
cipher_algo AES_CBC --cipher_op ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:
0f:10
```

- Cipher - hash

```
# /usr/bin/dpdk-example/l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1 --chain CIPHER_HASH  --
cipher_algo AES_CBC --cipher_op ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:
0f:10 --auth_algo SHA1_HMAC --auth_op GENERATE
```

- Hash only

```
# /usr/bin/dpdk-example/l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_ONLY --auth_algo
SHA1_HMAC --auth_op GENERATE
```

NOTE

For multi core and multi-port, modify portmask (-p) and coremask (-c) options as described in ARMCE cipher only use case.

Ipsec-secgw - DPAA-SEC-HW or DPAA2-SEC-HW

For IPsec application, two boards need to be configured as endpoint 0 (ep0) and endpoint 1 (ep1). Port 0 and port 1 of ep0 should be connected to port 0 and port 1 of ep1 respectively. Port 2 and port 3 of ep0 should be connected to packet generator tool (eg. Spirent) with destination IP addresses as 192.168.105.XXX and 192.168.106.XXX respectively. Port 2 and port 3 of ep1 should be connected to packet generator tool (eg. Spirent) with destination IP addresses as 192.168.115.XXX and 192.168.116.XXX respectively. For more information, refer to IPsec Security Gateway Sample Application .

Endpoint 0 configuration

```
./ipsec-secgw -c 0xf  -- -p 0xf -P -u 0x3 --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" --ep0
```

Endpoint 1 configuration

```
./ipsec-secgw -c 0xf  -- -p 0xf -P -u 0x3 --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" --ep1
```

ARM-CE based Ipsec-secgw

For IPsec application, two boards need to be configured as endpoint 0 (ep0) and endpoint 1 (ep1). Port 0 and port 1 of ep0 should be connected to port 0 and port 1 of ep1 respectively. Port 2 and port 3 of ep0 should be connected to packet generator tool (eg. Spirent) with destination IP addresses as 192.168.105.XXX and 192.168.106.XXX respectively. Port 2 and port 3 of ep1 should be connected to packet generator tool (eg. Spirent) with destination IP addresses as 192.168.115.XXX and 192.168.116.XXX respectively. For more information, refer to IPsec Security Gateway Sample Application.

DPAA

Endpoint 0 configuration

```
./ipsec-secgw -c 0xf --vdev "cryptodev_armce_pmd0" --vdev "cryptodev_armce_pmd1" --vdev
"cryptodev_armce_pmd2" --vdev "cryptodev_armce_pmd3" -b 0002:10:10.0 -b 0002:10:11.0 -b
0002:10:12.0 -b 0002:10:13.0 -- -p 0xf -P -u 0x3 --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" --ep0
```

Endpoint 1 configuration

```
./ipsec-secgw -c 0xf --vdev "cryptodev_armce_pmd0" --vdev "cryptodev_armce_pmd1" --vdev
"cryptodev_armce_pmd2" --vdev "cryptodev_armce_pmd3" -b 0002:10:10.0 -b 0002:10:11.0 -b
0002:10:12.0 -b 0002:10:13.0 -- -p 0xf -P -u 0x3 --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" --ep1
```

DPAA2

Endpoint 0 configuration

```
./ipsec-secgw -c 0xf --vdev "cryptodev_armce_pmd0" --vdev "cryptodev_armce_pmd1" --vdev
"cryptodev_armce_pmd2" --vdev "cryptodev_armce_pmd3" -b 0000:00:00.3 -b 0000:00:01.3 -b
0000:00:02.3 -b 0000:00:03.3 -- -p 0xf -P -u 0x3 --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" --ep0
```

Endpoint 1 configuration

```
./ipsec-secgw -c 0xf --vdev "cryptodev_armce_pmd0" --vdev "cryptodev_armce_pmd1" --vdev
"cryptodev_armce_pmd2" --vdev "cryptodev_armce_pmd3" -b 0000:00:00.3 -b 0000:00:01.3 -b
0000:00:02.3 -b 0000:00:03.3 -- -p 0xf -P -u 0x3 --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" --ep1
```

Pktgen

#3 port - 1 core each

```
# ./pktgen -l 0-3 -n 3 --proc-type auto --file-prefix pg --log-level 8 -- -T -P -m "[1].0, [2].1,
[3].2"
```

#2 port - 2 core each

```
# ./pktgen -l 0-3 -n 3 --proc-type auto --file-prefix pg --log-level 8 -- -T -P -m "[0:1].0, [2:3].
1"
```

#To start traffic on specific port:

```
start 0
stop 0
```

#To start on all ports

```
str
stp
```

OVS

Run OVS switch with two DPDK ports (physical ports) added to ovs switch and hard coded flows programed as:

- Incoming traffic Port 1 -> output to port 2
- Incoming traffic Port2 -> output to port 1

Mount huge pages for OVS:

```
#mkdir -p /dev/hugepages
#mount -t hugetlbfs -o pagesize=1G none /dev/hugepages
#mkdir -p /usr/local/etc/openvswitch
#mkdir -p /usr/local/var/run/openvswitch
#rm /usr/local/etc/openvswitch/conf.db
#export SOCK_MEM=1024
```

Setup and execute ovs-vswitchd, and configure flows.

Create and configure OVS data base:

```
#cd /usr/bin/ovs-dpdk/
#./ovsdb-tool create /usr/local/etc/openvswitch/conf.db ./vswitch.ovsschema
#./ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock --
remote=db:Open_vSwitch,Open_vSwitch,manager_options --pidfile=/tmp/ovsdb-server.pid --detach --log-
file=/var/log/openvswitch/ovs-vswitchd.log
```

Initialize OVS switch configuration:

```
#./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
#./ovs-vsctl  --no-wait set Open_vSwitch . other_config:pmd-cpu-mask=6
#./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="$SOCK_MEM"
```

Launch ovs vswitchd daemon:

```
#export DB_SOCK=/usr/local/var/run/openvswitch/db.sock
#./ovs-vswitchd unix:$DB_SOCK --pidfile --detach -c 0x6
```

NOTE

—detach option makes the daemon run in background. If this option is given same shell can be used to run further commands, otherwise ssh to the target board and run further commands.

Add bridge interface and DPDK ports to vswitch:

```
#./ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
#./ovs-vsctl add-port br0 dpdk0 -- set Interface dpdk0 type=dpdk
#./ovs-vsctl add-port br0 dpdk1 -- set Interface dpdk1 type=dpdk
```

Configure Flows to set data path across the ports:

```
#./ovs-ofctl del-flows br0
#./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=1,actions=output:2
#./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=2,actions=output:1
```

The flows above configure a hard coded bi-directional data path between port 1 and port 2.

9.2.7 OVS-DPDK and DPDK in VM with VIRTIO Interfaces

DPDK example and DPDK-based applications can also run inside the virtual machine. This section describes steps to run these applications inside the virtual machine on both DPAA and DPAA2 platforms.

The virtual machine runs inside the host Linux system and is launched by an application called QEMU.

Note that console logs for the guest Linux do not appear on the host Linux console (i.e UART). The guest logs are exposed through telnet, and they can be accessed by doing telnet on the host board's IP Address (IP_ADDR_BRD) and GUEST_CONSOLE_TELNET_PORT. Each Virtual machine that is run on a single host is allocated a different GUEST_CONSOLE_TELNET_PORT, and this port number is specified by user running virtual machine through the QEMU command line.

This chapter is organized as follows:

- *Generic steps – DPAA & DPAA2 platforms* describes generic or common steps to be executed on DPAA and DPAA2 platforms in order to run virtual machine.
- *OVS-switch as VHOST USER backend* describes OVS switch execution. OVS acts as backend for VHOST user devices. If DPDK applications are run in VM using VIRTIO and VHOST USER then this section has to be followed.
- *Launch virtual machine and access console* describes launching QEMU and accessing virtual machines.
- *Running DPDK applications in VM* describes Running of DPDK and DPDK based applications in virtual machine.

9.2.7.1 Generic steps - DPAA & DPAA2 platforms

Mount huge tlbfs so that QEMU can use huge pages. Also give IP address to board so that virtual machine console can be accessed using telnet.

```
#echo hugetlbfs /hugetlbfs hugetlbfs defaults,mode=0777 0 0 >> /etc/fstab
#mkdir /hugetlbfs
```

```
#mount /hugetlbfs
#ifconfig eth<x> <IP_ADDR_BRD>
```

Follow steps given in *Generic Setup – DPAA* or *Generic Setup – DPAA2* to setup DPDK for your platform. Note that only one of these sections must be followed depending on whether you are usin a DPAA or DPAA2-based platform.

9.2.7.2 OVS-switch as VHOST USER backend

OVS is used as backend for VHOST USER ports. The physical ports on the target platform and the vhost user ports (virtio devices) are added to ovs-vswitch and hard-coded flows are created to establish traffic switching between physical ports and vhost devices as follows:

Incoming traffic Physical port1 -> output to vhost-user port 1

Incoming traffic on vhost-user port1 -> output on physical port 1

Incoming traffic on physical port 2 -> output on vhost-user port 2

Incoming traffic on vhost-user port 2 -> output on physical port 2

The following steps must be followed to setup OVS as vhost switching backend:

Mount huge pages and create OVS directories:

```
#mkdir -p /dev/hugepages
#mount -t hugetlbfs -o pagesize=1G none /dev/hugepages
#mkdir -p /usr/local/etc/openvswitch
#mkdir -p /usr/local/var/run/openvswitch
#rm /usr/local/etc/openvswitch/conf.db
```

Export socket memory:

```
#export SOCK_MEM=1024
```

NOTE

For LS208X platforms use SOCK_MEM=1024, for LS104X platforms use SOCK_MEM=200

Initialize OVS DB:

```
#mkdir -p /usr/local/etc/openvswitch
#mkdir -p /usr/local/var/run/openvswitch
#./ovsdb-tool create /usr/local/etc/openvswitch/conf.db ./vswitch.ovsschema
```

Start the OVS Server:

```
#./ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock --
remote=db:Open_vSwitch,Open_vSwitch,manager_options --pidfile=/tmp/ovsdb-server.pid --detach --log-
file=/var/log/openvswitch/ovs-vswitchd.log
```

Initialize OVS:

```
#./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
#./ovs-vsctl  --no-wait set Open_vSwitch . other_config:dpdk-lcore-mask=3
#./ovs-vsctl  --no-wait set Open_vSwitch . other_config:pmd-cpu-mask=3
#./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="$SOCK_MEM"
```

Start the vSwitch daemon:

```
#export DB_SOCK=/usr/local/var/run/openvswitch/db.sock
#./ovs-vswitchd unix:$DB_SOCK --pidfile --detach -c 0x3
```

Initialize the vSwitch bridge and define DPDK ports:

```
#./ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
#./ovs-vsctl add-port br0 dpdk0 -- set Interface dpdk0 type=dpdk
#./ovs-vsctl add-port br0 dpdk1 -- set Interface dpdk1 type=dpdk

#./ovs-vsctl add-port br0 vhost-user1 -- set Interface vhost-user1 type=dpdkvhostuser
#./ovs-vsctl add-port br0 vhost-user2 -- set Interface vhost-user2 type=dpdkvhostuser
```

Clear flows and add new flows for switching packets arriving on port 1 to port 2 and reverse. In case a different combination of ports and switching is required, modify accordingly:

```
#./ovs-ofctl del-flows br0
#./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=1,actions=output:3
#./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=3,actions=output:1
#./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=2,actions=output:4
#./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=4,actions=output:2
```

NOTE

OVS Switch must be run before launching the virtual machine using QEMU, otherwise the virtual machine launch will fail.

9.2.7.3 Launch virtual machine

```
#export ROOTFS_IMG=[VM_ROOTFS_IMG]
#export VM_MEM=650M
# export CORES=2
# export NUM_QUEUES=1
# export VHOST1_PATH=/usr/local/var/run/openvswitch/vhost-user1
# export VHOST2_PATH=/usr/local/var/run/openvswitch/vhost-user2
#qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/
hugetlbfs,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm -initrd $ROOTFS_IMG
-smp 2 -serial tcp::4446,server,telnet -append 'root=/dev/ram0 rw console=ttyAMA0,115200 rootwait
earlyprintk ramdisk_size=1000000' -m $VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=
$VHOST1_PATH -netdev type=vhost-user,id=hostnet1,chardev=char1 -device virtio-net-pci,disable-
modern=false,addr=0x3,netdev=hostnet1,id=net1 -chardev socket,id=char2,path=$VHOST2_PATH -netdev
type=vhost-user,id=hostnet2,chardev=char2 -device virtio-net-pci,disable-
modern=false,addr=0x4,netdev=hostnet2,id=net2 -smp $CORES -S
```

NOTE

* `GUEST_CONSOLE_TELNET_PORT` is given as 4446 in the example above. You need to telnet to `IP_ADDR_BRD` and `GUEST_CONSOLE_TELNET_PORT` to access guest terminal

* `VM_ROOTFS_IMG` has to be replaced with correct rootfs image name from *Supported platforms and platform specific details*.

* For LS104xA platforms, set `VM_MEM` to 650M. For L208xA platforms, set `VM_MEM` to 2048M.

* Set ROOTFS_IMG name as "fsl-image-core-lsxxxxardb.ext2.gz" as per LS1 and LS2 platform.

The following logs appear on Host UART console:

```
# /usr/bin/qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=800M,mem-path=/
hugetlbfs,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm -initrd $
{ROOTFS_IMG} -smp 4 -serial tcp::4446,server,telnet -append 'root=/dev/ram0 rw
console=ttyAMA0,115200 rootwait earlyprintk ramdisk_size=1000000' -m 800M -numa node,memdev=mem -
chardev socket,id=char1,path=/home/root/sock1 -netdev type=vhost-user,id=hostnet1,chardev=char1 -
device virtio-net-pci,disable-modern=false,addr=0x3,netdev=hostnet1,id=net1 -chardev
socket,id=char2,path=/home/root/sock2 -netdev type=vhost-user,id=hostnet2,chardev=char2 -device
virtio-net-pci,disable-modern=false,addr=0x4,netdev=hostnet2,id=net2 -
qemu-system-aarch64: -netdev type=vhost-user,id=hostnet1,chardev=char1: chardev "char1" went up
qemu-system-aarch64: -netdev type=vhost-user,id=hostnet2,chardev=char2: chardev "char2" went up
QEMU 2.6.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server
VHOST_DATA: (0) Failed to add device MAC address to VMDQ
(qemu)
```

NOTE

QEMU logs come completely only when you log-in to the guest machine using telnet as specified in section *Accessing virtual machine console*.

The `-S` option mentioned in the qemu command stops the Virtual Machine bootup after initial setup. Run the `info cpus` command on QEMU CLI interface to see the QEMU threads.

```
(qemu) info cpus
* CPU #0: thread_id=2559
  CPU #1: (halted) thread_id=2560
```

SSH on the board (telenet to `IP address IP_ADDR_BRD`) from other console and affine the threads to the cores using the `taskset` command:

```
taskset -p 0x4 <tid1>
taskset -p 0x8 <tid1>
```

NOTE

It is better to affine the VCPUs to the cores on which OVS threads are not running. For better performance VCPU threads should be given one physical CPU each if possible.

Run the `c` command from the QEMU CLI to continue the VM boot-up:

```
(qemu) c
```

9.2.7.4 Accessing virtual machine console

Telnet to the `IP_ADDR_BRD` at port `GUEST_CONSOLE_PORT` from any machine, which can reach `IP_ADDR_BRD` over network:

```
$ telnet 192.168.3.155 4446
Trying 192.168.3.155...
Connected to 192.168.3.155.
Escape character is '^]'.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 4.1.8-rt8+gf5a59f7 (b27504@b27504-OptiPlex-790) (gcc version 4.9.3
20150311 (prerelease) (Linaro GCC 4.9-2015.03) ) #1 SMP Mon May 9 18:50:49 IST 2016
[    0.000000] CPU: AArch64 Processor [410fd034] revision 4
[    0.000000] Detected VIPT I-cache on CPU0
[    0.000000] alternatives: enabling workaround for ARM erratum 845719
```

```
[    0.000000] efi: Getting EFI parameters from FDT:
[    0.000000] efi: UEFI not found.
```

NOTE

The complete Linux log on the terminal is not shown above.

9.2.7.5 Launching two virtual machines

This section is optional. It is only required if two virtual machines need to be launched for a use case.

In case of two virtual machines, please take care of following:

- Memory assigned to each virtual machine should not exceed the total number of huge pages assigned on system. In above example (step 8) giving 2048 Mega bytes to each virtual machine works fine.
- Console telnet port of both virtual machine must be different, for example 4446 could be console of VM1 and 4447 could be console for VM2.

Following are the example launch commands for two VMs:

VM1:

```
# qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/
hugetlbfs,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm -initrd /boot/$
{ROOTFS_IMG}  -serial tcp::4446,server,telnet -append 'root=/dev/ram0 rw console=ttyAMA0,115200
rootwait earlyprintk ramdisk_size=1000000' -m $VM_MEM -numa node,memdev=mem -chardev
socket,id=char1,path=$VHOST1_PATH -netdev type=vhost-user,id=hostnet1,chardev=char1 -device virtio-
net-pci,disable-modern=false,addr=0x3,netdev=hostnet1,id=net1 -chardev socket,id=char2,path=
$VHOST2_PATH -netdev type=vhost-user,id=hostnet2,chardev=char2 -device virtio-net-pci,disable-
modern=false,addr=0x4,netdev=hostnet2,id=net2 -smp $CORES -S
```

VM2:

```
# qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/
hugetlbfs,share=on -cpu host -machine type=virt -kernel /boot//Image -enable-kvm -initrd /boot/$
{ROOTFS_IMG}  -serial tcp::4447,server,telnet -append 'root=/dev/ram0 rw console=ttyAMA0,115200
rootwait earlyprintk ramdisk_size=1000000' -m $VM_MEM -numa node,memdev=mem -chardev
socket,id=char1,path=$VHOST1_PATH -netdev type=vhost-user,id=hostnet1,chardev=char1 -device virtio-
net-pci,disable-modern=false,addr=0x3,netdev=hostnet1,id=net1 -chardev socket,id=char2,path=
$VHOST2_PATH -netdev type=vhost-user,id=hostnet2,chardev=char2 -device virtio-net-pci,disable-
modern=false,addr=0x4,netdev=hostnet2,id=net2 -smp $CORES -S
```

9.2.7.6 Running DPDK applications in VM

All the DPDK applications mentioned in this section have been tested in following configuration:

- Two Physical network interfaces.
- Two virtio-net devices in the virtual machine.

Following figure illustrates the test setup:



Figure 205. DPDK virtionet test setup

Generic Setup (for all the applications)

All the DPDK binaries and other supporting XML files are part of the rootfs and can be found at:

```
#/usr/bin/dpdk-example/
```

Setup huge pages

```
# mkdir /mnt/hugepages
# mount -t hugetlbfs none /mnt/hugepages
# echo 512 > /proc/sys/vm/nr_hugepages
```

Bind the virtio-net devices

```
# /usr/share/tools/dpdk_nic_bind.py --status
# /usr/share/tools/dpdk_nic_bind.py -b uio_pci_generic 0000:00:03.0
# /usr/share/tools/dpdk_nic_bind.py -b uio_pci_generic 0000:00:04.0
```

L2fwd

```
# /usr/bin/dpdk-example/l2fwd -c 0xf -n 1 -- -p 0x3 -q 1
```

L3fwd

```
# /usr/bin/dpdk-example/l3fwd -c 0x1 -n 1 -- -p 0x1 --config="(0,0,0)" --parse-ptype
```

Testpmd

For TX only:

```
#/usr/bin/dpdk-example/testpmd -c 1 -n 1 -- -i --nb-cores=1 --nb-ports=1 --total-num-mbufs=1025 --
forward-mode=txonly --disable-hw-vlan --port-topology=chained
```

For RX only:

```
# /usr/bin/dpdk-example/testpmd -c 1 -n 1 -- -i --nb-cores=1 --nb-ports=1 --total-num-mbufs=1025 --
forward-mode=rxonly --disable-hw-vlan --port-topology=chained
```

9.2.7.7 Multi Queue VIRTIO support

To scale the performance vs number of VM cores, the VIRTIO devices need to be configured with multiple queues. This section explains the steps required for setup and usage of multi queue virtio devices.

Adding multiple queues to physical devices and host-user devices is control by parameters in sections *Generic Setup – DPAA2* and *OVS-switch as VHOST USER backend*. These parameter additions/values are given in sections *Multi queue parameters in Common steps for DPAA2* and *Additional steps for OVS setup*.

QEMU commands for multiqueue vhost devices are different and are shown later in the section.

Multi queue parameters in Common steps for DPAA2

For using multiple queues on physical devices on DPAA2, `MAX_TCS` and `MAX_DIST_PER_TC` has to be set to 8 in generic steps for DPAA2 as given in *Generic Setup – DPAA2*

```
# export DPRC=dprc.2
# export MAX_TCS=8
# export MAX_DIST_PER_TC=8,8,8,8,8,8,8,8
# /usr/bin/dpdk-example/extras/dynamic_dpl.sh <DPMAC 1> <DPMAC 2>
```

Multi queue parameters in Common steps for DPAA1

For using multiple queues on physical devices on DPAA1, DPAA_NUM_RX_QUEUES has to be set to '2' or '4' in generic steps for DPAA1 as given in *Generic Setup – DPAA. Also, fmc should be run accordingly. Considering 2 queue scenario:*

```
# export DPAA_NUM_RX_QUEUES=2
# fmc -c /usr/bin/dpdk-example/extras/usdpaa_config_ls1043.xml -p /usr/bin/dpdk-example/extras/
usdpaa_policy_hash_ipv4_2queue.xml -a
```

Additional steps for OVS setup

Set number of queues for all the devices while configuring devices. Underlined steps configure the number of queues for all the devices (physical and vhost) added to OVS. All other steps (not underline) remain the same as given in section *OVS-switch as VHOST USER backend* (all other steps remains same as given in the section)

Run following commands after adding DPDKx and vhost-user ports to the bridge:

```
./ovs-vsctl set Interface dpdk0 options:n_rxq=2
./ovs-vsctl set Interface dpdk1 options:n_rxq=2
./ovs-vsctl set Interface dpdk0 options:n_txq=2
./ovs-vsctl set Interface dpdk1 options:n_txq=2
./ovs-vsctl set Interface vhost-user1 options:n_rxq=2
./ovs-vsctl set Interface vhost-user2 options:n_rxq=2
./ovs-vsctl set Interface vhost-user1 options:n_txq=2
./ovs-vsctl set Interface vhost-user2 options:n_txq=2
```

Launch VM with multi queue VHOST devices

Underlined text is the addition to the QEMU command and parameters given in section Launch virtual machine

```
export VM_MEM=2048M (For DPAA1 use VM_MEM=650M)
export CORES=2
export NUM_QUEUES=2
ROOTFS_IMG=<VM_ROOTFS_IMG>
SERIAL_PORT=4446
export VHOST1_PATH=/usr/local/var/run/openvswitch/vhost-user1
export VHOST2_PATH=/usr/local/var/run/openvswitch/vhost-user2
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/
hugetlbfs,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm -initrd
$ROOTFS_IMG  -serial tcp::$SERIAL_PORT,server,telnet -append 'root=/dev/ram0 rw
console=ttyAMA0,115200 rootwait earlyprintk ramdisk_size=1000000' -m $VM_MEM -numa node,memdev=mem
-chardev socket,id=char1,path=$VHOST1_PATH -netdev type=vhost-
user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
modern=false,addr=0x3,netdev=hostnet1,mq=on,id=net1,vectors=6 -chardev socket,id=char2,path=
$VHOST2_PATH -netdev type=vhost-user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES -
device virtio-net-pci,disable-modern=false,addr=0x4,netdev=hostnet2,mq=on,id=net2,vectors=6 -smp
$CORES
```

DPDK applications in VM

Connect to VM terminal as explained in *Accessing Virtual Machine Console*. Once logged-in as Guest, DPDK applications using multiple queues can be run in VM.

NOTE

DPDK application in VM must use the same number of queues as used in VM setup in section *Launch VM with multiqueue VHOST devices* (`NUM_QUEUES`). If the number of queues is not same then virtio device will fail to start and it will not work in VM.

Common steps

```
mkdir /mnt/hugepages
mount -t hugetlbfs none /mnt/hugepages
echo 512 > /proc/sys/vm/nr_hugepages
/usr/share/tools/dpdk_nic_bind.py -b uio_pci_generic 0000:00:03.0
/usr/share/tools/dpdk_nic_bind.py -b uio_pci_generic 0000:00:04.0
```

L3fwd

```
/usr/bin/dpdk-example/l3fwd -c 0x3 -n 1 -- -p 0x3 --config="(0,0,0),(0,1,1), (1,0,0), (1, 1, 1)" --
parse-ptype
```

Testpmd

```
/usr/bin/dpdk-example/testpmd -c 3 -n 1 -- -i --nb-cores=1 --nb-ports=1 --total-num-mbufs=1025 --
forward-mode=txonly --disable-hw-vlan --rxq=2 --txq=2 --port-topology=chained
```

9.2.8 Known Limitations and Future Work

1. DPAA: Ports assigned to user space cannot be assigned dynamically to kernel space or vice-versa.
2. DPAA: The FMAN configuration script cannot be re-run without system restart. i.e. number of queues configured cannot be changed.
3. DPAA: Ipsec-secgw: Traffic halts with error messages when running medium to large size packets on all 4 ports.

4. DPAA: DPDK-204 Performance of DPAA1 application degrades on assigning some ports to linux.
5. DPAA2: Performance drop is observed in case of Host and VM because of use of Physical addressing in default code. To switch to virtual addressing, use CONFIG_RTE_LIBRTE_DPAA2_USE_PHYS_IOVA=n option while compiling the source code.
6. DPAA and DPAA2: Some of the DPDK Ethernet ops are not yet implemented or are empty e.g Multicast fitlering.
7. DPAA and DPAA2: The hardware internally uses the hardware access portals for each thread doing packet I/O, which limits the number of I/O threads thereby impacting the performance. The number of available portals is different for DPAA and DPAA2.

9.2.9 Troubleshooting

Following are some common steps and suggestions outlined for best performance from DPDK Applications:

1. To obtain best performance, please ensure that the boot-up time command line arguments are similar to below:

```
console=ttyS1,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0600 ramdisk_size=2000000
default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7
```

   `isolcpus` in the above ensures that only Linux Kernel schedules its threads on Core 0 only. Core 1-7 would be used for DPDK application threads.

   Hugepage count defined by `hugepages` should also be modified to maximum possible so as to allow DPDK applications to have larger buffers.

2. If RX or TX is not happening, verify the following points:
   a. Ensure that no error has been reported by DPDK application at startup. Generally the output is descriptive enough for cause of problem.
   b. Ensure that correct `dpni.X` has been used in the `dynamic_dpl.sh` script while creating the `dprc` containers. A common pitfall is to use an incorrect `dpni` as against the physical port being used for IO.
   c. Ensure that traffic generator to board connectivity is proper. You may run testpmd in tx_only mode to validate if the packets are going out on specific interfaces.
   d. Ensure that the traffic generator Stream settings are correct and enough streams are being generated for proper distribution between DPDK application cores.
   e. Ensure that the MAC address of stream generated by traffic generator matches that of the `dpni` port, or the interface is in promiscous mode.
3. If the performance is not as expected:
   a. Ensure that the stream configuration of the traffic generator is approrpriate and that it can generate multiple streams. In case the streams have all same IP destination and/or source, the distribution of traffic across multiple cores wouldn't happen.
   b. Using standard process tools in Linux, for example `ps`, `top`, verify that all the DPDK application threads have been started (as per application configuration on command line) and busy looping.

9.3 IPC

9.3.1 IPC Specification

9.3.1.1 Introduction

Scope

The scope of the IPC (inter-processor communication) software module is to provide a framework for communication between two heterogeneous systems. These systems can be connected through either of Shared Memory or other interconnect mechanism like Queue Manager.

Examples of heterogeneous systems:

- 8156-P2020 SoC on a combo card
- DSP-PA in 9131 SoC running an LTE solution
- DSP1-PA and DSP2-PA in 9132 SoC running LTE and WCDMA together
- DSP-PA on B4860

A Heterogeneous system is a logical entity which defines an application use case.

IPC, in this document refers to communication between s/w running on PA core and DSP core across heterogeneous systems. For intra PA and intra DSP (Homogeneous) communication existing OS supported communication mechanisms should be used.

The scope of IPC is limited to providing a communication framework which may not conform to any existing standard like MCAPI. SoC which would be supported by the IPC framework are B4860 and B4420.

Audience

- System architects
- Linux and SmartDSP OS developers, and
- LTE L1 L2 protocol developers

9.3.1.2 General Description

This section describes the general factors that affect the product and its requirements. It provides background for specifications that will be defined in detail in the section Specifications below.

Product Perspective

In a typical use case of communication between two heterogeneous systems, a communication framework layer is required for exchanging data between the two. The heterogeneous systems may have different operating systems and processor architecture. A block diagram of such an organization is:



Figure 206. Communication Framework

In the above diagram, two systems are connected using an interconnect. The systems have their OS, own cores, system bus and other set of peripherals. The applications which require transferring data to/from other system take the services of IPC layer, which consists of 2 parts API layer and transport layer.

The API layer provides an interface for applications, while the transport layer manages interconnect. When moving from one interconnects to another, this is the only layer that requires change.

Use Cases

- LTE L1-L2 communication
- Shared Devices Initialization
- L1 and L2 framework synchronization as required for a fully synchronized system startup procedure
- Multi RAT (LTE and WCDMA or 2 instances of LTE or WCDMA)
- L1 –L1 communication depends upon the Application

9.3.1.3 Definitions

Table 210. Definitions and Terminology

Term	Definition	Notes
Producer	A ‘producer’ is defined as a processing entity which creates a message to be sent over to ‘consumer’ using the communication framework.	They are also referred as ‘endpoints’
Consumer	A ‘consumer’ is a processing entity which consumes the message.	A consumer polls for reception of a message.
Message	Message’ is defined as a sequence of bytes. No limitation is set on the endianness/ data format.	A message is sent by producer to a consumer on a specified channel.

Table continues on the next page...

Table 210. Definitions and Terminology (continued)

Term	Definition	Notes
Channel	1. Acts as a logical medium of communication between producer and consumer. 2. Is unidirectional. a. If a producer interacts with two consumers a separate channel is required for each consumer. b. If a consumer interacts with two producers, separate channel is required for each producer-consumer pair. 3. The channel storage is modelled as a circular queue (BD ring). The transport layer may allocate the memory for storage, or the producer/consumer allocates the memory for the message. 4. Has a depth which is fixed at the time of initialization. 5. A channel has 2 indices a producer index and consumer index. This is used to maintain lock free access to the BD ring. 6. A channel's notifications can be VIRQ on DSP side whereas it is only polling mode at PA side. 7. There are essentially 2 types of channels: a. Pointer Channel • Producer-Consumer exchange pointers rather than actual messages. b. Message Channel • Actual Message is exchanged.	see the figure below.
Channel-ID	A 32bit static value which is predefined and used as a way to identify the consumer and its channels.	
Notification	The method of notifying the consumer that a message has been added in the BD ring. The consumer provides the details of how the notification has to be generated.	This information is filled in the channel structure at the time of initialization
IPC Instance	An object of ipc data structure. An ipc object contains set of channel structures.	Each ipc instance object would point to its own set of channel structures.



Figure 207. Consumer/Producer Dynamic Model

9.3.1.4 IPC Architecture

IPC Framework Details

Table 211. IPC Detail Described

Detail	Description
A	The IPC framework provides the basic building blocks for the communication between the producer and the consumer. While there are different types of channels from the applications view, internally all are derived from the pointer channel.
B	IPC framework does not limit the number of channels. The default values is 64, any value less than or equal to 0 will be set to 64.
C	IPC framework can have instances and applications can communicate with each other based on an instance ID. This would help in providing isolation between LTE and WCDMA L2 stacks using IPC. For example: • LTE L1 (DSP) and LTE L2 (PA) would communicate on ipc instance 1 • WCDMA L1 (DSP) and WCDMA L2 (PA) would communicate on ipc instance 2

IPC Layer Initialization

As the actual data needs to be in shared memory, IPC requires the calling application to provide the physical to virtual address conversion callback (p2vcb). The p2v function should be passed as an argument to the init function

```
IPC Initialization APIs
-----------------------

Old API
-------
fsl_ipc_t fsl_ipc_init(ipc_p2v_t p2vcb,
                       mem_range_t sh_ctrl_area,
                       mem_range_t dsp_ccsr,
                       mem_range_t pa_ccsr,
                       char uiodevbuf[]);


NOTE: This API is now deprecated. It now internally calls following API "fsl_ipc_init_rat" with a
new argument rat_id.

New API
-------
```

```
fsl_ipc_t fsl_ipc_init_rat(uint32_t rat_id,
                           ipc_p2v_t p2vcb,
                           mem_range_t sh_ctrl_area,
                           mem_range_t dsp_ccsr,
                           mem_range_t pa_ccsr,
                           char uiodevbuf[]);

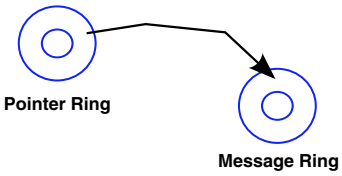
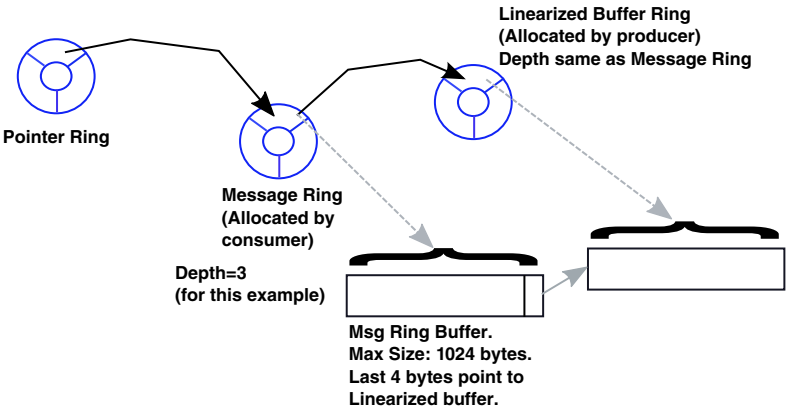
Whereas:
       rat_id - is the multi-rat instance it can be 0/1
       uiodevbuf[] - is the uio interface corresponding to the DMA channels
```

The above API internally binds to the specific ipc_t object created in the shared control area. Any subsequent ipc calls will try to get information from that ipc_t object.

IPC Channel Initialization

(d) There are separate producer and consumer API's for opening a channel, unless the consumer has opened the channel producer should not start the communication.

The APIs are

```
Linux Producer API
------------------
int fsl_ipc_open_prod_ch(uint32_t channel_id,
                         fsl_ipc_t ipc);

Linux Consumer API
------------------
int fsl_ipc_configure_channel(uint32_t channel_id,
                              uint32_t depth,
                              ipc_ch_type_t channel_type,
                              unsigned long msg_ring_paddr,
                              uint32_t msg_size,
                              ipc_cbfunc_t cbfunc,
                              fsl_ipc_t ipc)

*For more details on API refer Appendix
```

Generic IPC Layer

Pointer Ring

The generic IPC layer provides a method of transferring a pointer value on a channel. Along with the pointer the length of the message it points is also stored in the bd ring.

```
Linux User Producer API
-----------------------
int fsl_ipc_send_ptr(uint32_t channel_id,
                     unsigned long buffer_ptr,
                     uint32_t len,
                     fsl_ipc_t ipc)


Linux User Consumer API
-----------------------
int fsl_ipc_recv_ptr(uint32_t channel_id,
                     unsigned long *addr,
                     uint32_t *len,
```

```
                     fsl_ipc_t ipc);

int fsl_ipc_recv_ptr_hold(uint32_t channel_id,
                          unsigned long *addr,
                          uint32_t *len,
                          fsl_ipc_t ipc);

Note: the "hold" API does not increment the consumer index. Calling fsl_ipc_set_consumed_status on
the channelid increments the consumer index
```

Message Ring

In cases where a requirement of sending small or medium messages of size <1024 bytes, IPC layer provides another ring called a message ring.

Applications which does not want to incur the overhead of DMA and if the message size is small to be copied by core , the IPC layer message api can be used. IPC layer will copy the application buffer into the message ring.

NOTE

The consumer allocates the message buffers for the message ring using the (`unsigned long msg_ring_paddr, uint32_t msg_size`) in the fsl_ipc_configure_channel API.



Figure 208. Message Ring Model

```
Linux User API
--------------
fsl_ipc_send_msg(uint32_t channel_id,
                 void *src_buf_addr,
                 uint32_t len,
                 fsl_ipc_t ipc);
```

IPC: L2 I/F Layer

The IPC layer provides a method for L2 stack to send the data PDU's using DMA. This is in addition to the Generic API layer and is provided specific for L2 layer.

This layer has API's which are tightly coupled with the L2 downlink flow.

For example an API Is provided to send a tx.request with a set of Transfer blocks to the L1 stack. The individual TB's are further scattered in the memory so it is the task of this api to linearize the TB's. This linearization can be achieved using DMA engine rather than using core.

Table 212. Memory Allocation for the Buffers

Detail	Description
1	L1 as a consumer creates a MSG_CH and provide memory to store TxRequest messages.

Table continues on the next page...

Table 212. Memory Allocation for the Buffers (continued)

Detail	Description
2	L2 allocates memory for linearized TB's and provides to IPC using `fsl_ipc_configure_tq_req`.
3	IPC will use the linearized TB's as DMA destination to linearize the scatter gather buffers.



Figure 209. Message Ring Producer/Consumer Behavior

IPC does the following

- Finds the next position (ping/pong/pang) where the message has to be copied
- Does a memcpy to copy the TxRequest fapi message in Message Ring based on the position
- Creates a DMA descriptor based on Scatter gather list
- Add entries in DMA descriptor to increment the producer index and generate a interrupt to DSP

Using a DMA chain ensures that all the operations are done in a serial fasion.

The API polls for DMA completion so as to notify the producer that the buffers can be freed.

Note: DMA Controller requires the DMA descriptor list be stored in DDR, as the IPC does not allocate memory, it requests the producer to allocate depth + 2 number of buffer in the fsl_ipc_configure_txreq api.

IPC Initialization Sequence

Table 213. Sequential Flow

Term	Definition
1	Linux IPC driver, running on PA reads the boot arguments to find: 1. IPC instance. 2. Numbers of channels in the system. 3. Maximum size of the pointer ring.
2	For each IPC instance `ipc_t` data structure is allocated, which has its own set of channel structures in shared control area.
3	The channel data structures and the pointer rings are allocated in the shared control area.

Table continues on the next page...

Table 213. Sequential Flow (continued)

Term	Definition
4	The consumer application using IPC should set the depth of the channel and if the channel is a MSG channel, set the message size using `fsl_ipc_configure_channel` (on Linux).
5	The consumer application using IPC should set the channel parameter pertaining to indication triggering.
6	For Tx request channel, the producer (PA) has to call the api `fsl_ipc_configure_txreq` to allocate the memory to hold the linearized TBs
7	The Tx Request IPC api `fsl_ipc_send_tx_req` triggers the DMA channel to send the TB's and Tx Request (Linux only).
8	The `fsl_ipc_get_last_tx_req_status` should be called by the producer application to get the status of the last tx request send API (Linux only).
9	IPC layer provides `fsl_ipc_chk_recv_status` api to get the status of all the consumer channels which are not empty. This returns a bitmask where each bit represents a channel number which when set denotes that the channel is not empty (i.e has something to consume). This API is only for polling based consumption

IPC Layer memory requirement

1. Linux

   The IPC layer never allocates any memory, the application invoking IPC should pre-allocate memory and attach to the channel.

   For example:

   - - For MSG_CH, the consumer application when calling fsl_ipc_configure_channel, should set the msg_ring memory equal to the msg_size*depth and aligned to the msg_size
   - - For TXREQ_CH, the producer application should call fsl_ipc_configure_txreq to provide buffers where linearization should be done by DMA.

2. SmartDSP OS

   The IPC layer will allocate OS internal structures for optimizing the access to the IPC control area. Buffers, used by message channels, are to be pre-allocated by the application and the memory manager (os_mem_part_t*) handle will be passed during opening the channel.

9.3.1.5 IPC Layer Components

Linux

The IPC layer is implemented as a library and is dynamically linked with a user space application. The IPC framework has the following components

IPC Lib - The IPC lib is dynamically linked with the application requiring the IPC functionality.

usmmgr (User Space Memory Manager) - Not directly a part of IPC but provides DMAable buffers to user space application. The usmmgr internally uses Linux hugetlbfs framework. Applications need to grab all the required memory during their initialization time and should not request memory from this memory manager during their run time.

Dsp boot loader Application - This apllication does the following

- Load the dsp data in dsp DRR and enable the dsp core/cores

- Allocates the channel control structures at the last 1MB of hugepage

- Allocates the channel pointer bd rings from hugepage

User Space DMA Driver

- Provides an interface for the application to reserve the DMA channels. This will make the fsl DMA driver API not allow the usage of channels reserved by IPC. The IPC layer does not poll on DMA completion rather provides an api for the caller to return the status of last dma activity for the particular instance.

- User space DMA driver requires one dedicated h/w DMA channel per ipc instance for simultaneously sending tx.request packets.

- User space DMA requires an UIO interface to bind it with its dma channels. UIO interface is part of user I/O driver in kernel, so it has to be enabled before running IPC in the Linux Kernel.

9.3.1.6 Functional Specification

IPC Lib Operation

The IPC lib provides shared libraries for the user space application. The library maps the shared control structure in user space. After which the api directly updates the channel structure.

Shared Control Area and IPC channels

For B4 platforms - Some portion of Shared Memory Area is reserved to holds the IPC data structure and the channel structures which are referenced by both Linux and SmartDSP OS. Also the pointer Bd rings are allocated from this area. Refer to the appendix for details

Since Shared Control Area is not mapped as cacheable at PA side by Linux kernel therefore IPC metadata is moved to Shared Memory Area which is mapped as cachable.

9.3.1.7 Linux IPC API

```
/****************

Abbreviations

IN  - Input Parameter

OUT - Output Paramater

M   - Mandatory

O   - Optional

***************/


#include "fsl_ipc_types.h"


/* Defines */

/*****************************************************************************/

#define MAX_TX_REQ_MSG_SIZE 1020
```

```
#define MAX_SG_ENTRIES 128

/*****************************************************************************/

typedef void *fsl_ipc_t;

/*****************************************************************************
 * @ipc_ch_type_t
*****************************************************************************/

typedef enum {

IPC_MSG_CH = 0,

IPC_PTR_CH,

IPC_TXREQ_CH

} ipc_ch_type_t;


/*****************************************************************************
 * Format of the tx.request message associated with Tx.Req Channel type
*****************************************************************************/

typedef struct {

uint8_ttx_request_buf[MAX_TX_REQ_MSG_SIZE];

unsigned longtxreq_linearized_buf;/* Pointer to the linearized buffer*/

} txreq_ctrl_t;


/*****************************************************************************
 * @sg_entry_t
 * Scatter gather table entry
 *
 * src_addr    - dma_capable_physical_addr;
 * is_tb_start - This field is currently ignored.
 * len         - length of tb part
 * is_valid    - is this a valid entry.
 *
*****************************************************************************/
```

```
typedef struct {

unsigned longsrc_addr;

uint32_t len;

uint8_t is_tb_start;

uint8_t is_valid;

} sg_entry_t;



typedef struct {

sg_entry_t  entry[MAX_SG_ENTRIES];

} sg_list_t;



/**************************************************************************

 * @ipc_cbfunc_t

 *

 * Consumer callback function data type.

 * channel_id - [IN][M] unique id of the channel

 * context    - [IN][M] This parameter has different meaning based on

 * channel type:

 *     On a IPC_MSG_CH   - the context is the ipc buffer pointer

 *                         from which the consumer should copy in local buffer

 *

 *          IPC_PTR_CH   - the context may be a buffer pointer

 *          IPC_TXREQ_CH - not valid

 *

 * len        - [IN][M] usually contains the length of the context

 *

**************************************************************************/

typedef void (*ipc_cbfunc_t)(uint32_t channel_id,
                             void *context,
                             uint32_t msg_len);
```

```
/*****************************************************************************
 * @ipc_p2v_t
 *
 * IPC callback function to get the virtual address from ipc user.
 *
 * phys_addr - [IN][M] physical address
 *
 * Return value:
 *             void*     - virtual address
 *
*****************************************************************************/
typedef void* (*ipc_p2v_t)(unsigned long phys_addr);


/*****************************************************************************
 * @fsl_ipc_init
 *
 * Init function to initialize the IPC subsystem.
 *
 * p2vcb        - [IN][M]pointer to a function which does p2v
 * sh_ctrl_area - [IN][M]mem_range_t for shared control area
 * dsp_ccsr     - [IN][M]mem_range_t for dsp_ccsr
 * pa_ccsr      - [IN][M]mem_range_t for pa_ccsr
 * uiodevbuf    - [IN][M]UIO INTERFACE to used
 *
 * Return Value -
 *                fsl_ipc_t handle.
 *                This has to be provided in all subsequent calls to ipc
 *
*****************************************************************************/
fsl_ipc_t fsl_ipc_init(ipc_p2v_t p2vcb,
                       mem_range_t sh_ctrl_area,
```

```
                        mem_range_t dsp_ccsr,
                        mem_range_t pa_ccsr,
                        char uiodevbuf[]);



/*****************************************************************************
 * @fsl_ipc_init_rat
 *
 * Init function to initialize the IPC subsystem.
 *
 * rat_id       - [IN][M]id of the rat for which ipc is instantiated.
 *                 This is used on in multiRAT scenerio.
 * p2vcb        - [IN][M]pointer to a function which does p2v
 * sh_ctrl_area - [IN][M]mem_range_t for shared control area
 * dsp_ccsr     - [IN][M]mem_range_t for dsp_ccsr
 * pa_ccsr      - [IN][M]mem_range_t for pa_ccsr
 * uiodevbuf    - [IN][M]UIO INTERFACE to used
 *
 * Return Value -
 *                fsl_ipc_t handle.
 *                This has to be provided in all subsequent calls to ipc
 *
*****************************************************************************/
fsl_ipc_t fsl_ipc_init_rat(uint32_t rat_id,
                           ipc_p2v_t p2vcb,
                           mem_range_t sh_ctrl_area,
                           mem_range_t dsp_ccsr,
                           mem_range_t pa_ccsr,
                           char uiodevbuf[]);



/*****************************************************************************
 * @ipc_configure_channel
 *
 * To be called one time per channel by the consumer. The channel pointer
 * ring is already created during ipc kernel driver initialization.
```

```
 * NOTE: The number of channels and the max depth of channels is taken as a

 * boot argument to Linux kernel.

 *

 * channel_id      - [IN][M]unique id of the channel

 *

 * depth           - [IN][M]user configurable number of entries in the ring.

 *                   depth <= max depth

 *

 * channel_type    - [IN][M]either of IPC_PTR_CH/IPC_MSG_CH

 *

 * msg_ring_paddr  - [IN]Physical address of the message ring. Required

 *                   only for IPC_MSG_CH

 *

 * msg_size        - [IN]max size of each message.

 *                   For PTR_CH, msg_ring_vaddr, msg_ring_paddr, msg_size

 *                   are all NULL. Required only for IPC_MSG_CH

 *

 * cbfunc          - [IN]The callback function called on receiving interrupt

 *                   from the producer. If cbfunc is NULL, channel does not

 *                   support notifications.

 *

 * Return Value:

 *                   ERR_SUCCESS - no error

 *                   Non zero value - error (check fsl_ipc_errorcodes.h)

 *

*******************************************************************************/

int fsl_ipc_configure_channel(uint32_t channel_id,
                              uint32_t depth,
                              ipc_ch_type_t channel_type,
                              unsigned long msg_ring_paddr,
                              uint32_t msg_size,
                              ipc_cbfunc_t cbfunc,
                              fsl_ipc_t ipc);
```

```
/*****************************************************************************
 * @fsl_ipc_open_prod_ch
 *
 * All params [IN][M]
 * Sets the Producer Initialized value in the channel structure
 *
 * Return Value:
 *              ERR_SUCCESS - no error
 *              Non zero value - error (check fsl_ipc_errorcodes.h)
 *****************************************************************************/
int fsl_ipc_open_prod_ch(uint32_t channel_id,
                         fsl_ipc_t ipc);


/*****************************************************************************
 * @fsl_ipc_configure_txreq
 *
 * For tx request the DSP side creates a msg channel of a particular depth
 * having each entry the size of tx_request fapi message. The size of a
 * tx_request message in IPC is 1024bytes at max. The fapi message
 * can be of max 1020 bytes. Last 4 bytes points to the linearized buffer
 * corresponding to the tx_request fapi message.
 * While DSP allocates memory for tx_request FAPI message, the producer
 * PA allocates memory for linearized buffers.
 * PA allocate memory = (Max size of 1 linearized buffer)*(channel depth +2)
 *
 * NOTE: The extra buffer is used as a spare for IPC internal operations
 *
 * channel_id                    - [IN]unique id of the channel
 * max_txreq_linearized_buf_size - [IN][M] max size of a buffer which holds the
                                   linearized TB.
```

```
 * lbuff_phys_addr               - [IN][M]Start address of the allocated buffer

 *

 * Note: PA should not use this buffer for other operation.

 * Return Value:

 *              ERR_SUCCESS - no error

 ***************************************************************************/

int fsl_ipc_configure_txreq(uint32_t channel_id,
                            unsigned long lbuff_phys_addr,
                            uint32_t max_txreq_linearized_buf_size,
                            fsl_ipc_t ipc);



/***************************************************************************

 *@fsl_ipc_send_ptr

 *

 *Type 1 PRODUCER API, For sending a buffer pointer from producer to consumer.

 *

 * buffer_ptr - [IN][M]The producer buffer pointer which is visible

 *              to producer and consumer

 * len        - [IN][M]length of the producer buffer.

 * Return Value:

 *              ERR_SUCCESS - no error

 *              Non zero value - error (check fsl_ipc_errorcodes.h)

 ***************************************************************************/

int fsl_ipc_send_ptr(uint32_t channel_id,
                     unsigned long buffer_ptr,
                     uint32_t len,
                     fsl_ipc_t ipc);

/***************************************************************************

 *@fsl_ipc_send_msg

 *

 *Type 2 PRODUCER API. For sending a buffer from producer to consumer.

 *IPC copies the buffer into internal message ring.

 *
```

```
 * src_buf_addr - [IN][M]virtual address of the producer buffer

 *

 * len          - [IN][M]length of the producer buffer.

 * Return Value:

 *                ERR_SUCCESS - no error

 ****************************************************************************/

int fsl_ipc_send_msg(uint32_t channel_id,
                     void *src_buf_addr,
                     uint32_t len,
                     fsl_ipc_t ipc);



/****************************************************************************

 *@fsl_ipc_send_tx_req

 *

 * sgl          - [IN][M] A scatter gather list of tb parts

 *

 * tx_reg_addr  - [IN][M] Virtual Address of tx request buffer in producer's

 *                memory this buffer is copied on to the message ring

 * Return Value -

 *                ERR_SUCCESS - no error

 *                Non zero value - error (check fsl_ipc_errorcodes.h)

 ****************************************************************************/

int fsl_ipc_send_tx_req(uint32_t channel_id,
                        sg_list_t *sgl,
                        void *tx_req_vaddr,
                        uint32_t tx_req_len,
                        fsl_ipc_t ipc);



/****************************************************************************

 * @fsl_ipc_get_last_tx_req_status

 *

 *This api should be called by the producer to check the completion of

 *last DMA transfer initiated by fsl_ipc_send_tx_req API.
```

```
 *
 * Return Value
 *     int-status - TXREQ_DONE
 *                  TXREQ_IN_PROCESS
 *                  TXREQ_ERR
 *(defined in fsl_ipc_errorcodes.h)
 ***************************************************************************/
int fsl_ipc_get_last_tx_req_status(fsl_ipc_t ipc);


/***************************************************************************
 * @fsl_ipc_recv_ptr
 *
 *Consumer API, called when the consumer is polling
 *
 * addr - [IN][M]ipc copies this value from the ptr ring, and increments the
 *        consumer index. (value is of unsigned long in most cases)
 * len  - [IN][M]length provided by the producer
 *
 * Return Value:
 *              ERR_SUCCESS - no error
 *              Non zero value - error (check fsl_ipc_errorcodes.h)
 ***************************************************************************/
int fsl_ipc_recv_ptr(uint32_t channel_id,
                     unsigned long *addr,
                     uint32_t *len,
                     fsl_ipc_t ipc);


/***************************************************************************
 * @fsl_ipc_recv_ptr_hold
 *
 *Consumer API, called when the consumer is polling
 *
```

```
 * addr - [IN][M]ipc copies this value from the ptr ring, and does not
          increment the consumer index. (value is of unsigned long in most cases).
 *        The consumer index is updated by calling fsl_ipc_set_consumed_status

 *

 * len  - [IN][M]length provided by the producer

 *

 * Return Value:

 *              ERR_SUCCESS - no error

 *              Non zero value - error (check fsl_ipc_errorcodes.h)

 ***************************************************************************/

int fsl_ipc_recv_ptr_hold(uint32_t channel_id,
                          unsigned long *addr,
                          uint32_t *len,
                          fsl_ipc_t ipc);



/***************************************************************************

 * @fsl_ipc_recv_msg

 *

 * Consumer API, called when the consumer is polling

 *

 * addr - [IN][M]IPC copies from the message ring into the buffer pointer
          provided by the consumer, and increments the consumer index.

 *

 * len  - [IN][M]length of the copied buffer

 *

 * Return Value:

 *              ERR_SUCCESS - no error

 *              Non zero value - error (check fsl_ipc_errorcodes.h)

 ***************************************************************************/

int fsl_ipc_recv_msg(uint32_t channel_id,
                     void *dst_buffer,
                     uint32_t *len,
                     fsl_ipc_t ipc);
```

```
/*****************************************************************************
 * @fsl_ipc_recv_msg_ptr
 *
 * Consumer API, called when the consumer is polling, and when the
 * consumer is using the buffer in the message ring without copying
 * in the local buffer. (Zero Copy)
 * When consumed fully the API fsl_ipc_set_consumed_status should be
 * called, this would increment the consumer index.
 *
 * channel_id - [IN][M] unique id of the channel
 *
 * dst_buffer - [OUT][M] IPC copies the virtual address of message buffer
 *
 * len        - [IN][M] length of the copied buffer
 *
 * Return Value:
 *              ERR_SUCCESS - no error
 *              Non zero value - error (check fsl_ipc_errorcodes.h)
 *****************************************************************************/
int fsl_ipc_recv_msg_ptr(uint32_t channel_id,
                         void **dst_buffer,
                         uint32_t *len,
                         fsl_ipc_t ipc);


/*****************************************************************************
 * @fsl_ipc_set_consumed_status
 *
 * channel_id - [IN][M] unique id of the channel
 *
 * Called along with fsl_ipc_recv_msg_ptr to increment the consumer index
 * on that channel
 * Return Value:
```

```
 *              ERR_SUCCESS - no error

 *              Non zero value - error (check fsl_ipc_errorcodes.h)

 ***************************************************************************/

int fsl_ipc_set_consumed_status(uint32_t channel_id,
                                fsl_ipc_t ipc);



/***************************************************************************

 * @fsl_ipc_chk_recv_status

 *

 * The api checks all the consumer channels owned by the _calling process_ to

 * find out which has a msg/ptr received.

 *

 * bmask  - [OUT][M] There can be a max of 64 channels. Each bit set

 *          represents a channel has recieved a message/ptr.

 *          Bit 0 MSB - Bit 64 LSB

 *          (Bit = Channel id)

 * Return Value -

 *          ERR_SUCCESS - no error

 *          Non zero value - error (check fsl_ipc_errorcodes.h)

 ***************************************************************************/

int fsl_ipc_chk_recv_status(uint64_t *bmask,
                            fsl_ipc_t ipc);

/ ***************************************************************************
*
* IPC library will be enhanced to provide following API for L2/Demo application in order to
*  enable them for performing DSP recovery:
*        Call back registration API
*         Since DSP core/L1 crash is an asynchronous event therefore IPC library will
*         provide following API to L2/demo application to register a call back which will be
*         invoked whenever one or more DSP cores have gone into stray state.
*
* INPUT
* whereas:
    typedef void (*fsl_defense_cb)(uint32_t core_mask)
     cb    : pointer to function of type “fsl_defence_cb”
*  Return Value - core_mask

 ***************************************************************************/
 void fsl_L1_defense_register_cb(fsl_defense_cb *cb);
```

```
/ ************************************************************************
*
* Once L2/Demo application gets the notification of DSP core(s) stray state via its registered
callback,
   it needs to trigger the recovery using following API
*
* INPUT
* Whereas:
              typedef struct{
                int     reset_core_flag;
                int     core_id;
                char     *dsp_filename;
              } reload_dsp_core_info;

              typedef struct{
                int     hw_sem_num;
                int     reset_mode;
                int     maple_reset_mode;
                int         debug_print;
                reload_dsp_core_info    shDspCoreInfo[6];
                reload_dsp_core_info     reDspCoreInfo[6];
              } dsp_core_info;

 * reset_core_flag    is the bitmask generated from WSRSR for individual
                        core.
 * dsp_filename        name of the dsp image with complete path.
 * hw_sem         specifies is hardware semaphore is used or not.
 * reset_mode         specifies which mode to be used MODE1, MODE2, or
                        MODE3.
 * maple_reset_mode    specifies which in mode maple reset is to be done.
                     "maple_reset_mode;" can have any combination if below mentioned enums
                         RESET_MAPLE_1 = 0x2, reset first maple engine
                         RESET_MAPLE_2 = 0x4, reset second maple engine
                         RESET_MAPLE_3 = 0x8,  reset third maple engine
                         Other values which can be true are 0x6,
                         0xA, 0xC, 0xE

* Return Value - WARM RESET status

* NOTE:  If the L2/demo application do not intend to use IPC the first argument should
               be passed as NULL
*************************************************************************/
int fsl_start_L1_defense(fsl_ipc_t ipc, dsp_core_info DspCoreInfo);
```

9.3.1.8 Shared Control Structure

```
/************************************************************************
 @Description   Heterogeneous OS global control structure

*************************************************************************/
#ifdef B913x
typedef struct {
os_het_init_t           initialized;
/* Initialization indication strcuture */
uint32_tshared_ctrl_size;
```

```
/* Size of the shared memory for control information in bytes -
   starts at the base of the PA managed shared memory.
   Mimumum (and default) size is 4 KB */
os_het_mem_t            pa_shared_mem;
/* PA shared memory region */
os_het_mem_t            sc_shared_mem;
/* SC shared memory region */

os_het_ipc_t            (*ipc)[];

void                    *aic;
/* Pointer to shared AIC configuration control structure;
 * as an offset from the base of the shared address space */
os_het_smartdsp_log_t   (*smartdsp_debug)[];
/* Pointer to where SmartDSP logs system events; PA initializes
 * an array with the number of entries as there is SC cores*/
os_het_debug_print_t    *het_debug_print;

#ifdef CONFIG_MULTI_RAT
uint32_t                num_ipc_regions;
#endif
} os_het_control_t;
#else /* B4860 */

typedef struct
{
   uint32_t                start_validation_value;
    /* Initialization indication strcuture */
    os_het_init_t           initialized;
    /* SET BY PA: PA shared memory region; */
    os_het_mem_t            pa_shared_mem;
    /* SET BY DSP: SC shared memory region; */
    os_het_mem_t            sc_shared_mem;
    /* Pointer to array of os_het_ipc_t
     * Pointer IPC heterogeneous structure */
    uint64_t                ipc;
    /* Pointer l1_defence pointer */
    uint64_t                 l1d;
    /* SET BY PA: Pointer to where SmartDSP logs system events
     * PA initializes an array with the number of entries as
     * there is SC cores*/
    uint64_t                smartdsp_debug;
    os_het_debug_print_t    het_debug_print;
    /* SET BY DSP: Size of the shared memory for control information in
     * bytes - Mimumum size is 4 KB set by PA*/
    uint32_t                shared_ctrl_size;

    /* Number of IPC regions - only for multimode usages */
    uint32_t                num_ipc_regions;
    uint32_t                end_validation_value;
} os_het_control_t;

#endif
extern os_het_control_t  *g_os_het_control;
/* Pointer to the base address of the heterogeneous OS control strcuture */

/*****************************************************************************//**
 @Description   IPC message descriptor

*//*****************************************************************************/
```

```
#ifdef B913x
typedef struct {
    /* Pointer to the message; as an offset from the base of the shared
       address space */
    uintptr_t   msg_ptr;
    /* Size of the message; may be set to OS_HET_UNSPECIFIED_LEN on
       OS_HET_IPC_POINTER_CH channel types only */
    uint32_t    msg_len;
} os_het_ipc_bd_t;
#else
typedef struct {
    /* Pointer to the message; as an offset from the base of the shared
       address space */
    uint64_t   msg_ptr;
    /* Size of the message; may be set to OS_HET_UNSPECIFIED_LEN on
       OS_HET_IPC_POINTER_CH channel types only */
    uint32_t   msg_len;
} os_het_ipc_bd_t;
#endif

/*************************************************************************//**
 @Description   Types of IPC channels

*//*************************************************************************/
typedef enum {
    /* Indicates that the consumer will pre-populate the channel's
       ptr_bd_base[].msg_ptr entires; The producer MUST use the pre-populated
       pointers when producing messages on the channel */
    OS_HET_IPC_MESSAGE_CH = 0,
    /* Indicated that the producer MUST provide pointers to the
       ptr_bd_base[].msg_ptr entries whenever a new messag in produced */
    OS_HET_IPC_POINTER_CH = 1
} os_het_ipc_ch_types_t;

/*************************************************************************//**
 @Description   IPC channel control structure

 @Cautions      Using the macros OS_HET_INCREMENT_CONSUMER() and
OS_HET_INCREMENT_PRODUCER()
is the recommended way of
incrementing the tracker counters;
Direct access should be avoided.


*//*************************************************************************/
typedef struct {
#ifdef B4860
    uint32_t                start_validation_value;
#endif
    /* Indicates whether the mailbox is initialized by the producer;
       This field is written by the producer */
    uint32_t                producer_initialized;
    /* Indicates whether the mailbox is initialized by the consumer;
       This field is written by the consumer */
    uint32_t                consumer_initialized;
    /* Mailbox ID - may be used by the applicaiton to identify the channel;
       This field is written by Linux during boot. Can be an arbitrary number
       provided that it is unique in the system */
    uint32_t                id;
    /* Producer/consumer tracker;
```

```
       This field is set to {0} by Linux during boot */
    os_het_tracker_t        tracker;
    /* Size of the mailbox BD ring;
       This field is set by the consumer. May not be larger than
       os_het_ipc_t.ipc_max_bd_size */
    uint32_t                bd_ring_size;
    /* Size (in Bytes) of the maximal message that can be passed
       on this channel;
       This field is set by the consumer ONLY if ch_type is
       OS_HET_IPC_MESSAGE_CH otherwise MUST be set to 0xFFFFFFFF */
    uint32_t                max_msg_size;
    /* Type of the channel; This field is set by the consumer */
    os_het_ipc_ch_types_t   ch_type;
    /* Base of the mailbox pointer BD; as an offset from the base of
       the shared address space;
       This field is allocated by Linux during boot based on
       os_het_ipc_t.ipc_max_bd_size */
#ifdef B913x
    os_het_ipc_bd_t         (*bd_base)[];
#else
    uint64_t                bd_base;
#endif
    /* Type of indication to generate to the destination;
       This field is written by the consumer */
    os_het_ipc_ind_t        ipc_ind;
    /* Offset address from base of specific indication register set;
       This field is written by the consumer */
    uint32_t                ind_offset;
    /* Value to write to ind_offset;
       This field is written by the consumer */
    uint32_t                ind_value;
    /* reserved field for pa usage */
    uint32_t                pa_reserved[2];
    /* Future compatibility semaphore pointer;
       This field is set to NULL by Linux during boot */
#ifdef B913x
    void                    *semaphore_pointer;
#else
    uint64_t                semaphore_pointer;
    uint32_t                 end_validation_value;
#endif
} os_het_ipc_channel_t;


/*****************************************************************************//**
 @Description   IPC global control structure

*//*****************************************************************************/
typedef struct {
#ifdef B4860
    uint32_t                start_validation_value;
#endif
    /* Number of channels in the ipc_channels array;
       this is a bootarg to Linux and will be set by Linux accordingly */
    uint32_t                num_ipc_channels;
    /* Maximal size that the ipc_channel.bd_ring_size may have;
       this is a bootarg to Linux and will be set by Linux accordingly */
    uint32_t                ipc_max_bd_size;
    /* Pointer to the mailboxes control structure array;
       has os_het_ipc_t.num_ipc_channels entries
```

```
       as an offset from the base of the shared address space */
#ifdef B913x
    os_het_ipc_channel_t    (*ipc_channels)[];
#else
    uint64_t                ipc_channels;
    uint32_t                end_validation_value;
#endif
} os_het_ipc_t;

static inline int os_het_ch_free_bds(void *ipc_ch)
{
os_het_ipc_channel_t *l_ipc_ch;
l_ipc_ch = (os_het_ipc_channel_t *)ipc_ch;

if ((l_ipc_ch)->tracker.producer_num >=
(l_ipc_ch)->tracker.consumer_num)
return (l_ipc_ch)->bd_ring_size -
((l_ipc_ch)->tracker.producer_num -
(l_ipc_ch)->tracker.consumer_num);
else
return (l_ipc_ch)->bd_ring_size -
((l_ipc_ch)->tracker.producer_num -
(l_ipc_ch)->tracker.consumer_num + (1ULL << 32));
}

/*************************************************************//**
 @Collection    Initialization indication values

 @{
*//*************************************************************/
/* The resource is initializaed */
#define OS_HET_INITIALIZED              0xFEDCBA98UL
/* The resource is initializaed for multimode work*/
#define OS_HET_INITIALIZED_MULTIMODE    0xEDCBA987UL
/* The resource is uninitialized */
#define OS_HET_UNINITIALIZED            0x00000000UL
/* @} */


/*************************************************************//**
 @Collection    Hardware/Software semaphore values

 @{
*//*************************************************************/
/* Sempahore is taken by the PA domain */
#define OS_HET_PA_SEMAPHORE_VAL         0xFF
/* Sempahore is taken by the SC domain */
#define OS_HET_SC_SEMAPHORE_VAL         0xFE
/* Semaphore is free */
#define OS_HET_FREE_SEMAPHORE_VAL       0x00
/* @} */

/* PASS / FAIL macro */
typedef enum {
    OS_HETERO_FAIL      = 0,
    OS_HETERO_SUCCESS   = 1
} os_het_status_t;


/*************************************************************************//**
```

```
 @Description   Initialization Control structure

This structure will be used to indicate that both OS
domains have initialized a specific structure

*//*************************************************************************/
typedef struct {
    /* Indicates whether the overall control structure is initialized
       (PA side) */
    uint32_t                pa_initialized;
    /* Indicates whether the overall control structure is initialized
       (SC side) */
    uint32_t                sc_initialized;
} os_het_init_t;

/*************************************************************************//**
 @Description   Producer/Consumer tracker

The producer and consumer each perform counter++ to
their counter. It is assumed the size of what the
tracker is tracking is less than < MAX_UINT_32

 @Cautions      The counters must only be incremented, never decremented

*//*************************************************************************/
typedef struct {
    /* Number of items the producer produced */
    uint32_t      producer_num;
    /* Number of items the consumer consumed */
    uint32_t      consumer_num;
} os_het_tracker_t;


#define OS_HET_CALCULATE_ADDR(BASE, OFFSET) (void *)\
((uint8_t *)(BASE) + (uint32_t)(OFFSET))
/* Used by the various OS to calculate an address in it's
   own virtual address space  */

/*************************************************************************/
/*@Description   SmartDSP Event log*/

/*************************************************************************/
#ifdef B913x
typedef struct {
    /* Pointer to the core's event log(physical address) */
    uintptr_t   base_address;
    /* Size of the event log (in bytes) */
    uint32_t    size;
    /* Pointer to the last error in SmartDSP. A value of 1 (OS_SUCCESS)
       means no error */
    uint32_t*   last_error;
} os_het_smartdsp_log_t;
#else /* B4860 */
typedef struct {
    uint32_t    start_validation_value;
    /* Pointer to the core's event log(physical address) */
    uint64_t    base_address;
    /* Size of the event log (in bytes) */
    uint32_t    size;
    /* Pointer to the last error in SmartDSP. A value of 1 (OS_SUCCESS)
```

```
       means no error */
    uint64_t    last_error;
    uint32_t    end_validation_value;
} os_het_smartdsp_log_t;
#endif

#ifdef B913x
/* The number of tables used by debug print for each core */
#define NUM_OF_DBGP_TABLES_PER_SC           2

/*  The number of SC cores supported by debug print */
#define NUM_OF_DBGP_SC_CORES                1

#else
/*  The Maximum nuber of segments of the buffer*/
#define MAX_NUM_OF_SEGMENT                    32
#endif
typedef struct {
uint64_t system_clock;
uint64_t DSP_clock;
} debug_print_clocks_t;

/*****************************************************************************//**
 @Description   SmartDSP Debug print structure

*//*****************************************************************************/
#ifdef B913x
typedef struct {

/* Pointer to the base address of the SC VTB */
void                *buffer_location;
/* Size of each segment in VTB */
uint32_t             segment_size;
/* Number of VTB segments */
uint32_t             num_of_segments;
/* Tracker for segment number; SC client is the producer and
   PA engine is the consumer */
os_het_tracker_t     tracker;
/* Clock synchronization */
debug_print_clocks_t sample_clock;
/* Overflow indicator */
uint32_t             overflow;
/* segmet information */
void                 *segment_info;
/* reserved */
uint32_t             reserved[4];

} os_het_debug_print_sc_t;


typedef struct {

    /* PA Debug Print shared memory region */
    os_het_mem_t            pa_debug_print_shared;
    /* SC Debug Print shared memory region */
    os_het_mem_t            sc_debug_print_shared;
    /* Number of entries in sc_debug_print[];
       Should be NUM_OF_DBGP_TABLES_PER_SC*NUM_OF_DBGP_SC_CORES */
    uint32_t                sc_array_size;
    /* SC debug print main structure, size of */
```

```
    os_het_debug_print_sc_t (*sc_debug_print)[];

} os_het_debug_print_t;
#else /* B4860 */

typedef struct
{

    /* Pointer to the base address of the SC VTB */
    uint64_t                buffer_location;
    /* Size of each segment in VTB */
    uint32_t                segment_size;
    /* Number of VTB segments */
    uint32_t                num_of_segments;
    /* Tracker for segment number
     * SC client is the producer and PA engine is the consumer */
    os_het_tracker_t        tracker;
    /* 64 bits clock for each segment */
    debug_print_clocks_t    segment_clock[MAX_NUM_OF_SEGMENT];
    /* Overflow indicator */
    uint32_t                overflow;
    uint64_t                reserved[4];

}os_het_debug_print_sc_t;


typedef struct
{
    /* PA Debug Print shared memory region */
    os_het_mem_t            pa_debug_print_shared;
    /* SC Debug Print shared memory region */
    os_het_mem_t            sc_debug_print_shared;
    /* SC Debug Print control region */
    uint32_t                sc_array_size;
    /* SC debug print main structure*/
    os_het_debug_print_sc_t sc_debug_print;
} os_het_debug_print_t;

#endif
/*****************************************************************************//**
 @Description   Memory Descriptor structure.

This structure will be used to describe a shared memory slab.

*//*****************************************************************************/
#ifdef B913x
typedef struct {
    /* Indicates the start address of a memory region */
    uintptr_t   start_addr;
    /* Indicates the size of a memory region */
    uint32_t    size;
} os_het_mem_t;
#else /*B4860 */
typedef struct {
    /* Indicates the start address of a memory region */
    uint64_t    start_addr;
    /* Indicates the size of a memory region */
    uint64_t    size;
} os_het_mem_t;
#endif
```

```
/*************************************************************************
@L1 defense Structures and Definations
*************************************************************************/
#ifdef B4860
#define MAX_NUM_OF_DSP_CORES 6

#define HET_START_VALID_VALUE    0x12345678
#define HET_END_VALID_VALUE      0x87654321
/*************************************************************************
@Description warm reset modes
*************************************************************************/
typedef enum {
        NO_WARM_RESET = 0x0,
        MODE_1_ACTIVE = 0x1, /* L1 scenrio mode 1 */
        MODE_2_ACTIVE = 0x2, /* L1 scenrio mode 2 */
        MODE_3_ACTIVE = 0x3  /* L1 scenrio mode 3 */
} os_het_l1d_mode_t;

/*************************************************************************
@Description mode 2 maple reset activation
*************************************************************************/
typedef enum {
        RESET_MAPLE_1 = 0x2, /* reset first maple engine */
        RESET_MAPLE_2 = 0x4, /* reset second maple engine */
        RESET_MAPLE_3 = 0x8  /* reset third maple engine */
} os_het_l1d_reset_maple_t;

/*************************************************************************
 @Description   L1 level status enumeration
 *************************************************************************/
 typedef enum {
        /* data was corrupted */
        OS_HET_ERR_L1D_MEMORY_CORRUPTED     = 0x0037FFBF,
        /* Warm reset mode is invalid or unsupported */
        OS_HET_ERR_L1D_MODE_INVALID         = 0x0037FFED,
        /* invalid L1-defense initiation parameters */
        OS_HET_ERR_L1D_FUNCTION_INVALID     = 0x0037FFEC,
        /* System is already in reset mode */
        OS_HET_ERR_L1D_ALREADY_ACTIVE       = 0x0037FFC3,
        /* Unknown error */
        OS_HET_ERR_L1D_UNKNOWN              = 0x00370002,

        /* NMI received, Indication that core is ready for reset */
        OS_HET_INFO_L1D_READY_FOR_RESET     = 0x1037FFBE,
        /* beginning execution of warm reset os initialization */
        BEGIN_WARM_RESET_OS_INIT            = 0x1037FFBD,
        /* beginning execution of warm reset application initialization */
        BEGIN_WARM_RESET_APP_INIT           = 0x1037FFBC,
        /* warm reset was completed successfully */
        WARM_RESET_SUCCESS                  = 0x1037FFBB

} os_het_l1d_status;
/*************************************************************************
@Description Heterogeneous OS L1 defense control structure
*************************************************************************/
typedef struct {
        /* validation value for checking for corrpution in case of reset */
        uint32_t start_validation_value;
```

```
        /* definition of the current L1 defense mode to be used*/
        uint32_t warm_reset_mode;
        /* only relevant if MODE_2 is defined in reset_mode - whether/which
         * maple engine should be reset */
        uint32_t reset_maple;
        /* status and error codes for each of the DSP cores */
        uint32_t reset_status[MAX_NUM_OF_DSP_CORES];
        /* validation value for checking for corrpution in case of reset */
        uint32_t end_validation_value;
} os_het_l1d_t;

#endif
```

9.3.2 IPC User Guide

9.3.2.1 IPC User Guide

Heterogeneous IPC Module

IPC module is provided as user space library (libipc.so) and loadable kernel modules. Apart from providing APIs for L2-L1 communication, a hugepage based user space shared memory manager is also provided as a user space library (libmem.so). In order to use IPC services, user space application needs to be linked with these libraries

Directory Structure

```
ipc/include  - Include directory for IPC public and private header
               files
ipc/lib      - Source code of IPC, user space ipc helper memory
               manager, user space dma driver
fsl_shm      - Source code of user space Shared Memory Manager
dsp_boot     - Source code of DSP boot loader
test         - Reference Test application which uses IPC channels
Kernel       - It contains the IPC kernel code. Compilation will
               generate three kernel loadable modules namely,
               hetmgr.ko, shm.ko and l1d.ko
```

Compilation Procedure

```
Following needs to be modified in kernel image before booting Linux:
Enable the DMA UIO user driver in kernel so to access DMA channel from user space and disable
kernel dma engine.
        Disable the Kernel dma engine.
        [ ] DMA Engine support  --->

        Enable Userspace's I/O driver
        Device Drivers  --->
               <*> Userspace I/O drivers  --->
                    <*> Freescale DMA support

I. Kernel loadable modules:
   1. cd ipc/kernel
   2. Set KERNEL_DIR as the path of Linux source code
   3. Set CROSS_COMPILE as the <path of toolchain>/.../powerpc-linux-gnu- (toolchain should be 64
bit toolchain)
```

```
   4. run make B4860=1
        This will generate the hetmgr.ko, shm.ko and l1d.ko.
        Install the ".ko" with the following command after the Linux is booted.
        Run step 5 only if you want to build kernel modules for multirat applications
   5. run make clean; make B4860=1 CONFIG_MULTI_RAT=1

II. User space code:
   1. cd ipc
   2. Set CROSS_COMPILE as the <path of toolchain>/.../powerpc-linux-gnu- (toolchain should be 32
bit toolchain)
   3. make B4860=1
   4. Following user space libraries and binaries get generated:
      1. under ipc directory
         - libipc.so,
         - libmem.so,
         - ipc_test,
         - ipc_test67
         - l1d_app

      2. under dsp_boot directory
         - libdspboot.so
         - dsp_bt
```

Required u-boot bootargs

```
Please append the following in bootargs:
  - default_hugepagesz=256m
  - hugepagesz=256m
  - hugepages=1

Set DDR size visible to Linux.
  - setenv bootm_size=0x100000000 (This make 0-4 GB for linux and rest for SDOS 4-6 GB)
OR
Set DDR size visible to Linux.
  - setenv bootm_size=0x70000000 (any value less than 2 GB is good enough)
```

Update DDR size to 6GB on uboot

```
This is required for only when L1d and IPC is suppose to work with New Memory Map (4GB-2GB)
DDRC1:
 tftp the SPD binary onto the board
   - tftp 0x1000000 <path/of/bin>spd_1866_dual_rank_4GB_DDRC1.bin
 Update SPD binary with help of i2c
   - mw.b ffdf0055 0x8
   - i2c write 0x1000000 0x51 0 0x80
DDRC2:
 tftp the SPD binary onto the board
   - tftp 0x2000000 <path/of/bin>spd_1866_single_rank.bin
 Update SPD binary with help of i2c
   - mw.b 0xffdf0055 0x8
   - i2c write 0x2000000 0x53 0 0x80
```

Test Application (Single RAT only Solutions)

```
1. insmod /usr/driver/IPC/single_rat/hetmgr.ko
     Following modules params can be passed to override their default
     values:
```

```
       - dsp_shared_size  (default value 0x1000000)
       - dsp_private_addr (default value 0x80000000)
       - dsp_private_size (default value 0x7FF00000)
       - shared_ctrl_addr (default value 0xFFF00000)
       - shared_ctrl_size (default value 0x100000)
       - max_num_ipc_channels (default value 64)
       - max_channel_depth    (default value 16)

NOTE: Module params are optional, required only when default setting needs to be altered.

2. insmod /usr/driver/IPC/single_rat/shm.ko

3. insmod /usr/driver/IPC/single_rat/l1d.ko

4. Create device nodes with major number flashed while loading kernel modules
     cat /proc/devices will show the major numbers as well

     mknod /dev/fsl_shm c <major number> 0
     mknod /dev/het_mgr c <major number> 0
     mknod /dev/fsl_l1d c <major number> 0


5. Populate shmmax value in proc file system
     echo 0x10000000 > /proc/sys/kernel/shmmax

6. Command to load SDOS image

    Format
    ------
    $./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">

    $ ./dsp_bt -h 1 -c 0 -i c0.bin

             Usage: ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">
             whereas,
              <hw_sem> : 1 to 7 for use hardware semaphore
                       : 0 No hardware semaphore used
              <core_id> : 0 to 5 dsp core number
              <image_name> : dsp image name
                          -c option should be followed by -i option
              <shared_image> : Shared Image name

NOTE: If the semaphore number is specified as 0, it means semaphore is not used.
      Core number specifies the star core number, ranges 0-5.

7. Test applications (ipc_test, ipc_test67) can be started once the DSP/SDOS image
   is loaded successfully with the help of dsp loader(dsp_bt).

   Test runs with the help of 6 IPC channels, where for each pair of channels PowerPC is producer
   and DSP is consumer. For each of the channel pairs a seperate thread is
   started by the test application.

   The channel pairs are
        2-3     Channel 2 is a MSG Channel where DSP is consumer
                Channel 3 is a PTR Channel where PowerPC is consumer.

                Linux sends a message on Channel 2, DSP returns back the pointer of the
                message on channel 3.

                On Channel 2, DSP receives VIRQ
```

```
                On Channel 3, Linux polls for recv pointer.

        4-5     Channel 4 is a PTR channel where Linux is a consumer.
                Channel 5 is a MSG channel where DSP is a consumer.

                Linux receives a ptr on channel 4, copies the length number of characters
                from the pointer into a local buffer and sends the buffer on to channel 5.

                On Channel 4, Linux Polls
                On Channel 5, DSP Polls

        6-7     Channel 6 is a TX Request Channel where Linux is producer.
                Channel 7 is a PTR channel were linux is consumer.

                Linux sends a dummy TX request fapi message a set of buffer using DMA.
                DSP returns to linux a pointer which points to a memory location
                containing first 16 bytes of fapi message and the first buffer.

   Format
   ------
   For channel 2-3,4-5
   $./ipc_test
   Usage: ./ipc_test -r <rat_id> -i <nr_msg>
   whereas,
   <rat_id> : 0 for SingleRAT
            : 1 for MultiRAT
   <nr_msg> : Number of Messages to be exchanged on an IPC channel
   OR
   $ ./ipc_test -r 0 -i 20

   For Channel 6-7
   $./ipc_test67
   Usage: ./ipc_test67 -r <rat_id> -i <nr_msg>
   whereas,
   <rat_id> : 0 for SingleRAT
            : 1 for MultiRAT
   <nr_msg> : Number of Messages to be exchanged on an IPC channel
   OR
   $ ./ipc_test67 -r 0 -i 20

NOTE: ipc_test67 requires a user input,which is the uio interface corresponding
      to the DMA channel.

     for Eg:
        Enter uio_interface you want to use
        like:
        /dev/uio0

        /dev/uio0

NOTE: If -i option is not provided the test run till eternity.
      If -r option is not provided, it take rat_id as 0.
```

Test Application (Multi RAT)

```
0. Since SDOS cannot be reloaded, reboot the system to load multi rat kernel modules.

1. insmod /usr/driver/IPC/multi_rat/hetmgr.ko
     Following modules params can be passed to override their default
```

```
     values:
       - dsp_shared_size  (default value 0x1000000)
       - dsp_private_addr (default value 0x80000000)
       - dsp_private_size (default value 0x7FF00000)
       - shared_ctrl_addr (default value 0xFFF00000)
       - shared_ctrl_size (default value 0x100000)
       - max_num_ipc_channels (default value 64)
       - max_channel_depth    (default value 16)

NOTE: Module params are optional, required only when default setting needs to be altered.

2. insmod /usr/driver/IPC/multi_rat/shm.ko

3. insmod /usr/driver/IPC/multi_rat/l1d.ko

4. Create device nodes with major number flashed while loading kernel modules
     cat /proc/devices will show the major numbers as well

     mknod /dev/fsl_shm c <major number> 0
     mknod /dev/het_mgr c <major number> 0
     mknod /dev/fsl_l1d c <major number> 0

5. Populate shmmax value in proc file system
     echo 0x10000000 > /proc/sys/kernel/shmmax

6. Command to load SDOS Multi-RAT image

    Format
    ------
    $ ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">

    $ ./dsp_bt -h 1 -c 0 -i c0.bin -c 1 -i c1.bin

             Usage: ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">
             whereas,
              <hw_sem> : 1 to 7 for use hardware semaphore
                       : 0 No hardware semaphore used
              <core_id> : 0 to 5 dsp core number
              <image_name> : dsp image name
                          -c option should be followed by -i option
              <shared_image> : Shared Image name


NOTE: If the semaphore number is specified as 0, it means semaphore is not used.
      Core number specifies the star core number, ranges from 0-5.

7. Test applications (ipc_test, ipc_test67) can be started once the DSP multi rat images
   are loaded successfully with the help of dsp loader(dsp_bt),present in rootfs under /ipc.


   ipc_test and ipc_test67 are upgraded for testing multirat support from PA side.
   rat id should be added after the executable name

   Format
   ------
   $./ipc_test
   Usage: ./ipc_test -r <rat_id> -i <nr_msg>
   whereas,
   <rat_id> : 0 for SingleRAT
            : 1 for MultiRAT
```

```
   <nr_msg> : Number of Messages to be exchanged on an IPC channel
   OR
   $ ./ipc_test -r 1 -i 20

   For Channel 6-7
   $./ipc_test67
   Usage: ./ipc_test67 -r <rat_id> -i <nr_msg>
   whereas,
   <rat_id> : 0 for SingleRAT
            : 1 for MultiRAT
   <nr_msg> : Number of Messages to be exchanged on an IPC channel
   OR
   $ ./ipc_test67 -r 1 -i 20

NOTE: ipc_test67 requires a user input,which is the uio interface corresponding
      to the DMA channel.

     for Eg:
        Enter uio_interface you want to use
        like:
        /dev/uio0

        /dev/uio1

NOTE: If -i option is not provided the test run till eternity.
      If -r option is not provided, it take rat_id as 0.
```

l1d Test Application (Single Core only Solutions)

```
1. insmod /usr/driver/IPC/single_rat/hetmgr.ko
     Following modules params can be passed to override their default
     values:
       - dsp_shared_size  (default value 0x1000000)
       - dsp_private_addr (default value 0x80000000)
       - dsp_private_size (default value 0x7FF00000)
       - shared_ctrl_addr (default value 0xFFF00000)
       - shared_ctrl_size (default value 0x100000)
       - max_num_ipc_channels (default value 64)
       - max_channel_depth    (default value 16)

NOTE: Module params are optional, required only when default setting needs to be altered.

2. insmod /usr/driver/IPC/single_rat/shm.ko

3. insmod /usr/driver/IPC/single_rat/l1d.ko

4. Create device nodes with major number flashed while loading kernel modules
     cat /proc/devices will show the major numbers as well

     mknod /dev/fsl_shm c <major number> 0
     mknod /dev/het_mgr c <major number> 0
     mknod /dev/fsl_l1d c <major number> 0

5. Populate shmmax value in proc file system
     echo 0x10000000 > /proc/sys/kernel/shmmax

6. Command to load SDOS image

    Format
```

```
    ------
    $./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">

    $ ./dsp_bt -h 1 -c 0 -i c0.bin

             Usage: ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">
             whereas,
              <hw_sem> : 1 to 7 for use hardware semaphore
                       : 0 No hardware semaphore used
              <core_id> : 0 to 5 dsp core number
              <image_name> : dsp image name
                         -c option should be followed by -i option
              <shared_image> : Shared Image name

NOTE: If the semaphore number is specified as 0, it means semaphore is not used.
      Core number specifies the star core number, ranges 0-5.

7. Test application (l1d_app) can be started once the DSP/SDOS image
   is loaded successfully with the help of dsp loader(dsp_bt).

   Test sends an ioctl call to the kernel, inside the kernel it waits on a waitqueue,
   the ioctl returns only if an interrupt is generated by the dsp core towards the PA core via
MPIC interface.
   This interrupt sets the WSRSR register, value set in this register is returned to the proccss
invoking the ioctl
   and this register is cleared.

   Format
   ------
   L1 defense without IPC is use
   $ ./l1d_app 0 0
   Usage:
     ./l1d_app <single core/multi core> <ipc in use>
   Where as,
      0: single core
      1: multi core
      0: IPC not used
      1: IPC used

   OR
   Usage:
     ./l1d_app
   Where as,
     "./l1d_app" means <single core> <ipc not used>
     Same as "./l1d_app 0 0"

NOTE: This test requires user inputs,which are mentioned below:

     for Eg:
        Enter your choice
          0 means not in use for all Parameters
          Only Values mentioned below are valid, rest all values are invalid

        WARM_RESET_MODE <1 or 2 or 3> Enter value as <0x1, 0x2, 0x4>
        MAPLE_RESET_MODE <0x0,0x2,0x4,0x8,0x6,0xA,0xC,0xE>
        Debug_print <0x0,0x1>
        HW_SEM_NUM <0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7>
        Number of Shared images <0x0,0x1,0x2,0x3,0x4>

        4 0 1 1 0
```

```
        Is it B4420 ? <0x0, 0x1>
        0

NOTE: SDOS images should be located in /ipc/ directory, with the names as
      c0.bin, c1.bin, c2.bin, c3.bin, c4.bin, c5.bin.
      L1d feature can also be used in others MODE-1 and MODE-2
      Shared images are valid for MODE-3 only.
```

l1d Test Application (Multi RAT)

```
0. Since SDOS cannot be reloaded, reboot the system to load multi rat kernel modules.

1. insmod /usr/driver/IPC/multi_rat/hetmgr.ko
     Following modules params can be passed to override their default
     values:
       - dsp_shared_size  (default value 0x1000000)
       - dsp_private_addr (default value 0x80000000)
       - dsp_private_size (default value 0x7FF00000)
       - shared_ctrl_addr (default value 0xFFF00000)
       - shared_ctrl_size (default value 0x100000)
       - max_num_ipc_channels (default value 64)
       - max_channel_depth    (default value 16)

NOTE: Module params are optional, required only when default setting needs to be altered.

2. insmod /usr/driver/IPC/multi_rat/shm.ko

3. insmod /usr/driver/IPC/multi_rat/l1d.ko

4. Create device nodes with major number flashed while loading kernel modules
     cat /proc/devices will show the major numbers as well

     mknod /dev/fsl_shm c <major number> 0
     mknod /dev/het_mgr c <major number> 0
     mknod /dev/fsl_l1d c <major number> 0

5. Populate shmmax value in proc file system
     echo 0x10000000 > /proc/sys/kernel/shmmax

6. Command to load SDOS image

    Format
    ------
    $./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">

    $ ./dsp_bt -h 1 -c 1 -i c1.bin -c 0 -i c0.bin -c 2 -i c2.bin -c 3 -i c3.bin -c 4 -i c4.bin -c
5 -i c5.bin -s sh0.bin

             Usage: ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">
             whereas,
              <hw_sem> : 1 to 7 for use hardware semaphore
                       : 0 No hardware semaphore used
              <core_id> : 0 to 5 dsp core number
              <image_name> : dsp image name
                         -c option should be followed by -i option
              <shared_image> : Shared Image name
```

```
NOTE: If the semaphore number is specified as 0, it means semaphore is not used.
      Core number specifies the star core number, ranges 0-5.

7. Test application (l1d_app) can be started once the DSP/SDOS image
   is loaded successfully with the help of dsp loader(dsp_bt).

   Test sends an ioctl call to the kernel, inside the kernel it waits on a waitqueue,
   the ioctl returns only if an interrupt is generated by the dsp core towards the PA core via
MPIC interface.
   This interrupt sets the WSRSR register, value set in this register is returned to the proccss
invoking the ioctl
   and this register is cleared.

   Format
   ------
   L1 defense without IPC is use
   $ ./l1d_app 1 0
   Usage:
     ./l1d_app <single core/multi core> <ipc in use>
   Where as,
      0: single core
      1: multi core
      0: IPC not used
      1: IPC used

   OR
   Usage:
     ./l1d_app
   Where as,
     "./l1d_app" means <single core> <ipc not used>
     Same as "./l1d_app 0 0"

NOTE: This test requires user inputs,which are mentioned below:

     for Eg:
        Enter your choice
          0 means not in use for all Parameters
          Only Values mentioned below are valid, rest all values are invalid

        WARM_RESET_MODE <1 or 2 or 3> Enter value as <0x1, 0x2, 0x4>
        MAPLE_RESET_MODE <0x0,0x2,0x4,0x8,0x6,0xA,0xC,0xE>
        Debug_print <0x0,0x1>
        HW_SEM_NUM <0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7>
        Number of Shared images <0x0,0x1,0x2,0x3,0x4>

        4 0 1 1 1

        Is it B4420 ? <0x0, 0x1>
        0



NOTE: SDOS images should be located in /ipc/ directory, with the names as
      c0.bin, c1.bin, c2.bin, c3.bin, c4.bin, c5.bin.

      L1d feature can also be used in others MODE-1 and MODE-2.

      Shared images are valid for MODE-3 only.
```

l1d Test Application with IPC used (Single Core only Solutions)

```
1. insmod /usr/driver/IPC/single_rat/hetmgr.ko
     Following modules params can be passed to override their default
     values:
       - dsp_shared_size  (default value 0x1000000)
       - dsp_private_addr (default value 0x80000000)
       - dsp_private_size (default value 0x7FF00000)
       - shared_ctrl_addr (default value 0xFFF00000)
       - shared_ctrl_size (default value 0x100000)
       - max_num_ipc_channels (default value 64)
       - max_channel_depth    (default value 16)

NOTE: Module params are optional, required only when default setting needs to be altered.

2. insmod /usr/driver/IPC/single_rat/shm.ko

3. insmod /usr/driver/IPC/single_rat/l1d.ko

4. Create device nodes with major number flashed while loading kernel modules
     cat /proc/devices will show the major numbers as well

     mknod /dev/fsl_shm c <major number> 0
     mknod /dev/het_mgr c <major number> 0
     mknod /dev/fsl_l1d c <major number> 0

5. Populate shmmax value in proc file system
     echo 0x10000000 > /proc/sys/kernel/shmmax

6. Command to load SDOS image

    Format
    ------
    $./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">

    $ ./dsp_bt -h 1 -c 0 -i c0.bin

              Usage: ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">
              whereas,
               <hw_sem> : 1 to 7 for use hardware semaphore
                        : 0 No hardware semaphore used
               <core_id> : 0 to 5 dsp core number
               <image_name> : dsp image name
                           -c option should be followed by -i option
               <shared_image> : Shared Image name

NOTE: If the semaphore number is specified as 0, it means semaphore is not used.
      Core number specifies the star core number, ranges 0-5.

7. Run IPC tests:
         $ ./ipc_test -r 0 -i 10
         $ ./ipc_test -r 0 -i 10
         $ ./ipc_test67 -r 0 -i 10
         $ ./ipc_test67 -r 0 -i 10
         $ ./ipc_test -r 0 -i 100
```

```
NOTE: When total numbers message exchanged is 100 on any channel,
       DSP generated a watchdog interrupt, after this failures can be seen on IPC channels.

8. Recover DSP core with the help of test application (l1d_app).

     Test sends an ioctl call to the kernel, inside the kernel it waits on a waitqueue,
     the ioctl returns only if an interrupt is generated by the dsp core towards the PA core via
MPIC interface.
     This interrupt sets the WSRSR register, value set in this register is returned to the
proccss invoking the ioctl
     and this register is cleared.

   Format
   ------
   L1 defense with IPC is use
   $ ./l1d_app 0 1
   Usage:
     ./l1d_app <single core/multi core> <ipc in use>
   Where as,
      0: single core
      1: multi core
      0: IPC not used
      1: IPC used

   OR
   Usage:
     ./l1d_app
   Where as,
     "./l1d_app" means <single core> <ipc not used>
     Same as "./l1d_app 0 0"

NOTE: This test requires user inputs,which are mentioned below:

     for Eg:
        Enter your choice
          0 means not in use for all Parameters
          Only Values mentioned below are valid, rest all values are invalid

        WARM_RESET_MODE <1 or 2 or 3> Enter value as <0x1, 0x2, 0x4>
        MAPLE_RESET_MODE <0x0,0x2,0x4,0x8,0x6,0xA,0xC,0xE>
        Debug_print <0x0,0x1>
        HW_SEM_NUM <0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7>
        Number of Shared images <0x0,0x1,0x2,0x3,0x4>

        4 0 1 1 0

        Is it B4420 ? <0x0, 0x1>
        0

NOTE: SDOS images should be located in /ipc/ directory, with the names as
      c0.bin, c1.bin, c2.bin, c3.bin, c4.bin, c5.bin.

9. Kill the l1d_app when you see Warm reset success (ctrl+c)
         == DSP Booted up ==
         Still at BEGIN_WARM_RESET_OS_INIT or BEGIN_WARM_RESET_APP_INIT sleep 2 sec
         Warm reset success on core_id = 0x1
         sleep 5 sec
```

```
10. Rerun IPC tests to see that dsp core is recovered.
```

l1d Test Application with IPC used (Multi RAT)

```
0. Since SDOS cannot be reloaded, reboot the system to load multi rat kernel modules.

1. insmod /usr/driver/IPC/multi_rat/hetmgr.ko
     Following modules params can be passed to override their default
     values:
       - dsp_shared_size  (default value 0x1000000)
       - dsp_private_addr (default value 0x80000000)
       - dsp_private_size (default value 0x7FF00000)
       - shared_ctrl_addr (default value 0xFFF00000)
       - shared_ctrl_size (default value 0x100000)
       - max_num_ipc_channels (default value 64)
       - max_channel_depth    (default value 16)

NOTE: Module params are optional, required only when default setting needs to be altered.

2. insmod /usr/driver/IPC/multi_rat/shm.ko

3. insmod /usr/driver/IPC/multi_rat/l1d.ko

4. Create device nodes with major number flashed while loading kernel modules
     cat /proc/devices will show the major numbers as well

     mknod /dev/fsl_shm c <major number> 0
     mknod /dev/het_mgr c <major number> 0
     mknod /dev/fsl_l1d c <major number> 0


5. Populate shmmax value in proc file system
     echo 0x10000000 > /proc/sys/kernel/shmmax

6. Command to load SDOS image

    Format
    ------
    $./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">

    $ ./dsp_bt -h 1 -c 0 -i c0.bin

              Usage: ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">
              whereas,
               <hw_sem> : 1 to 7 for use hardware semaphore
                        : 0 No hardware semaphore used
               <core_id> : 0 to 5 dsp core number
               <image_name> : dsp image name
                          -c option should be followed by -i option
               <shared_image> : Shared Image name

NOTE: If the semaphore number is specified as 0, it means semaphore is not used.
      Core number specifies the star core number, ranges 0-5.

7. Run IPC tests:
         $ ./ipc_test -r 0 -i 10
         $ ./ipc_test -r 1 -i 10
         $ ./ipc_test67 -r 0 -i 10
```

```
         $ ./ipc_test67 -r 1 -i 10
         $ ./ipc_test -r 0 -i 100

NOTE: When total numbers message exchanged is 100 on any channel,
       DSP generated a watchdog interrupt, after this failures can be seen on IPC channels.

8. Recover DSP cores with the help of test application (l1d_app).

     Test sends an ioctl call to the kernel, inside the kernel it waits on a waitqueue,
     the ioctl returns only if an interrupt is generated by the dsp core towards the PA core via
MPIC interface.
     This interrupt sets the WSRSR register, value set in this register is returned to the
proccss invoking the ioctl
     and this register is cleared.

   Format
   ------
   L1 defense with IPC is use
   $ ./l1d_app 1 1
   Usage:
     ./l1d_app <single core/multi core> <ipc in use>
   Where as,
      0: single core
      1: multi core
      0: IPC not used
      1: IPC used

   OR
   Usage:
     ./l1d_app
   Where as,
     "./l1d_app" means <single core> <ipc not used>
     Same as "./l1d_app 0 0"

NOTE: This test requires user inputs,which are mentioned below:

     for Eg:
        Enter your choice
          0 means not in use for all Parameters
          Only Values mentioned below are valid, rest all values are invalid

        WARM_RESET_MODE <1 or 2 or 3> Enter value as <0x1, 0x2, 0x4>
        MAPLE_RESET_MODE <0x0,0x2,0x4,0x8,0x6,0xA,0xC,0xE>
        Debug_print <0x0,0x1>
        HW_SEM_NUM <0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7>
        Number of Shared images <0x0,0x1,0x2,0x3,0x4>

        4 0 1 1 0

        Is it B4420 ? <0x0, 0x1>
        0

        NR_DSP_CORE <0x2,0x6>
        2

NOTE: SDOS images should be located in /ipc/ directory, with the names as
      c0.bin, c1.bin, c2.bin, c3.bin, c4.bin, c5.bin.

9. Kill the l1d_app when you see Warm reset success (ctrl+c)
         == DSP Booted up ==
```

```
          Still at BEGIN_WARM_RESET_OS_INIT or BEGIN_WARM_RESET_APP_INIT sleep 2 sec
          Warm reset success on core_id = 0x1
          sleep 5 sec

10. Rerun IPC tests to see that dsp cores are recovered.
```

Hardware Watchpoint (For Multi RAT and Single RAT)

```
A hardware watchpoint register's primary task is to raise an exception, when the monitored
location is accessed.

After the reboot process is completed, initial Watchpoint that was configured in the DSP/Star
core's DTU gets deleted.

1. insmod /usr/driver/IPC/multi_rat/hetmgr.ko
     Following modules params can be passed to override their default
     values:
       - dsp_shared_size  (default value 0x1000000)
       - dsp_private_addr (default value 0x80000000)
       - dsp_private_size (default value 0x7FF00000)
       - shared_ctrl_addr (default value 0xFFF00000)
       - shared_ctrl_size (default value 0x100000)
       - max_num_ipc_channels (default value 64)
       - max_channel_depth    (default value 16)

NOTE: Module params are optional, required only when default setting needs to be altered.

2. insmod /usr/driver/IPC/multi_rat/shm.ko

3. insmod /usr/driver/IPC/multi_rat/l1d.ko

4. Create device nodes with major number flashed while loading kernel modules
     cat /proc/devices will show the major numbers as well

     mknod /dev/fsl_shm c <major number> 0
     mknod /dev/het_mgr c <major number> 0
     mknod /dev/fsl_l1d c <major number> 0


5. Populate shmmax value in proc file system
     echo 0x10000000 > /proc/sys/kernel/shmmax

6. Command to load SDOS image

    Format
    ------
    $./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">

    $ ./dsp_bt -h 1 -c 0 -i c0.bin

              Usage: ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image">
              whereas,
               <hw_sem> : 1 to 7 for use hardware semaphore
                        : 0 No hardware semaphore used
               <core_id> : 0 to 5 dsp core number
               <image_name> : dsp image name
                          -c option should be followed by -i option
               <shared_image> : Shared Image name
```

```
NOTE: If the semaphore number is specified as 0, it means semaphore is not used.
      Core number specifies the star core number, ranges 0-5.

7. Run l1d_app test

8. Add Watchpoint(wpt) in the l1d_app test

NOTE: This test requires user inputs,which are exemplified below:

   Format
   ------
   configure watchpoint ? (0/1)
   1
   Core ID (0-5), Begin_address (32-bit), End address (32-bit), type(w/r/b)

   0 0xfff00000 0xffffffff b
   Add More watchpoint ? (0/1)
   1
   Core ID (0-5), Begin_address (32-bit), End address (32-bit), type(w/r/b)

   1 0xfff003bc 0xfff003c0 b
   Add More watchpoint ? (0/1)
   0

NOTE:
    w stands for write access
    r stands for read access
    b stands for both, read and write

NOTE: SDOS images should be located in /ipc/ directory, with the names as
      c0.bin, c1.bin, c2.bin, c3.bin, c4.bin, c5.bin.
```

L1d and IPC with New Memory Map (For Multi RAT and Single RAT)

```
A New Memory map is created to provide Linux/PA with 4GB of DDR and SC with 2GB of DDR.
  - SDOS can now run in 4GB to 6GB Address space.
  - Prevents Star core from going into debug state.
  - MMU configuration is configured at run time.
  - DDRC target ID can now be configured with dsp_bt.

1. insmod /usr/driver/IPC/multi_rat/hetmgr.ko shared_ctrl_addr=0x17ff00000
dsp_private_addr=0x100000000
     Following modules params can be passed to override their default
     values:
       - dsp_shared_size  (default value 0x1000000)
       - dsp_private_addr (default value 0x80000000)
       - dsp_private_size (default value 0x7FF00000)
       - shared_ctrl_addr (default value 0xFFF00000)
       - shared_ctrl_size (default value 0x100000)
       - max_num_ipc_channels (default value 64)
       - max_channel_depth    (default value 16)

NOTE: Other Module params are optional, required only when default setting needs to be altered.

2. insmod /usr/driver/IPC/multi_rat/shm.ko

3. insmod /usr/driver/IPC/multi_rat/l1d.ko
```

```
4. Create device nodes with major number flashed while loading kernel modules
     cat /proc/devices will show the major numbers as well

     mknod /dev/fsl_shm c <major number> 0
     mknod /dev/het_mgr c <major number> 0
     mknod /dev/fsl_l1d c <major number> 0


5. Populate shmmax value in proc file system
     echo 0x10000000 > /proc/sys/kernel/shmmax

6. Command to load SDOS image

    Format
    ------
   $./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image"> -d <"DDR Controller ID">

    $ ./dsp_bt -h 1 -c 0 -i c0.bin -d 0x10

            Usage: ./dsp_bt -h <hw_sem> -c <core_id> -i <"image_name"> -s <"shared_image"> -d <"DDR
Controller ID">
             whereas,
              <hw_sem> : 1 to 7 for use hardware semaphore
                       : 0 No hardware semaphore used
              <core_id> : 0 to 5 dsp core number
              <image_name> : dsp image name
                         -c option should be followed by -i option
              <DDR Controller ID> : 0x10, 0x11, is an optional parameter
              <shared_image> : Shared Image name

NOTE: If the semaphore number is specified as 0, it means semaphore is not used.
      Core number specifies the star core number, ranges 0-5.

7. Run IPC tests:
         $ ./ipc_test -r 0 -i 10
         $ ./ipc_test -r 1 -i 10
         $ ./ipc_test67 -r 0 -i 10
         $ ./ipc_test67 -r 1 -i 10
         $ ./ipc_test -r 0 -i 100

NOTE: When total numbers message exchanged is 100 on any channel,
       DSP generated a watchdog interrupt, after this failures can be seen on IPC channels.

8. Recover DSP cores with the help of test application (l1d_app).

     Test sends an ioctl call to the kernel, inside the kernel it waits on a waitqueue,
     the ioctl returns only if an interrupt is generated by the dsp core towards the PA core via
MPIC interface.
     This interrupt sets the WSRSR register, value set in this register is returned to the proccss
invoking the ioctl
     and this register is cleared.

   Format
   ------
   L1 defense with IPC is use
   $ ./l1d_app 1 1
   Usage:
     ./l1d_app <single core/multi core> <ipc in use>
   Where as,
      0: single core
```

```
       1: multi core
       0: IPC not used
       1: IPC used

   OR
   Usage:
     ./l1d_app
   Where as,
     "./l1d_app" means <single core> <ipc not used>
     Same as "./l1d_app 0 0"

NOTE: This test requires user inputs,which are mentioned below:

     for Eg:
        Enter your choice
          0 means not in use for all Parameters
          Only Values mentioned below are valid, rest all values are invalid

        WARM_RESET_MODE <1 or 2 or 3> Enter value as <0x1, 0x2, 0x4>
        MAPLE_RESET_MODE <0x0,0x2,0x4,0x8,0x6,0xA,0xC,0xE>
        Debug_print <0x0,0x1>
        HW_SEM_NUM <0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7>
        Number of Shared images <0x0,0x1,0x2,0x3,0x4>

        4 0 1 1 0

        Is it B4420 ? <0x0, 0x1>
        0

        NR_DSP_CORE <0x2,0x6>
        6
    configure watchpoint ? (0/1)
    0

    Use Default DSP safe virtual address ? (0/1)
    1

    DSP safe virtual address used are:
    0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000

    Run it for infinite loop? (y/n)
    y

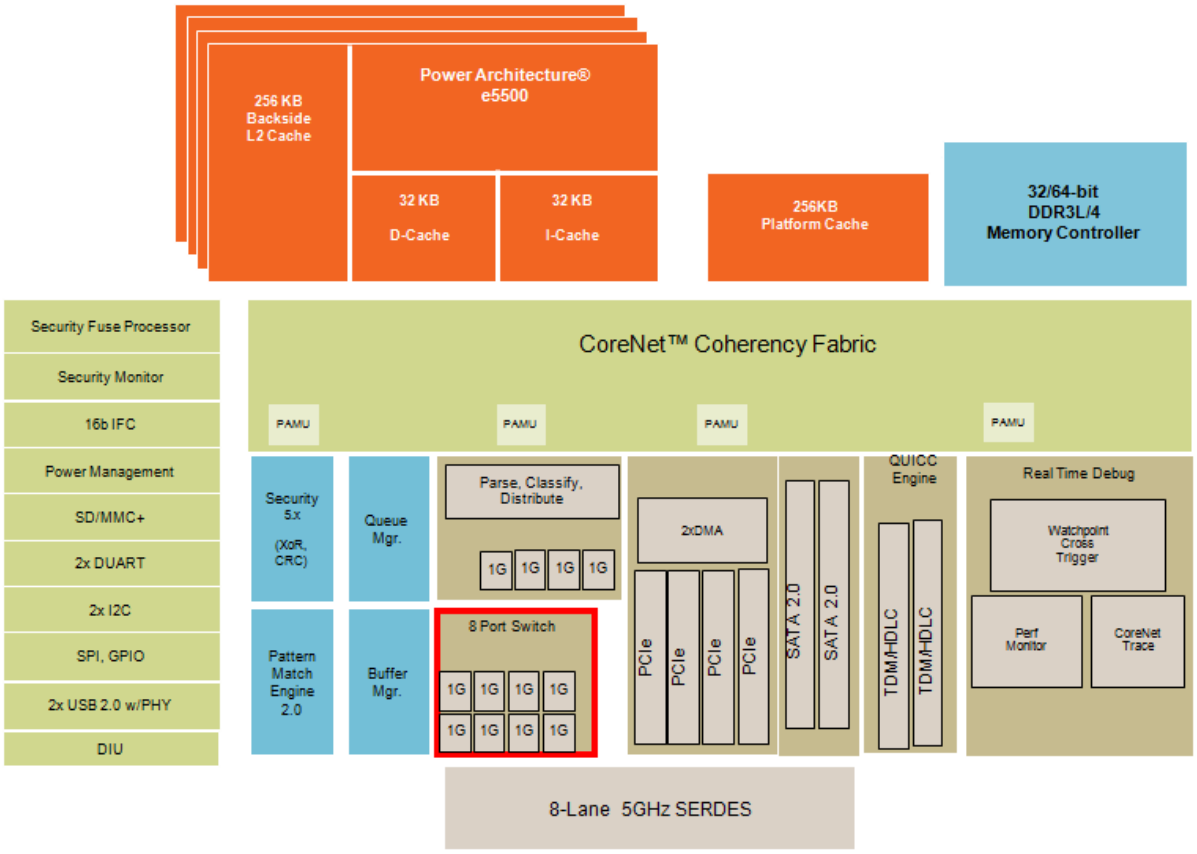
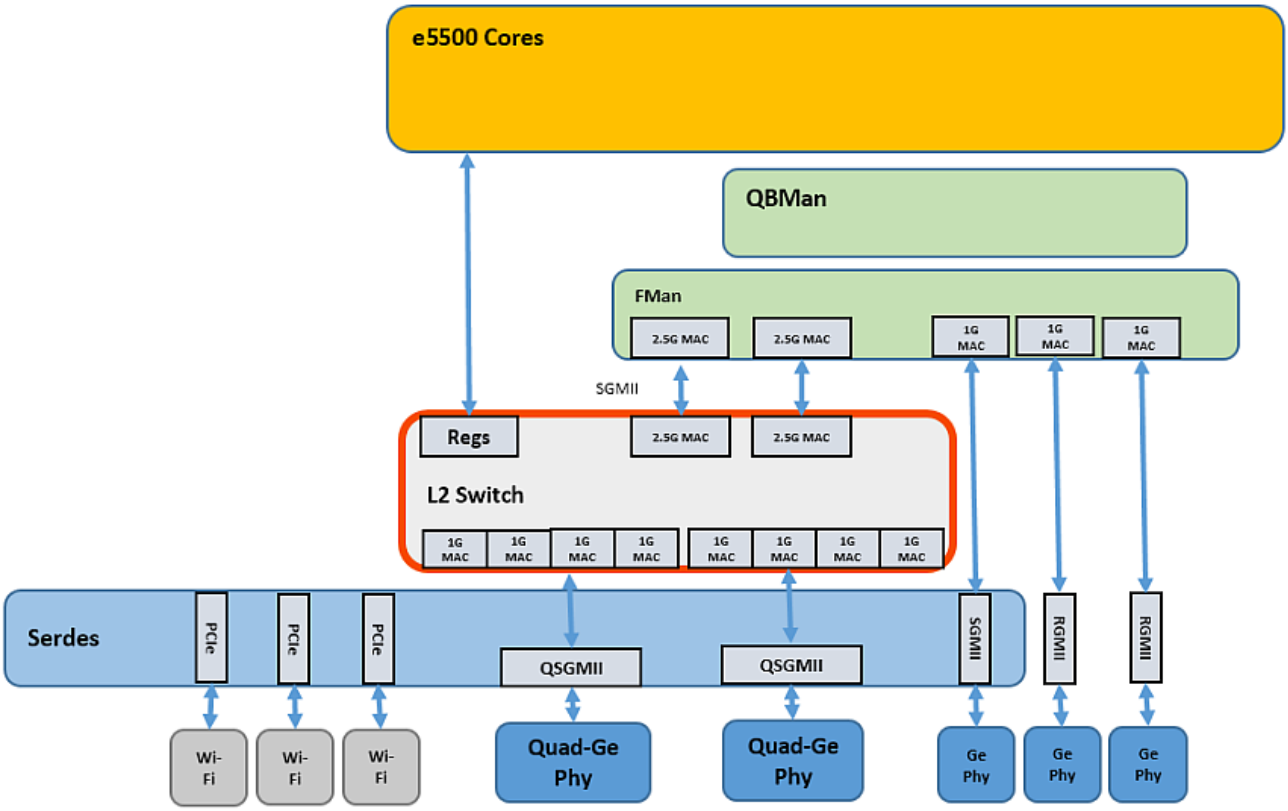
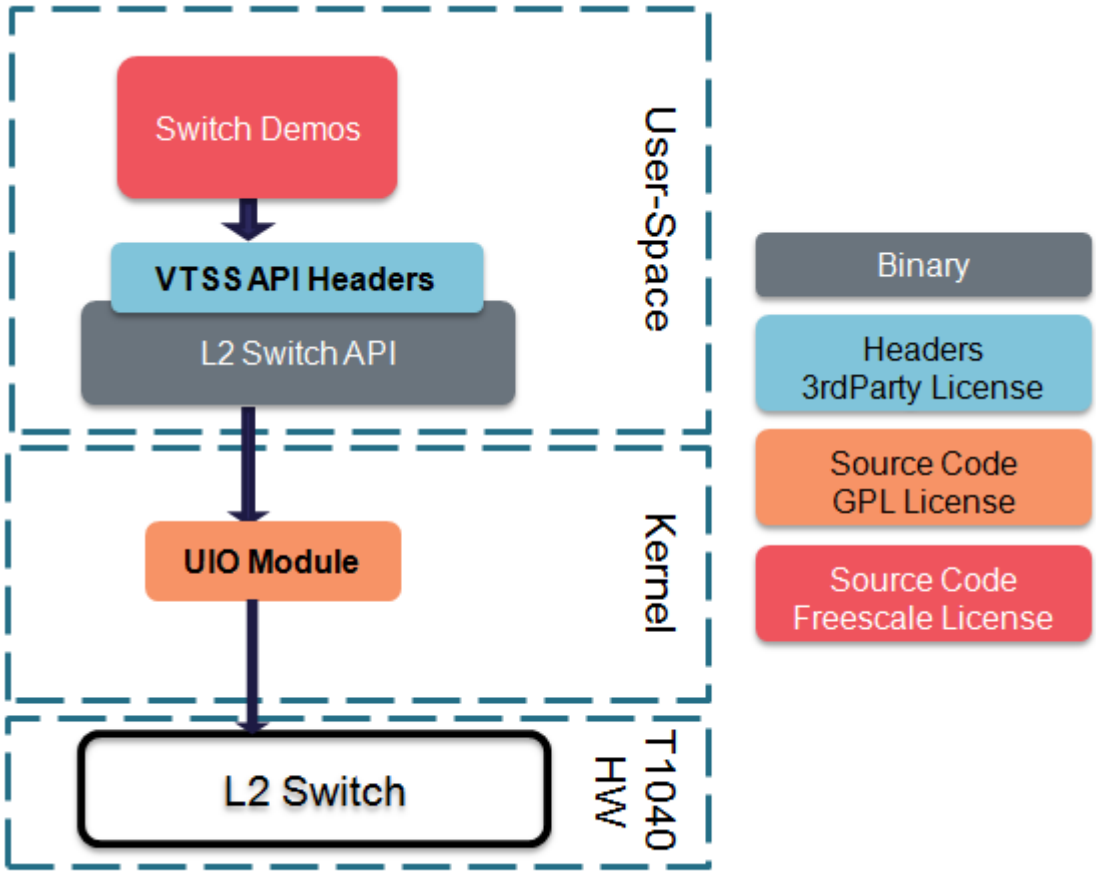

NOTE: SDOS images should be located in /ipc/ directory, with the names as
      c0.bin, c1.bin, c2.bin, c3.bin, c4.bin, c5.bin.

NOTE: Step 9 and Step 10 are reuired only for SDOS imagw which has both IPC and L1d together.

9. Kill the l1d_app when you see Warm reset success (ctrl+c)
         == DSP Booted up ==
         Still at BEGIN_WARM_RESET_OS_INIT or BEGIN_WARM_RESET_APP_INIT sleep 2 sec
         Warm reset success on core_id = 0x1
         sleep 5 sec

10. Rerun IPC tests to see that dsp cores are recovered.
```

SDOS IMAGE FORMAT

```
Dsp loader(dsp_bt) expects the SDOS image in the below mentioned format.

       ENDIANESS_BYTE (1Byte)
       <ADDRESS - 8Bytes><SIZE_IN_BYTES - 8Bytes><data_payload>
       <ADDRESS - 8Bytes><SIZE_IN_BYTES - 8Bytes><data_payload>
       <ADDRESS - 8Bytes><SIZE_IN_BYTES - 8Bytes><data_payload>
       ........
       <ADDRESS =0xFFFF_FFFF_FFFF_FFFF ><SIZE = 8><__crt0_start address>

       __crt0_start address should be 8k aligned.
```

Limitations

IPC support for B4 platforms is available only for 32-bit user space.

9.4 L2Switch (T1040)

9.4.1 Overview

NXP T1040 and T1020 SoC have an 8-port gigabit Ethernet switch integrated on the device.

T1040 Hardware Support

Linux User Space
L2Switch (T1040)



Figure 210. T1040 Architecture

The integrated 8-port gigabit Ethernet switch supports the following features:

- 8192 MAC addresses
  - Static Address provisioning
  - Dynamic learning of MAC addresses and aging
- 4096 VLANs
  - Independent VLAN learning (IVL)
  - VLAN editing, translation and remarking
- Link aggregation (IEEE Std 802.3ad)
- Policing with storm control and MC/BC protection
- Spanning Tree Protocol (STP, RSTP and MSTP) - IEEE 802.1w/IEEE 802.1s
- IPv4 and IPv6 multicast
- Jumbo frames (9.6 KB)
- Access Control List (ACLs)
- Hierarchical QoS with DWRR scheduling at different levels between the following:
  - Ingress ports
  - Priorities

- Services
- RMON counters per port

Switch interfaces:

- 8 Gigabit switch ports are external ports and are connected to external Phys
- 2 switch ports (2.5G) are connected to FMan ports



Figure 211. T1040 Port Connectivity

For more details check the T1040 Reference Manual.

Software Support

The Software Architecture of the NXP SDK enablement software is:



Figure 212. SW Architecture

NXP SDK includes an L2 Switch user space driver and a small demo application that uses the API provided by the switch driver.

L2 Switch protocol termination stack is not included in NXP SDK and can be provided through external means.

Features Supported

Functionalities supported by L2Switch API:

- Forwarding
  - Dynamic MAC Learning
  - Static MAC Forwarding
- Port Control & Statistics
- Link Aggregation
  - Static Link Aggregation
- VLAN
  - VLAN Configuration
  - Port Based VLAN
  - Private VLAN
  - Protocol based VLAN

- IP Subnet based VLAN
- MAC based VLAN
- Q-in-Q (IEEE-802.1ad Provider Bridge)
- Voice VLAN
- Port isolation

- BPDU Handling
- QoS
  - Classification
  - Policing
  - Scheduling and Shaping
  - Tag Remarking
  - DSCP based QoS (e.g. classification/translation, remarking)
  - PFC (Priority Based Flow Control IEEE 802.1qbb)
  - WRED
- 802.3x Flow control
- ACLs
  - ACE Configuration
  - ACL Port Configuration
- Port Mirroring
- Storm control
- 802.1X
  - Port-based 802.1X
  - Single 802.1X
  - Multiple 802.1X
- Port Mirroring
- sFlow

Functionalities supported by Demo Application

- Port Control
- Port Statistics
- MAC address table operations
- Configuration options for:
  - Static Link Aggregation
  - VLANs
  - VLAN Classification Entries
  - Private VLANs
  - Isolated ports
  - Storm control
  - Port Mirroring

- QoS
- Q-in-Q (IEEE-802.1ad Provider Bridge)
- Flow control

9.4.2 Build and Installation Summary

The driver and demo application is included by default in the rootfs generated by the NXP SDK. Demo application is configured to start at boot.

L2Switch components can also be built and installed manually.

Building L2Switch Driver with Yocto

The L2Switch driver can be built from a Yocto recipe included in NXP SDK:

```
> bitbake uio-seville
```

This will build and package a kernel module containing the driver.

Building L2Switch Demo Application with Yocto

The L2Switch demo application is a simple sample application that exercises the API to control the switch. It offers basic commands, like port control, port aggregation and port statistics information.

The L2Switch demo application can be built from a Yocto recipe included in NXP SDK:

```
> bitbake l2switch
```

This will build several packages containing the demo application, the shared library with the user-space driver and the API headers.

Installation

The packages obtained in the previous steps have to be installed on the rootfs:

1. Install the kernel module package and insert the module into the kernel:

   ```
   root@t1040rdb:~# insmod /lib/modules/`uname -r`/extra/uio_seville.ko
   ```

   If the insertion is successful the console will display this message:

   ```
   seville ffe800000.l2switch: Found Seville Switch, UIO device - IRQ 26, id 0x099530e9.
   ```

2. Install the shared object user-space driver package:

   ```
   root@t1040d4rdb:~# rpm –i libvtss-api0-0.1-r0.ppc64e5500.rpm
   ```

3. Install the L2Switch demo application:

   ```
   root@t1040d4rdb:~# rpm –i l2switch-0.1-r0.ppc64e5500.rpm
   ```

The switch demo application is started with the following command:

```
root@t1040rdb:~# /etc/init.d/l2switch start
```

This allows also stopping the application and restarting it.

```
root@t1040rdb:~# /etc/init.d/l2switch {start|stop|restart|reload|status}
```

Demo application default configuration:

- L2 switch is started with all ports enabled
- Internal ports are configured as fixed links at 2.5Gbps
- External ports have autonegotiation at 1Gbps
- Ports are in VLAN 1
- Maximum frame size is 10k
- Switch performs hardware MAC learning
- Flow Control is disabled by default

NOTE: Some L2 protocols require a MAC address for each switch port for proper functioning. An address can be specified in u-boot, that will serve as a seed for generating MAC addresses for each port:

```
setenv l2switchaddr 00:11:22:33:44:55
saveenv
```

This address will be available as a sysfs entry:

```
root@t1040d4rdb:~# cat /sys/bus/platform/devices/ffe800000.l2switch/mac_address
00:11:22:33:44:55
```

9.4.3 l2switch-cfg Command Reference

This section details the list of commands and parameters supported by the L2Switch demo application

Interaction with the L2 Switch is possible via l2switch-cfg script:

```
root@t1040d4rdb:~# l2switch-cfg
Usage: /usr/bin/l2switch-cfg [options] - basic script to configure the L2 switch

Options:
dscp <option>                  - Configure DSCP mode
                                 where option can be translate|remap|trust
isolated member(s) <option>    - Configure isolated members list
mac <option>                   - Configure MAC address table
port <option>                  - Configure a specific port where option
                                 can be stats|all|port_no
private vlan <option>          - add/remove port from/to private VLANs or
                                 show private VLANs
qos <option>                   - Configure QoS
set <option>                   - Set mirroring or max frame length or lag mode
                                 where option can be mirror|max_frame_length|
```

```
                                  lag
storm control <option>          - Set storm control rate or disable
vlan <option>                   - VLAN configuration:
                                  - add/remove port to/from VLAN
                                  - show VLANs
                                  - configure VLAN translation
                                  - show isolated VLANs
vce <add|remove> <option>       - VCE configuration(MAC,OUI,Type,Subnet)
control traffic <option>        - Control traffic configuration where option can
                                  be conf|stats|verbose|receive|reflect|ena|dis
show version                    - shows software version
poll <on|off>                   - enable/disable PHY polling
check                           - check for dropped or invalid frames;
                                  if any, a debug file will be generated
```

Note: In order to be able to use link aggregation on the internal ports (8 and 9) the bonding module has to be inserted in the kernel. See Appendix - Bonding configuration for internal ports on page 1405

9.4.4 U-Boot Command Reference

U-Boot is able to initialize the switch. The following message is displayed in logs:

```
VSC9953 L2 switch initialized
```

This enables using switch ports in unmanaged mode for access to network (TFTP file transfer, connectivity checks). FM1@DTSEC1 and FM1@DTSEC2 are connected to switch port 8 and 9 at 2.5Gbps.

Note: Switch does not have any loop protection mechanism running, so the user must take care not to create any network loops.

Switch configuration is maintained in Linux, if demo application is not loaded.

Limited configuration commands are available in U-Boot:

```
=> ethsw
ethsw - vsc9953 l2 switch commands

Usage:
ethsw port <port_nr> enable|disable
    - enable/disable an l2 switch port
      port_nr=0..9; use "all" for all ports
ethsw port <port_nr> show
    - show an l2 switch port's configuration
      port_nr=0..9; use "all" for all ports
```

Example: displaying L2 switch ports status

```
=> ethsw port all show
    Port    Status     Link    Speed   Duplex
       0  enabled       up     1000     full
       1  enabled       up     1000     full
       2  enabled       up     1000     full
       3  enabled     down       10     half
       4  enabled     down       10     half
       5  enabled     down       10     half
       6  enabled       up     1000     full
       7  enabled       up     1000     full
       8  enabled       up     2500     full
       9  enabled       up     2500     full
```

9.4.5 l2switch-cfg How To's

9.4.5.1 How to control Ports and Statistics

Port control:

```
root@t1040d4rdb:/# l2switch-cfg port <port_no> state enable
root@t1040d4rdb:/# l2switch-cfg port <port_no> state disable
```

Port status:

```
root@t1040d4rdb:/# l2switch-cfg port all config show
Port 0 state: enabled  link: up        mode: auto      speed: 1G       duplex: full
Port 1  state: enabled  link: up        mode: auto      speed: 1G       duplex: full
Port 2  state: enabled  link: up        mode: auto      speed: 1G       duplex: full
Port 3  state: enabled  link: up        mode: auto      speed: 1G       duplex: full
Port 4  state: enabled  link: down      mode: auto      speed: ???      duplex: half
Port 5  state: enabled  link: down      mode: auto      speed: ???      duplex: half
Port 6  state: enabled  link: down      mode: auto      speed: ???      duplex: half
Port 7  state: enabled  link: down      mode: auto      speed: ???      duplex: half
Port 8  state: enabled  link: down      mode: static    speed: 2.5G     duplex: full
Port 9  state: enabled  link: down      mode: static    speed: 2.5G     duplex: full
```

Port statistics:

```
root@t1040d4rdb:~# l2switch-cfg port stats all clear

root@t1040d4rdb:~# l2switch-cfg port stats all packets
Port 0 statistics:
---------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
---------------------------------------------------------
Port 1 statistics:
---------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
---------------------------------------------------------
Port 2 statistics:
---------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
---------------------------------------------------------
Port 3 statistics:
---------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
---------------------------------------------------------
Port 4 statistics:
---------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
---------------------------------------------------------
Port 5 statistics:
---------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
---------------------------------------------------------
Port 6 statistics:
---------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
```

```
-------------------------------------------------------
Port 7 statistics:
-------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
-------------------------------------------------------
Port 8 statistics:
-------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
-------------------------------------------------------
Port 9 statistics:
-------------------------------------------------------
Rx Packets:          0   Tx Packets:          0
-------------------------------------------------------
```

9.4.5.2 How to control Link Aggregation

Commands for setting LAG between two ports:

```
root@t1040d4rdb:/# l2switch-cfg set lag mode
root@t1040d4rdb:/# l2switch-cfg set lag no 1 members 5 6
       LAG NO 1 members: 5 6
```

9.4.5.3 How to configure port based VLAN

- By default all ports belong to VLAN 1:

```
root@t1040d4rdb:~# l2switch-cfg vlan show
Printing all VLANs...

VLAN 1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

- Adding a port to a specific VLAN:

```
root@t1040d4rdb:~# l2switch-cfg vlan 10 add 3
Adding port 3 as member for VLAN 10
VLAN 10: 3,

root@t1040d4rdb:~# l2switch-cfg vlan 11 add 4
Adding port 4 as member for VLAN 11
VLAN 11: 4,

root@t1040d4rdb:~# l2switch-cfg vlan show
Printing all VLANs...

VLAN 1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
VLAN 10: 3,
VLAN 11: 4,
```

- Removing a port from a specific VLAN:

```
root@t1040d4rdb:~# l2switch-cfg vlan 10 remove 3
Removing port 3 from VLAN 10 members

root@t1040d4rdb:~# l2switch-cfg vlan show
Printing all VLANs...
```

```
VLAN 1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
VLAN 11: 4,
```

9.4.5.4 How to control MAC table

Commands available for MAC table operations:

```
mac add <mac> <port< [<vid>]           - add static entry to FDB
mac del <mac> [<vid<]                  - delete static entry from FDB
mac lookup <mac> [<vid>]               - lookup for specficic entry in FDB
mac dump                               - dump FDB entries content
mac flush [port <port>] [vlan <vid>]   - flush dynamic entries
mac agetime [<age_time>]               - get/set agetime
mac learn <auto|off>                   - enable/disable hardware MAC learning
```

1. Adding a static entry:

```
root@t1040d4rdb:~# l2switch-cfg mac add 00:11:22:33:44:55 4 12

root@t1040d4rdb:~# l2switch-cfg mac dump
Type    VID  MAC Address        Ports
------  ---  -----------------  -----
Static  12   00:11:22:33:44:55  4

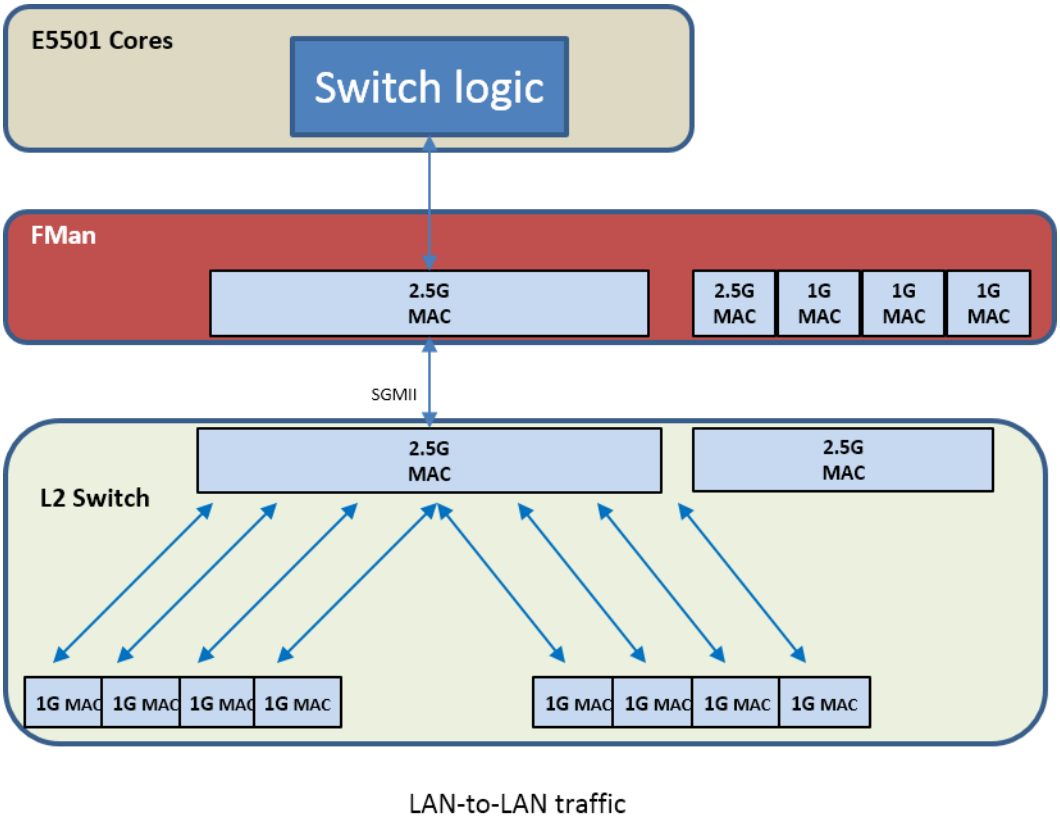
Static  entries: 1
Dynamic entries: 0
```

2. Deleting a static entry:

```
root@t1040d4rdb:~# l2switch-cfg mac del 00:11:22:33:44:55

root@t1040d4rdb:~# l2switch-cfg mac dump
Type    VID  MAC Address        Ports
------  ---  -----------------  -----

Static  entries: 0
Dynamic entries: 0
```

3. lookup for specific MAC address:

```
root@t1040d4rdb:~# l2switch-cfg mac dump
Type    VID  MAC Address        Ports
------  ---  -----------------  -----
Static  1    00:11:22:33:44:66  5
Static  1    00:11:22:33:44:67  5
Static  1    00:11:22:33:44:68  5
Static  10   00:11:22:33:44:60  3

Static  entries: 4
Dynamic entries: 0

root@t1040d4rdb:~# l2switch-cfg mac lookup 00:11:22:33:44:67
```

```
Type    VID  MAC Address        Ports
------  ---  -----------------  -----
Static  1    00:11:22:33:44:67  5

root@t1040d4rdb:~# l2switch-cfg mac lookup 00:11:22:33:44:50
No entries found
```

9.4.5.5 How to check for errors

Command to gather extra info in case of dropped/invalid frames. This is a diagnose command that indicates if frames are dropped by the switch. In case of dropped frames, errors are showed on the console and a file is created that gathers additional information.

```
root@t1040d4rdb:~# l2switch-cfg check
No dropped or invalid frames reported
```

9.4.5.6 How to control PHY polling

If no interrupts are present in device-tree for Seville PHYs, by default application polls for PHY status. Deactivating PHY polling is useful mainly for benchmarking scenarios where l2switch-cfg needs to consume 0 CPU cycles.

```
root@t1040d4rdb:~# l2switch-cfg poll on
polling is already turned on
```

To disable PHY polling:

```
root@t1040d4rdb:~# l2switch-cfg poll off
polling disabled
```

9.4.6 Use Cases

9.4.6.1 Using L2Switch as Port Extender

The LAN ports of the L2 Switch make forwarding decisions based only on L2 switch logic, with no involvement from FMAN or CPU (CPU can't track the packets that are switched between the 8 external L2 Switch ports). Depending on the use case, this behavior might not be desirable. In order to direct all traffic received on external ports to CPU, maintaining ingress port information and avoid switching in L2Switch IP the Port-based VLAN functionality needs to be used.



Figure 213. Port extender

Note: In the first example only one L2 Switch internal port (port 8) is used, ensuring a throughput of 2.5 Gbps, full duplex mode. The FMAN port corresponding to the second internal port (port 9) won't be used.

If the throughput is not enough, the second internal port (port 9) is required for increasing the throughput to 5 Gbps, full duplex.

Configuration when using one FMan Port

The following commands will add each external port <port_nr> as a member of VLAN 10 + <nr_port> to create a separate VLAN for each external port:

```
root@t1040d4rdb:~# l2switch-cfg vlan 10 add 0
Adding port 0 as member for VLAN 10
VLAN 10: 0,

root@t1040d4rdb:~# l2switch-cfg vlan 11 add 1
Adding port 1 as member for VLAN 11
VLAN 11: 1,

root@t1040d4rdb:~# l2switch-cfg vlan 12 add 2
Adding port 2 as member for VLAN 12
VLAN 12: 2,

root@t1040d4rdb:~# l2switch-cfg vlan 13 add 3
Adding port 3 as member for VLAN 13
VLAN 13: 3,

root@t1040d4rdb:~# l2switch-cfg vlan 14 add 4
```

```
Adding port 4 as member for VLAN 14
VLAN 14: 4,

root@t1040d4rdb:~# l2switch-cfg vlan 15 add 5
Adding port 5 as member for VLAN 15
VLAN 15: 5,

root@t1040d4rdb:~# l2switch-cfg vlan 16 add 6
Adding port 6 as member for VLAN 16
VLAN 16: 6,

root@t1040d4rdb:~# l2switch-cfg vlan 17 add 7
Adding port 7 as member for VLAN 17
VLAN 17: 7,
```

To add the internal port 8 as a member to all the previous created VLANs use the following commands:

```
root@t1040d4rdb:~# l2switch-cfg vlan 10 add 8
Adding port 8 as member for VLAN 10
VLAN 10: 0, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 11 add 8
Adding port 8 as member for VLAN 11
VLAN 11: 1, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 12 add 8
Adding port 8 as member for VLAN 12
VLAN 12: 2, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 13 add 8
Adding port 8 as member for VLAN 13
VLAN 13: 3, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 14 add 8
Adding port 8 as member for VLAN 14
VLAN 14: 4, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 15 add 8
Adding port 8 as member for VLAN 15
VLAN 15: 5, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 16 add 8
Adding port 8 as member for VLAN 16
VLAN 16: 6, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 17 add 8
Adding port 8 as member for VLAN 17
VLAN 17: 7, 8,
```

The Rx traffic for the L2 Switch external ports has to be classified and tagged with the proper VLAN, so that the application which will make the forwarding decision will be aware of the packet's ingress port by looking at the VID from the VLAN tag.

This can be done using Port-based VLAN L2 Switch functionality:

```
root@t1040d4rdb:~# l2switch-cfg port 0 vlan 10
0 - VID 10
```

```
root@t1040d4rdb:~# l2switch-cfg port 1 vlan 11
1 - VID 11

root@t1040d4rdb:~# l2switch-cfg port 2 vlan 12
2 - VID 12

root@t1040d4rdb:~# l2switch-cfg port 3 vlan 13
3 - VID 13

root@t1040d4rdb:~# l2switch-cfg port 4 vlan 14
4 - VID 14

root@t1040d4rdb:~# l2switch-cfg port 5 vlan 15
5 - VID 15

root@t1040d4rdb:~# l2switch-cfg port 6 vlan 16
6 - VID 16

root@t1040d4rdb:~# l2switch-cfg port 7 vlan 17
7 - VID 17
```

With the above commands ingress traffic on port <port_nr> is classified to VLAN 10 + <port_nr>. This traffic will be sent to internal port since there is no other member of the classified VLAN to forward the packet to. Notice that the default Port-based VLAN was not changed for port 8 (VLAN 1). This means that packets that are classified to a VLAN different than VLAN 1 will be tagged on the egress of port 8 with the classified VLAN ID. Also, when injecting a packet from the application to internal port 8, the packet must be tagged with the VLAN corresponding to the external port on which we want to send to. The VLAN tag will be removed on the egress of the external port.

Configuration output:

```
root@t1040d4rdb:~# l2switch-cfg vlan show
Printing all VLANs...

VLAN 1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
VLAN 10: 0, 8,
VLAN 11: 1, 8,
VLAN 12: 2, 8,
VLAN 13: 3, 8,
VLAN 14: 4, 8,
VLAN 15: 5, 8,
VLAN 16: 6, 8,
VLAN 17: 7, 8,

root@t1040d4rdb:~# l2switch-cfg port all vlan show
Vlan port-based configuration (port - VID):
0 - VID 10
1 - VID 11
2 - VID 12
3 - VID 13
4 - VID 14
5 - VID 15
6 - VID 16
7 - VID 17
8 - VID 1
9 - VID 1
```

Using VLANs for creating a Port Extender scenario is straight forward, easy to configure, there is no custom header involved.

The limitations are:

- VLAN functionality is reserved to the Port Extender use case and can't be used properly for any other configuration. If a packet arrives at an external port already tagged, it will be classified to the VID specified in the VLAN tag, not to its Port-based VLAN. In a future release, the VLAN Unaware functionality of a port will be added, which will always classify a packet to the Port-based VLAN, even if the packet is already tagged with a VLAN;
- Throughput of all 8 external ports can't be higher than 2.5 Gbps;
- One L2 switch port (port 9) and one FMAN internal port unused;

Configuration when using two FMan Ports

In the previous configuration, only one L2 Switch internal port has been used and the throughput to CPU is limited to 2.5 Gbps.

Using both internal ports the throughput is doubled and in order to do this, the traffic needs to be split on the external ports.

This may be done by grouping external ports 0…3 with internal port 8 and external ports 4..7 with internal port 9.

The commands for this scenario are:

```
root@t1040d4rdb:~# l2switch-cfg vlan 10 add 8
Adding port 8 as member for VLAN 10
VLAN 10: 0, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 11 add 8
Adding port 8 as member for VLAN 11
VLAN 11: 1, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 12 add 8
Adding port 8 as member for VLAN 12
VLAN 12: 2, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 13 add 8
Adding port 8 as member for VLAN 13
VLAN 13: 3, 8,

root@t1040d4rdb:~# l2switch-cfg vlan 14 add 9
Adding port 9 as member for VLAN 14
VLAN 14: 4, 9,

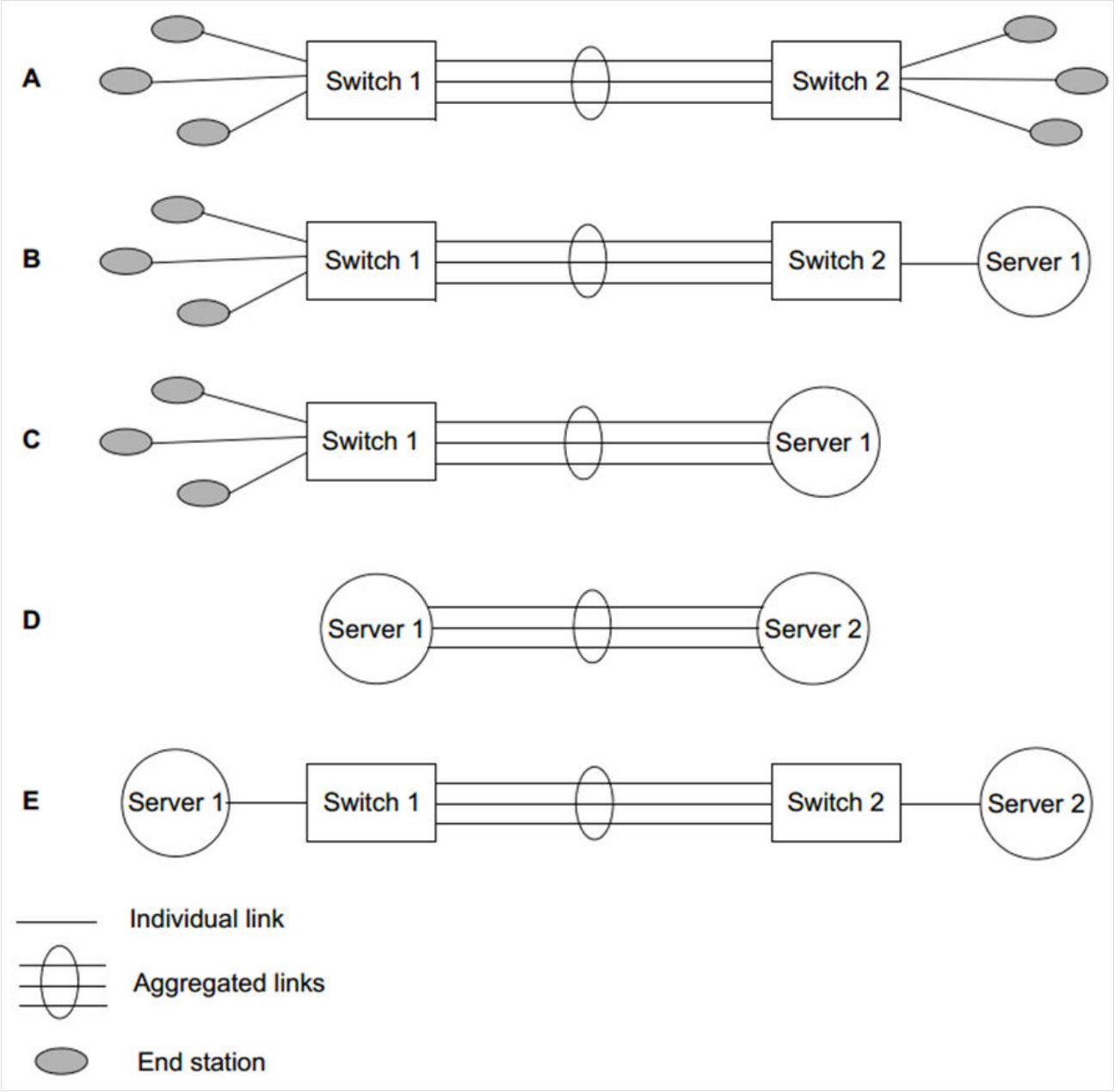
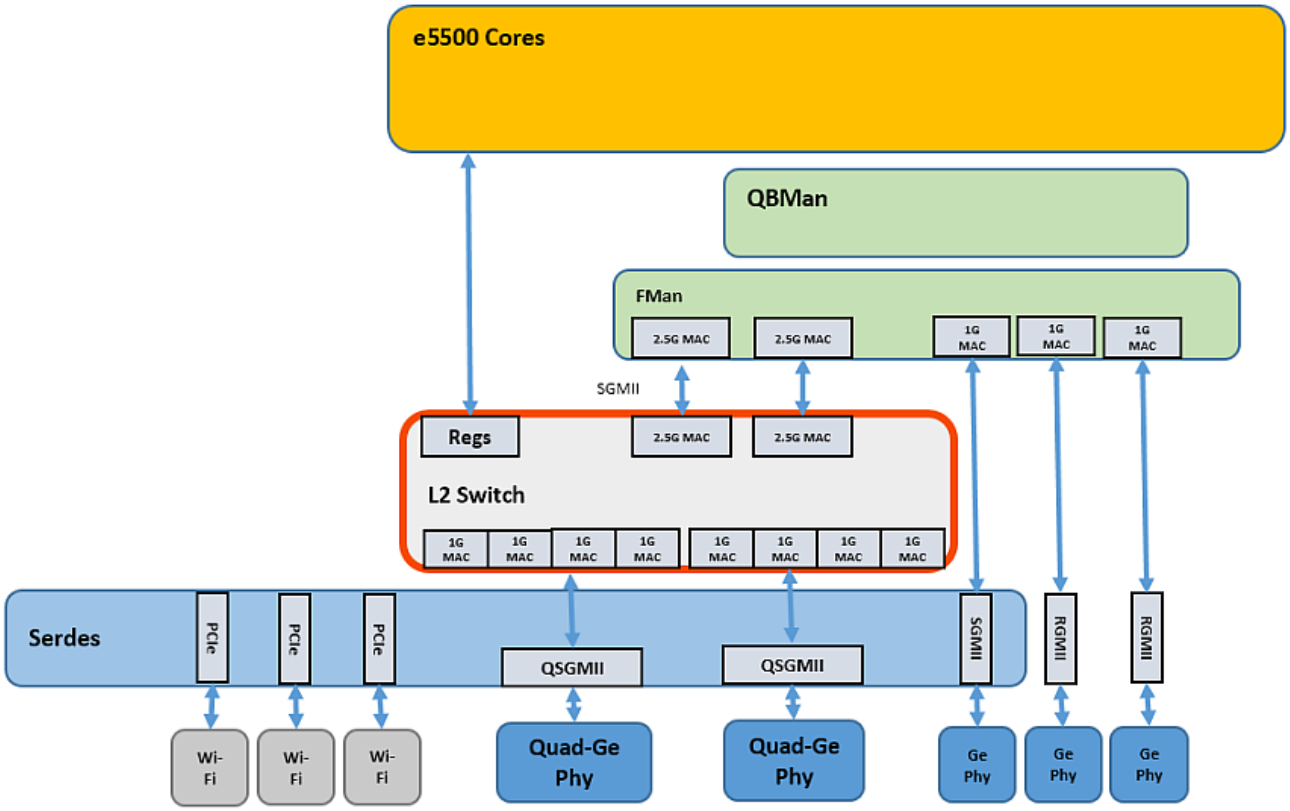
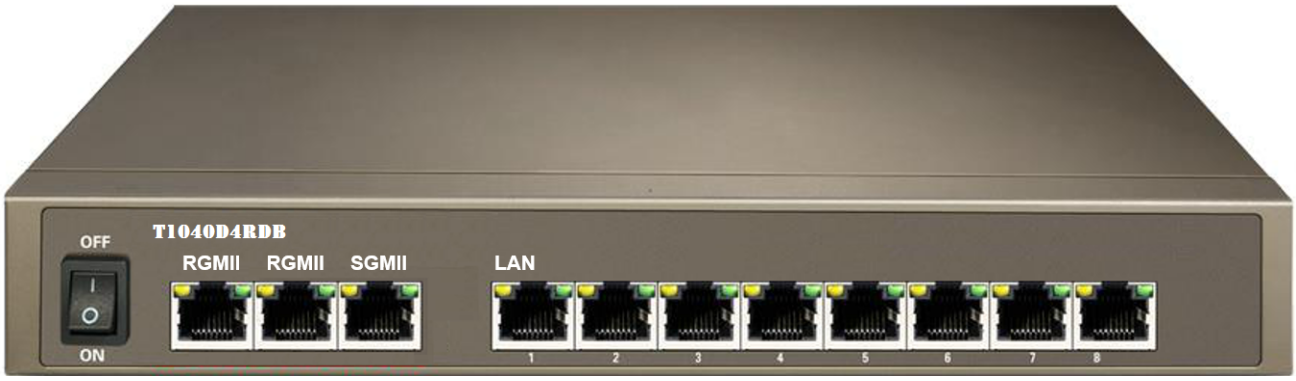
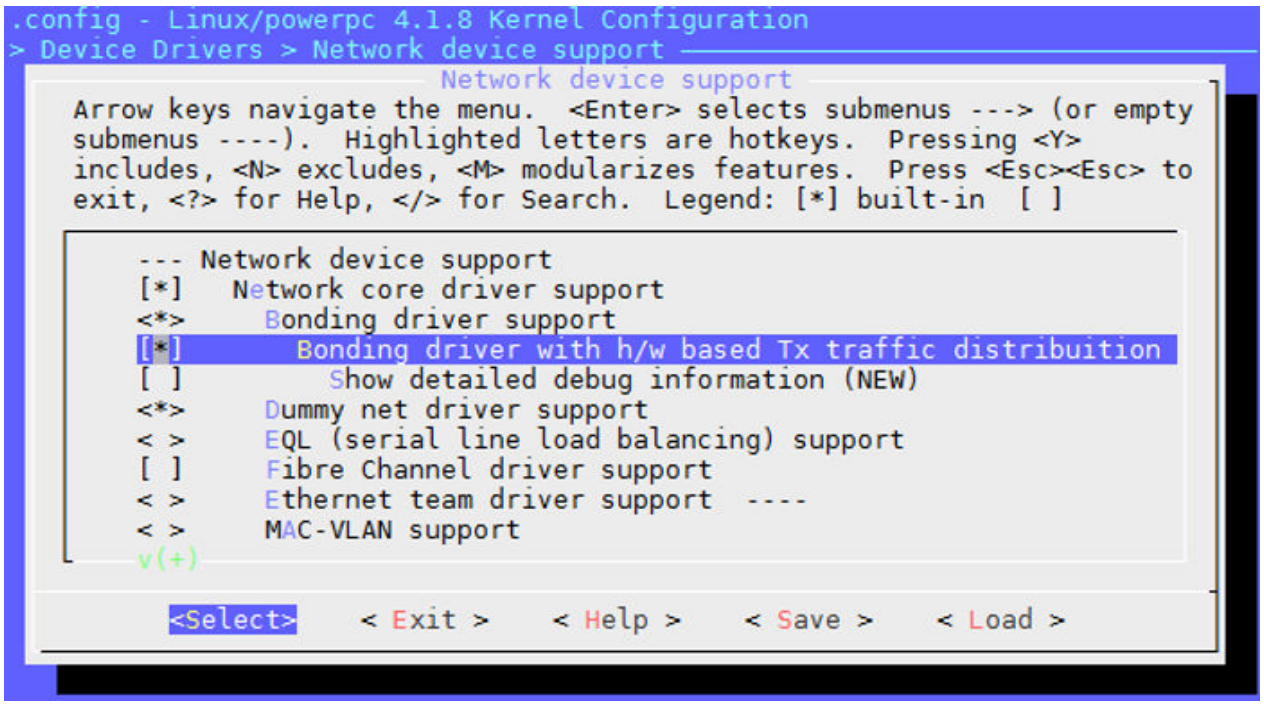
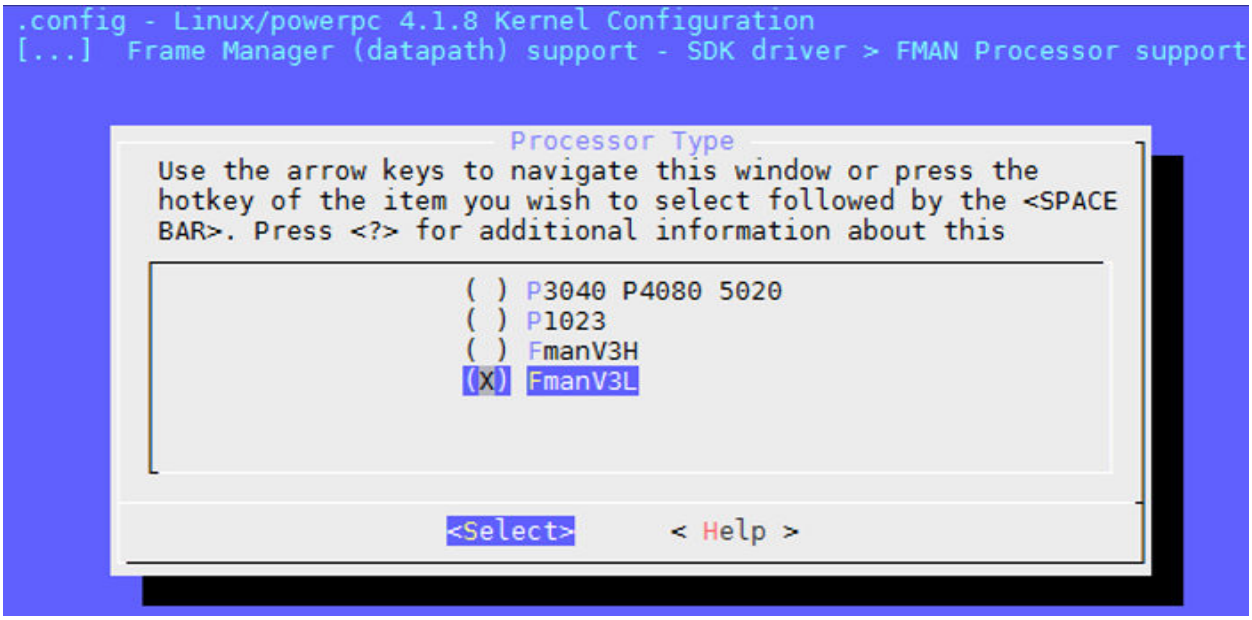
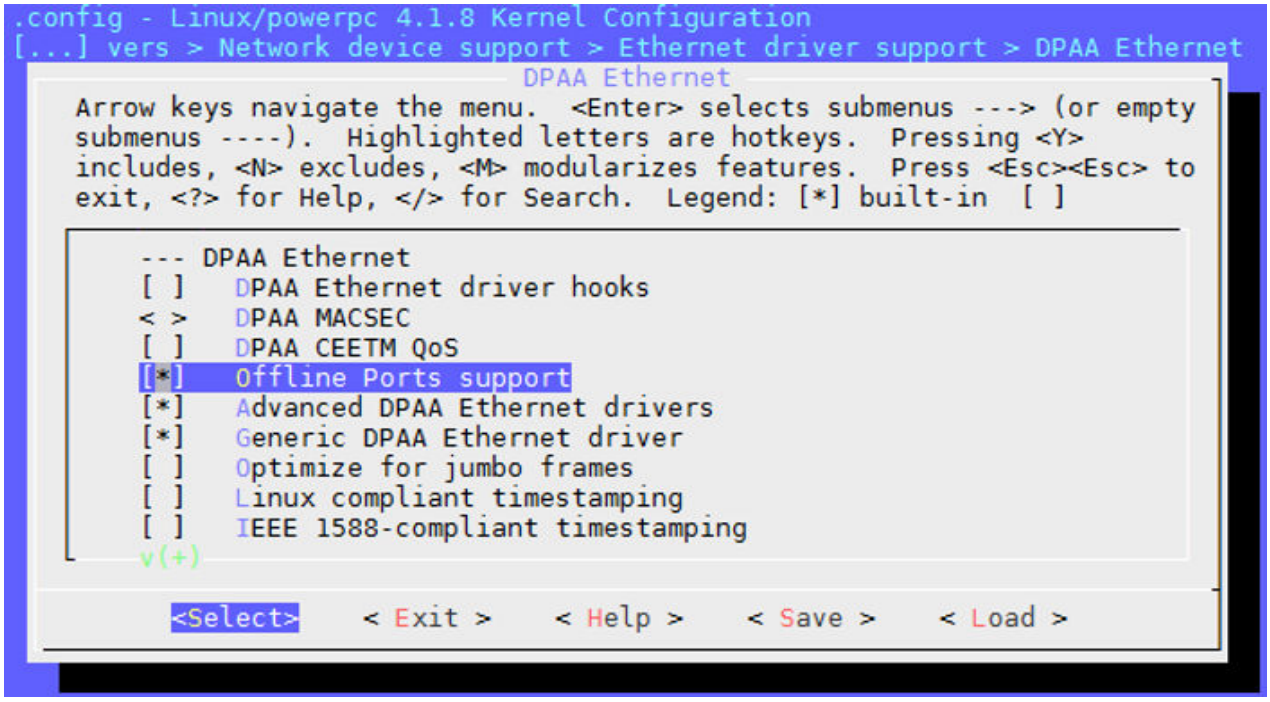
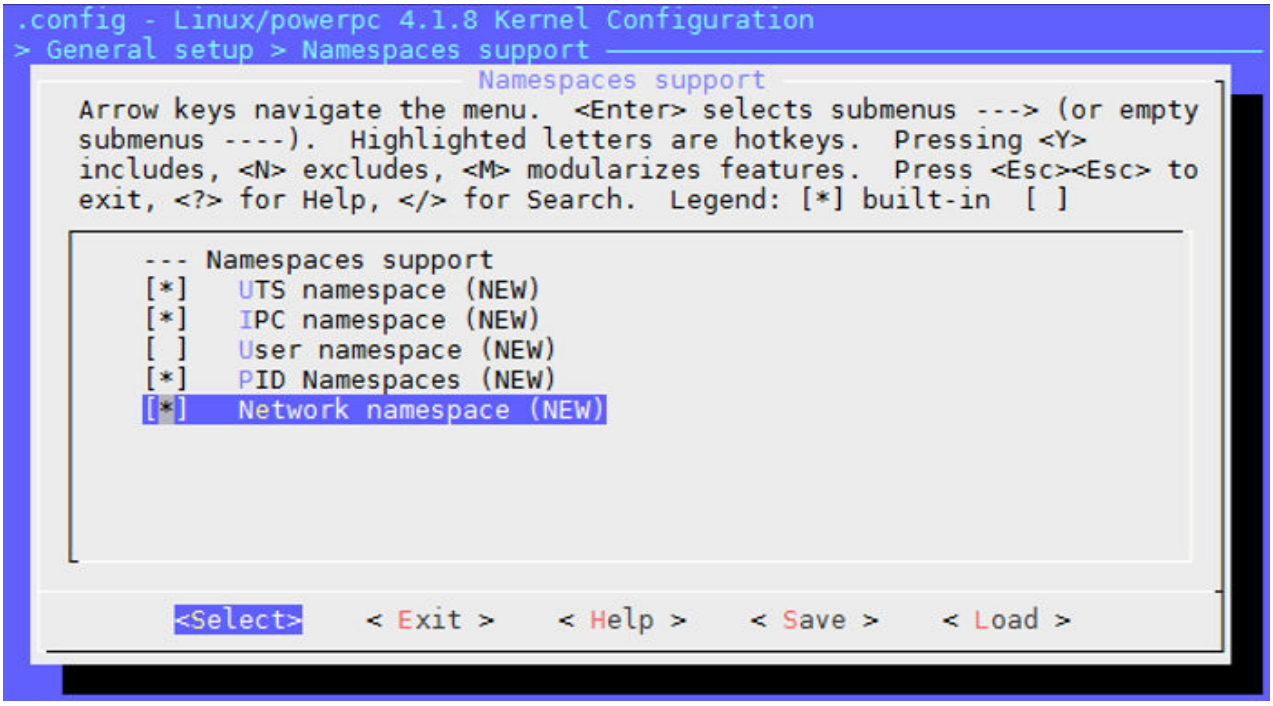
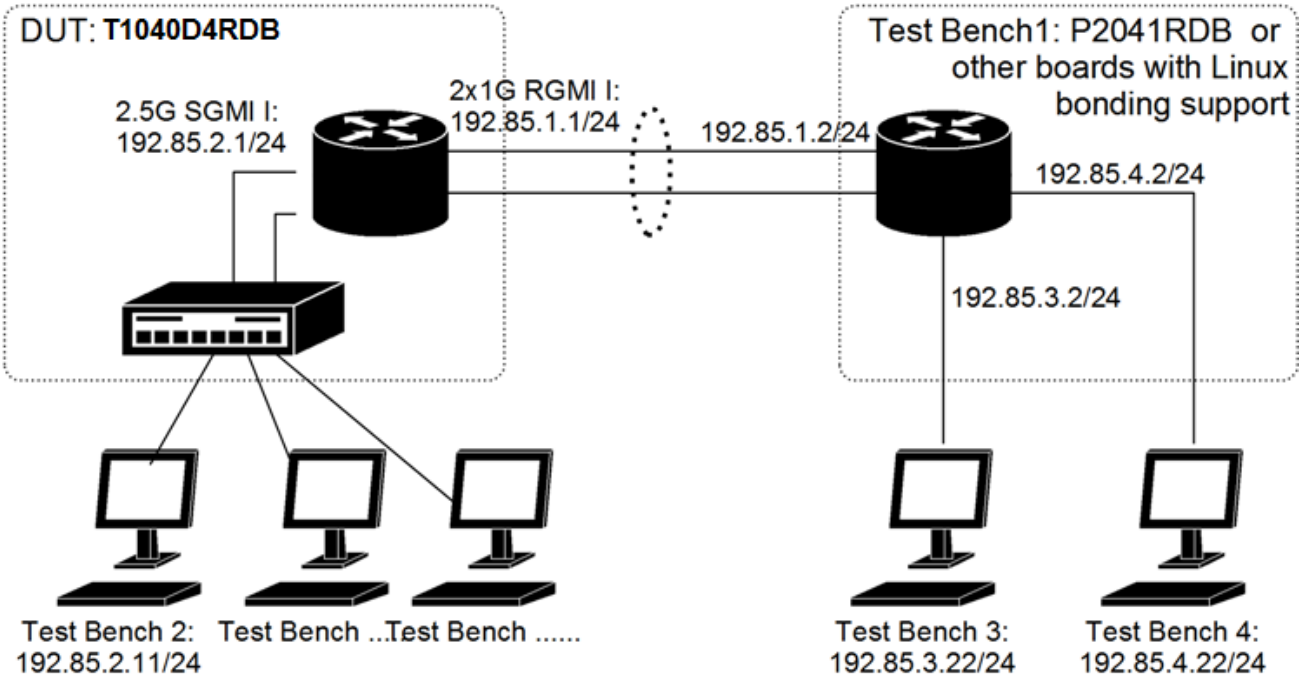
root@t1040d4rdb:~# l2switch-cfg vlan 15 add 9
Adding port 9 as member for VLAN 15
VLAN 15: 5, 9,

root@t1040d4rdb:~# l2switch-cfg vlan 16 add 9
Adding port 9 as member for VLAN 16
VLAN 16: 6, 9,

root@t1040d4rdb:~# l2switch-cfg vlan 17 add 9
Adding port 9 as member for VLAN 17
VLAN 17: 7, 9,

root@t1040d4rdb:~# l2switch-cfg vlan show
Printing all VLANs...

VLAN 1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
VLAN 10: 0, 8,
```

```
VLAN 11: 1, 8,
VLAN 12: 2, 8,
VLAN 13: 3, 8,
VLAN 14: 4, 9,
VLAN 15: 5, 9,
VLAN 16: 6, 9,
VLAN 17: 7, 9,
```

9.4.7 Troubleshooting

1. How to verify user-space driver status:

```
root@t1040d4rdb:~# /etc/init.d/l2switch status
   l2sw_bin is stopped
```

2. Check the uio-seville module is inserted:

```
root@t1040d4rdb:~# lsmod
Module                  Size  Used by
uio_seville            1589      1
```

9.4.8 Limitations

Switch driver:

- For Link Aggregation, only mode 2 is supported
- When link aggregation is enabled between FMAN ports and L2 switch internal ports, ICMP requests get duplicated after recovering from a link failure

Demo application:

- For internal ports (8 and 9) the link status is incorrectly displayed as down. Those ports do not have PHYs so no information about status can be reported

9.4.9 Appendix - Bonding configuration for internal ports

Building the Linux Kernel

Linux kernel has to be rebuilt with bonding support. In NXP SDK this can be done by:

```
> bitbake -c menuconfig virtual/kernel
```

Select "Bonding driver support" in the "Network device support" section. Configure the driver as a module.

Build the kernel with the updated configuration and also the bonding module:

```
> bitbake -c deploy virtual/kernel
```

Installing the module

The kernel module can now be copied onto the board, installed and loaded. Note that the bonding module resides within a tgz archive in the same location as the kernel does.

```
root@t1040d4rdb:~# insmod /lib/modules/`uname -r`/kernel/drivers/net/bonding/bonding.ko mode=2
miimon=100 downdelay=200 updelay=200
bonding: Ethernet Channel Bonding Driver: v3.7.1 (April 27, 2011)
```

```
bonding: MII link monitoring set to 100 ms
MODE parameter: specifies one of the bonding policies. 1 or active-backup provides fault tolerance
with only one slave in the bond being active; a different slave becomes active if the active slave
fails.
```

- MIIMON parameter: MII link monitoring frequency in milliseconds (how often the link state of each slave is inspected for link failures)
- DOWNDELAY parameter: time to wait before disabling a slave after a link failure has been detected.
- UPDELAY parameter: time to wait before enabling a slave after a link recovery has been detected.

Configuring the internal ports bonding

The L2Switch application has to be configured for ports bonding.

Enabling Link Aggregation on ports 8,9

```
root@t1040d4rdb:~# l2switch-cfg set lag mode

root@t1040d4rdb:~# l2switch-cfg set lag no 1 members 8, 9
LAG NO 1 members: 8 9
```

Disabling Link Aggregation on ports 8,9

```
root@t1040d4rdb:~# l2switch-cfg set lag no 1 members
```

The internal ports have two corresponding interfaces, named fm1-gb0 and fm1-gb1. The following commands have to be issued in order to configure them as bonded:

```
root@t1040d4rdb:~# ifconfig bond0 13.13.13.1 up
IPv6: ADDRCONF(NETDEV_UP): bond0: link is not ready
8021q: adding VLAN 0 to HW filter on device bond0
root@t1040d4rdb:~# ifenslave bond0 fm1-gb0 fm1-gb1
bonding: bond0: enslaving fm1-gb0 as an active interface with an up link.
bonding: bond0: enslaving fm1-gb1 as an active interface with an up link.
bonding: bond0: link status definitely up for interface fm1-gb0, 10 Mbps half duplex.
bonding: bond0: first active interface up!
IPv6: ADDRCONF(NETDEV_CHANGE): bond0: link becomes ready
bonding: bond0: link status definitely up for interface fm1-gb1, 10 Mbps half duplex.
```

9.5 Link Aggregation (T1040)

9.5.1 Introduction

LAG - Link Aggregation Overview

As defined in the IEEE 802.1AX-2014 Standard, Link Aggregation provides protocols, procedures, and managed objects that allow the following:

- One or more parallel instances of full-duplex point-to-point links to be aggregated together to form a Link Aggregation Group (LAG), such that a MAC Client can treat the LAG as if it were a single link.
- A resilient interconnect using multiple full-duplex point-to-point links among one to three nodes in a network and one to three nodes in another, separately administered, network, along with a means to ensure that frames belonging to any given service will use the same physical path in both directions between the two networks.

Link Aggregation allows the establishment of full-duplex point-to-point links that have a higher aggregate bandwidth than the individual links that form the aggregation, and the use of multiple systems at each end of the aggregation. This allows improved utilization of available links in bridged local area network (LAN) environments, along with improved resilience in the face of failure of individual links or systems.

LAG Topology Examples

For some topology examples for Link Aggregation, please refer to Figure 214. Topology Examples on page 1407 extracted from the IEEE 802.1AX-2014 Standard.



Figure 214. Topology Examples

T1040 Port Connectivity

There are 5 MAC ports in T1040 FMan, 3 of them can be pinout to Phys as 1Gbps RGMII/ SGMII links, others 2 of them can be pinout to Phys as 2.5G SGMII links or directly be connected to L2 Switch trunk ports via different RCW/ Serdes. Please refer to figure T1040 ports connectivity.

The L2 Switch has below interfaces:

- 8 Gigabit switch ports are external ports and are connected to external Phys
- 2 switch ports (2.5G) are connected to FMan ports

Please refer to figure T1040 Port Connectivity.



Figure 215. T1040 ports connectivity

For more details check the T1040 Reference Manual and T1040 L2 Switch Manual.

T1040 WAN ports and LAN ports

There are 3 WAN ports and 8 LAN ports in T1040, which look like figure T1040 WAN ports and LAN ports.



Figure 216. T1040D4RDB WAN ports and LAN ports

Why NXP style LAG is needed

NXP provides Data Path Acceleration Architecture (DPAA) which can offload CPU in network device driver. Linux bonding driver is at the same network device layer, in order to offload CPU while slave device selecting calculation for outgoing traffic with FSL DPAA value added – PCD (Parse, Classify, Distribute), FSL style LAG is introduced. For more detail information, please refer to FSL DPAA UM and Linux Documentation bonding.txt

Now, only T1040D4RDB board has this FSL style LAG feature.

9.5.2 Objectives

This document is based on NXP DPAA platform (T1040D4RDB) and introduces how to setup a LAG environment on NXP QorIQ DPAA system (T1040D4RDB). Hardware environment, software environment, various scenarios and Spirent Test Center (STC) ports setting are introduced. The purpose of this document is to make FAE/customers/ testing team can easily reproduce LAG cases.

9.5.3 NXP Style LAG (Link Aggregation) Features

There are 7 mode (policies) in Linux kernel bonding driver. NXP Style LAG supports mode 2 (balance-xor) and mode 4 (IEEE 802.1ax or the previous IEEE 802.3ad), here is the feature list:

- FMan 1G MAC ports: 2x1Gbps RGMII/SGMII WAN link aggregation
- FMan 2.5G MAC ports for L2 Switch: 2x2.5Gbps SGMII link aggregation
- Multi-cast/broad-cast forwarding on aggregated links
- traffics over IPv4 forwarding on aggregated links: TCP/UDP/ ICMP Destination unreachable/ICMP Echo Reply/ICMP Echo request/ICMP Info Reply/ICMP Info request/ICMP parameter problem/ICMP redirect/ICMP source quench/ICMP time exceeded/ICMP timestamp reply/ICMP timestamp requestIGMP v1 message/IGMP v2 query message/IGMP v2 report message/IGMP v3 query message/IGMP v3 report message/IPv6/IPv4 (My Test)
- traffics over IPv6 forwarding on aggregated links: TCP/ UDP/ MLDv1 Listener Done/LMDv1 Listener Report/ My Test / ICMPv6 Dest Unreachable/ ICMPv6 Echo Reply/ICMPv6 Echo Request/ICMPv6 packet Too Big/ ICMPv6 Parameter Problem/ ICMPv6 Time Exceeded/IPv4/ IPv6 Hop By Hop Header/IPv6 Authentication Header/IPv6 Authentication Header/IPv6 Routing Header /IPv6 Fragment Header/ IPv6 Encapsulation Header/ MLDv1/MLDv2 Query/MLDv2 Report /Neighbor Advertisement/ Neighbor Solicitation /Redirect/ Router Advertisement/ Router Solicitation/UDP-DHCP Ack/ UDP-DHCP Decline /UDP-DHCP Inform/ UDP-DHCP Discover/ UDP-DHCP Inform /UDP-DHCP Nak/ UDP-DHCP Offer/UDP-DHCP Release /UDP-DHCP Request/ UDP-VxLAN
- Supports both 32 bit and 64 bit Linux kernel

9.5.4 Test environment

Test environment includes hardware environment and software environment.

9.5.4.1 Hardware environment

- RCW/Serdes: RR_P_66, which means: RGMII@mac4 + RGMII@mac5 + One PCIe slot, SERDES1 Protocol is 0x66
- Switch: use the default switch, please see T1040d4rdb platform UM
- ucode: use fsl_fman_ucode_t1040_r1.1 version, please see T1040d4rdb platform UM

9.5.4.2 Software environment

9.5.4.2.1 Test benches software environment

In different test cases, software environment of test benches (end stations) are nearly the same. In most cases, any Linux machines like x86 PC/ PowerPC/ ARM with TCP/IP stack can work well as test benches. But in order to save the total count of test benches (end stations), network name space are required in test benches' Linux kernel and DUT's Linux kernel. Name space can isolate network links of one test bench, after add individual links to different network name spaces, those links seem like different links in different test benches (end stations). while aggregate 1 Gbps RGMII/SGMII WAN ports, in order to reply LACPDU, at least one test bench needs Linux bonding driver.

9.5.4.2.2 DUT software environment

9.5.4.2.2.1 Device tree

NXP Style LAG utilizes PCD function of offline port of FMan to distribute traffics. To enable offline port 2 and buffer pool 6 both which will be attached to the bundle link, please add below contents (offline port node) in the T1 board LAG dts. If they are not in, recompile the dts to get the dtb file. However the dts file has already been integrated into the NXP SDK.

Terms

```
  PCD - Parser, Classification, Distribution
  Offline ports - NXP FMan components which have PCD functionality
  Buffer pools - NXP BMan components which have buffer management functionality
```

The SDK includes the following contents in T1040 LAG dts file:

```
arch/powerpc/boot/dts/fsl/t1040d4rdb-usdpaa-lag.dts
#include "t1040d4rdb.dts"
{
    Bp6:buffer-pool@6 {
        compatible = "fsl,t1040-bpool", "fsl,bpool";
        fsl,bpid = <6>;
        fsl,bpool-ethernet-cfg = <0x0 0x0 0x0 0x6c0 0x0 0xfeedabba>;
        fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
    };
    fsl,dpaa {
        dpa-fman0-oh@2 {
            compatible = "fsl,dpa-oh";
            /* Define frame queues for the OH port*/
            /* <OH Rx error, OH Rx default> */
            fsl,qman-frame-queues-oh = <0x68 0x1 0x69 0x1>;
```

```
            fsl,bman-buffer-pools = < &bp6>;
            fsl,qman-frame-queues-tx = <0x90 0x1>;
            fsl,fman-oh-port = <&fman0_oh2>;
        };
    };
};
```

9.5.4.2.2.2 Linux kernel options

Linux kernel option - bonding is not enabled as default, to enable it and NXP style LAG, HW_DISTRIBUTION_WITH_OH must be selected manually based on optimized kernel configuration files corenet32_fmanv3l_smp_defconfig, or corenet64_fmanv3l_smp_defconfig. See the example below:

- Enable Linux bonding driver:

```
Symbol: HW_DISTRIBUTION_WITH_OH [=y]
Type  : boolean
Prompt: Bonding driver with h/w based Tx traffic distribution support
Location:
 ->Device Drivers
    -> Network device support (NETDEVICES [=y])
      -> Network core driver support (NET_CORE [=y])
        -> Bonding driver support (BONDING [=y])
```



Figure 217. kernel-config-bonding

- To enable T1040 chip, FMan V3L option is required:

```
Symbol: FMAN_V3L [=y]
Type  : boolean
Prompt: FmanV3L
  Location:
```

```
    -> Device Drivers
      -> Network device support (NETDEVICES [=y])
        -> Ethernet driver support (ETHERNET [=y])
          -> Freescale devices (NET_VENDOR_FREESCALE [=y])
            -> Frame Manager support
              -> Freescale Frame Manager (datapath) support - SDK driver (FSL_SDK_FMAN [=y])
                -> FMAN Processor support
                  -> Processor Type (<choice> [=y])
```



Figure 218. kernel-config-fman_v3l

- This feature also depends on the NXP DPAA Offline port; hence, offline ports driver should be enabled:

```
Symbol: FSL_DPAA_OFFLINE_PORTS [=y]
Type  : boolean
Prompt: Offline Ports support
  Location:
    -> Device Drivers
      -> Network device support (NETDEVICES [=y])
        -> Ethernet driver support (ETHERNET [=y])
          -> Freescale devices (NET_VENDOR_FREESCALE [=y])
            -> DPAA Ethernet (FSL_SDK_DPAA_ETH [=y])
```



Figure 219. kernel-config-offline-ports

- Enable Network namespace to make full use of network links:

```
Symbol: NET_NS [=y]
Type  : boolean
Prompt: Network namespace
  Location:
    -> General setup
      -> Namespaces support (NAMESPACES [=y])
```



Figure 220. kernel-config-ns

9.5.4.2.2.3 File system

In normal scenario, any changes to file system are unnecessary . File system built by yocto from fsl-image-core.bb in NXP SDK can fit all of below cases. For LACP (802.3ad mode) verification, to support traditional software based LACP on L2 Switch FMan ports, the 3rd-Party L2 Switch Stack is required, which is not included in NXP SDK and can be delivered as a separate package. There are lots of L2 Switch features have already been supported in NXP SDK. For more details about Switch support and configuration, please refer to the chapter L2Switch Driver User Guide.

9.5.5 Verification method

Software based verification method

- Using ping IPv4 uni-cast/multi-cast/broad-cast address, or using ping6 an IPv6 address from one end-station to others end-stations on the other side of the DUT (router/switch). If echos are received, which means traffics can pass the aggregated link of the DUT.
- Optionally, traceroute is for print the route packets trace to network host, and tracepath can help trace path to a network host discovering MTU along this path
- Netperf can be run from one end-station to others end-stations on the other side of the DUT (router/switch), which method can help verify TCP shake hands information.

STC (Spirent Test Center) traffics generator based verification method

Spirent traffics generator has many powerful functions, including traffics generation, receiving frames dump and so on. For any details, please refer to Spirent user manual.

In this user manual, only traffics flow with special traffics types are needed, which traffics types are listed in the features section above.

The required count of traffics flows and the required type of the traffics will be described alone with the following topics of test cases.

9.5.6 Test Procedure

In order to simplify test environment and save end stations (test benches), below 8 main test cases are selected for function verification of LAG. There are 2 method to verify LAG: Software based (ping/ tracerouth/ tracepath/ Netperf) and Spirent Test Center (STC) traffics generators.

9.5.6.1 Aggregation on 2x1G Links

9.5.6.1.1 Traffics over IPv4 forwarding verification

9.5.6.1.1.1 Verification with software

DUT and Test Benches Setting

Set network ports connections



Figure 221. 2x1G-IPv4-software

Set hardware environment as above, in a real case with NXP board farm, all boards and links connections look like below:

```
T1040RDB-4_RGMII1          P2041RDB-2_RGMII1
T1040RDB-4_RGMII2          P2041RDB-2_RGMII2
T1040RDB-4_SGMII1          NW_2:39
T1040RDB-4_SW-P1           P1022DS-2_eTSEC1

P2041RDB-2_RGMII1          T1040RDB-4_RGMII1
```

```
P2041RDB-2_RGMII2           T1040RDB-4_RGMII2
P2041RDB-2_SGMII1           P2041RDB-2_SGMII2
P2041RDB-2_SGMII2           P2041RDB-2_SGMII1
P2041RDB-2_Slot1_PCIe_NIC   NW_2:41
```

Boot up DUT:

Boot up DUT into Linux with right binaries like dtb, kernel uImage and rootfs, then get the prompt "#":

Commands in DUT

Check kernel version:

```
#uname -a
#cat /proc/version
```

Optional, kill irrelevant daemon

```
#killall -9 netserver
```

However, in order to avoid reset MAC table or STC ports mac addresses and reuse STC configuration file in other cases, below commands are plus, which can fit all DUT boards even they have different MAC addresses.

```
#ifconfig fm1-gb0 hw ether 00:e0:0c:00:81:00
#ifconfig fm1-gb1 hw ether 00:e0:0c:00:81:01
#ifconfig fm1-gb2 hw ether 00:e0:0c:00:81:02
#ifconfig fm1-gb3 hw ether 00:e0:0c:00:81:03
#ifconfig fm1-gb4 hw ether 00:e0:0c:00:81:04
```

Ensure L2 Switch driver enabled:

```
#lsmod
#insmod /lib/modules/xxx_kernel_version/extra/uio_seville.ko
#/etc/init.d/l2switch status
#/etc/init.d/l2switch help
#/etc/init.d/l2switch start
#/etc/init.d/l2switch status
#l2switch-cfg mac learn auto
```

Enable mode 4, (IEEE 802.1ax or the previous IEEE 802.3ad):

```
#cd /sys/class/net/bond0/bonding/
#echo 4 > mode
```

Enable IP Forwarding:

```
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Check which offline ports are probed:

```
#cat offline_ports
```

Attach one offline port to current bond as a hardware based traffic distribution helper:

```
#echo fman0-oh@2 > oh_needed_for_hw_distribution
```

Check whether they bind each other successfully or not:

```
#cat oh_needed_for_hw_distribution
```

Allow this offline port help this bond to do hardware based traffics distribution:

```
#cat oh_en
#echo 1 > oh_en
```

Check whether offline port agree to help the bond to do hardware based traffics distribution:

```
#cat oh_en
```

Add slave devices to the bond. if different version of file system is used, the name of the Ethernet may be different, maybe fmXX-gbYY or maybe fmXX-macYY, or maybe ethxxx:

```
#echo +fm1-gb3 >slaves
#echo +fm1-gb4 >slaves
#cat slaves
#cat xmit_hash_policy
```

Disable slave devices advanced features, which a bond is not aware or a bundle has not method to check, since FSL style LAG is bypassing slave devices driver:

```
#ethtool -K bond0 gso off sg off tx off
```

Configure the bonding interface with an IP address:

```
#ifconfig bond0 192.85.1.1/24 up
#route add default gw 192.85.1.2
```

Configure other network interface with an IP address to help test:

```
#ifconfig fm1-gb0 192.85.2.1/23 up
```

Enable multi-cast forwarding:

```
#cd /proc/sys/net/ipv4
#ls -l icmp_*
#cat icmp_*
#cat icmp_echo_ignore_broadcasts
#echo 0 > icmp_echo_ignore_broadcasts
#ip route add 224.0.0.0/4 dev bond0
#cd
```

Wait till others test benches setting finish:

```
#echo "Wait till others test benches setting finish:"
```

Commands in test bench1 and in the test bench4 (network name space ns1 of test bench1):

```
#ifconfig -a
#cd /sys/class/net/bond0/bonding/
#echo 4 > mode
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
#echo +fm1-gb3 > slaves
```

```
#echo +fm1-gb4 > slaves
#cat slaves

#ifconfig bond0 192.85.1.2/24 up
#route add default gw 192.85.1.1
#ip route add 224.0.0.0/4 dev bond0
#ifconfig fm1-gb0 192.85.4.2/22 up

#cd /proc/sys/net/ipv4
#ls -l icmp_*
#cat icmp_*
#echo 0 > icmp_echo_ignore_broadcasts
#cat icmp_echo_ignore_broadcasts
#cd

#killall -9 netserver
#netserver

#ifconfig
#arp -a
#route

#history
```

Wait till others test benches and DUT setting finish:

```
#echo "Wait till others test benches and DUT setting finish:"
```

Commands will be in network name space ns1 of test bench1:

```
#echo "=======below commands will be run in network namespace ns1===="
#ip netns add ns1
#ip link set fm1-gb1 netns ns1
#ip netns exec ns1 bash
```

Commands in test bench 4 (network name space ns1 of test bench1):

```
#echo $SHELL
#cat > ~/sshtest << 'EOF'
#hello, ssh test.................
#EOF

#ifconfig -a
#ifconfig fm1-gb1 192.85.4.22/22 up
#route add default gw 192.85.4.2
#ip route add 224.0.0.0/4 dev fm1-gb1

#killall -9 dropbear
#/usr/sbin/dropbear -r /etc/dropbear/dropbear_rsa_host_key -p 22 -B
#netserver -p 20000

#cd /proc/sys/net/ipv4
#ls -l icmp_*
#cat icmp_*
#cat icmp_echo_ignore_broadcasts
#echo 0 > icmp_echo_ignore_broadcasts
#cd
```

```
#echo "====wait for others testbench setting========="
```

Commands in test bench 2:

```
#ifconfig eth0 192.85.2.11/23 up
#route add default gw 192.85.2.1

#echo "======= multicast and broadcast forwarding setting====="
#cd /proc/sys/net/ipv4/
#ls -l icmp_*
#cat icmp_*
#echo 0 > icmp_echo_ignore_broadcasts
#cat icmp_echo_ignore_broadcasts
#ip route add 224.0.0.0/4 dev eth0
#cd

#echo "======create a file for ssh test====="
#cat > sshtest2 <<'EOF'
#hello, ssh test2.................
#EOF

#echo "=====wait for other testbench and DUT setting======"
#ifconfig
#arp -a
#route
```

Verification

Simple test in DUT:

```
#ping 192.85.2.11 -c 5
#ping 224.0.0.1 -c 5
```

Status checking and logs recording in DUT:

```
#ifconfig
#l2switch-cfg mac dump
#arp -a
#route

#history
```

Multi-cast and broadcast verification in test bench 1:

```
#ping -b 255.255.255.255 -c 5
#ping 224.0.0.1 -c 5
```

Verification in ns1 of testbench1

```
#traceroute 192.85.2.11
#tracepath -l 60 192.85.2.11
#ping -b 255.255.255.255 -c 5
#ping -b 192.85.3.255 -c 5
#ping 224.0.0.1 -c 5

#cd
#scp root@192.85.2.11:/home/root/sshtest2 ~/
```

```
#cat ~/sshtest2

#netperf -H 192.85.2.11 -t TCP_STREAM -l 10
#netperf -H 192.85.2.11 -t TCP_SENDFILE -l 10
#netperf -H 192.85.2.11 -t TCP_MAERTS -l 10
#netperf -H 192.85.2.11 -t TCP_RR -l 10
#netperf -H 192.85.2.11 -t TCP_CRR -l 10
#netperf -H 192.85.2.11 -t UDP_STREAM -l 10
#netperf -H 192.85.2.11 -t UDP_RR -l 10
#netperf -H 192.85.2.11 -t DLCO_STREAM -l 10
#netperf -H 192.85.2.11 -t DLCO_RR -l 10
#netperf -H 192.85.2.11 -t DLCL_STREAM -l 10
#netperf -H 192.85.2.11 -t DLCL_RR -l 10
#netperf -H 192.85.2.11 -t STREAM_STREAM -l 10
#netperf -H 192.85.2.11 -t STREAM_RR -l 10
#netperf -H 192.85.2.11 -t DG_STREAM -l 10
#netperf -H 192.85.2.11 -t DG_RR -l 10
#netperf -H 192.85.2.11 -t SCTP_STREAM -l 10
#netperf -H 192.85.2.11 -t SCTP_STREAM_MANY -l 10
#netperf -H 192.85.2.11 -t SCTP_RR -l 10
#netperf -H 192.85.2.11 -t SCTP_RR_MANY -l 10
#netperf -H 192.85.2.11 -t LOC_CPU -l 10
#netperf -H 192.85.2.11 -t REM_CPU -l 10

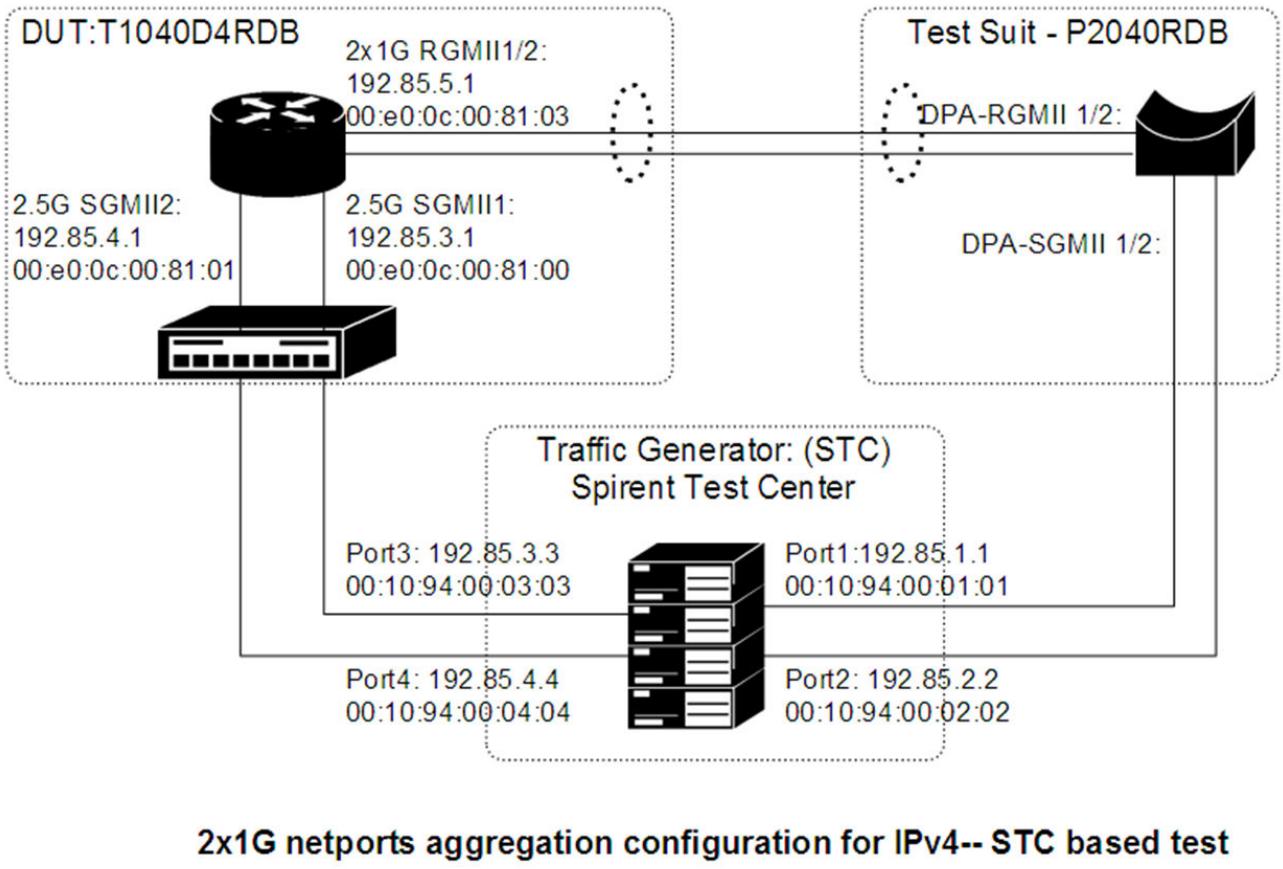
#ps aux
#ifconfig
#route
#arp -a
#history
```

Verification in test bench 2

```
#traceroute 192.85.4.22
#tracepath -l 60 192.85.4.22

#ping -b 192.85.3.255 -c 5
#ping -b 192.85.1.255 -c 5
#ping -b 192.85.7.255 -c 5
#ping -b 255.255.255.255 -c 5
#ping 224.0.0.1 -c 5


#ls -l
#scp root@192.85.4.22:/home/root/sshtest .
#cat sshtest

#echo Verify shake hands between test bench 2 and test bench 1
#netperf -H 192.85.1.2 -l 10 -t TCP_STREAM
#netperf -H 192.85.1.2 -l 10 -t TCP_SENDFILE
#netperf -H 192.85.1.2 -l 10 -t TCP_MAERTS
#netperf -H 192.85.1.2 -l 10 -t TCP_RR
#netperf -H 192.85.1.2 -l 10 -t TCP_CRR
#netperf -H 192.85.1.2 -l 10 -t UDP_STREAM
#netperf -H 192.85.1.2 -l 10 -t UDP_RR
#netperf -H 192.85.1.2 -l 10 -t DLCO_STREAM
#netperf -H 192.85.1.2 -l 10 -t DLCO_RR
#netperf -H 192.85.1.2 -l 10 -t DLCL_STREAM
#netperf -H 192.85.1.2 -l 10 -t DLCL_RR
#netperf -H 192.85.1.2 -l 10 -t STREAM_STREAM
#netperf -H 192.85.1.2 -l 10 -t STREAM_RR
```

```
#netperf -H 192.85.1.2 -l 10 -t DG_STREAM
#netperf -H 192.85.1.2 -l 10 -t DG_RR
#netperf -H 192.85.1.2 -l 10 -t SCTP_STREAM
#netperf -H 192.85.1.2 -l 10 -t SCTP_STREAM_MANY
#netperf -H 192.85.1.2 -l 10 -t SCTP_RR
#netperf -H 192.85.1.2 -l 10 -t SCTP_RR_MANY
#netperf -H 192.85.1.2 -l 10 -t LOC_CPU
#netperf -H 192.85.1.2 -l 10 -t REM_CPU

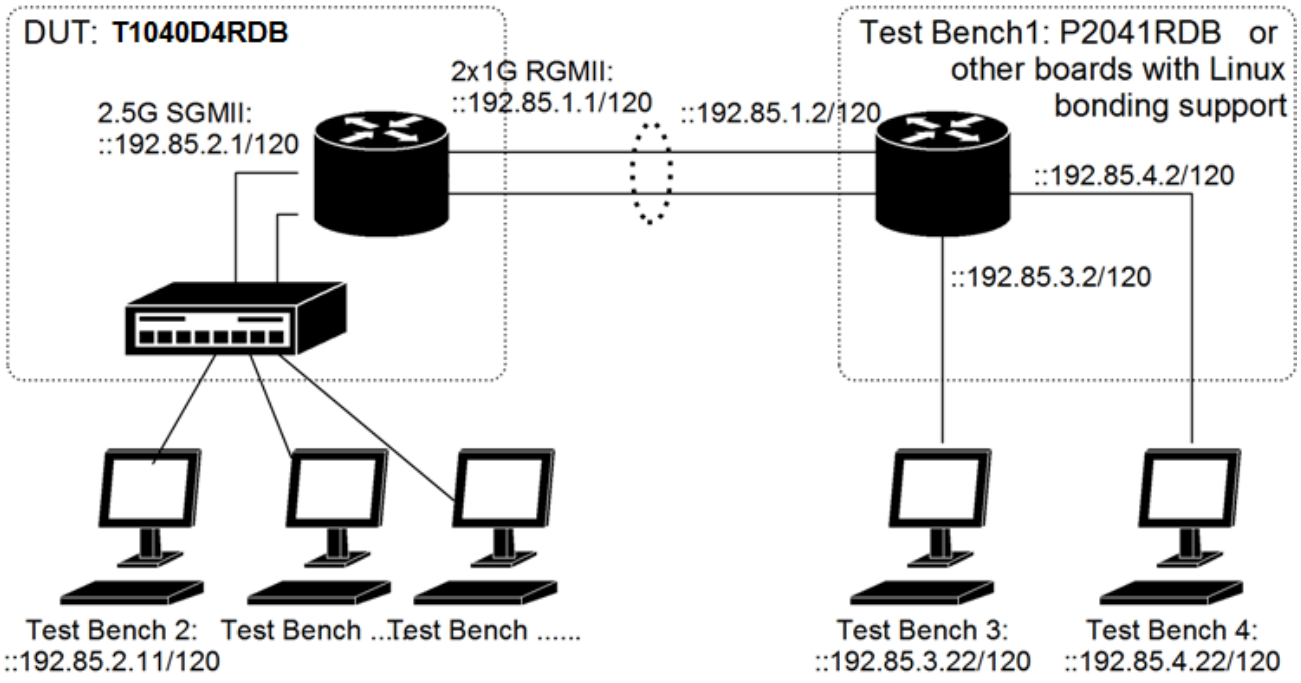
#echo Verify shake hands between test bench 2 and ns1 of test bench 1
#netperf -H 192.85.4.22 -p 20000  -l 10 -t TCP_STREAM
#netperf -H 192.85.4.22 -p 20000  -l 10 -t TCP_SENDFILE
#netperf -H 192.85.4.22 -p 20000  -l 10 -t TCP_MAERTS
#netperf -H 192.85.4.22 -p 20000  -l 10 -t TCP_RR
#netperf -H 192.85.4.22 -p 20000  -l 10 -t TCP_CRR
#netperf -H 192.85.4.22 -p 20000  -l 10 -t UDP_STREAM
#netperf -H 192.85.4.22 -p 20000  -l 10 -t UDP_RR
#netperf -H 192.85.4.22 -p 20000  -l 10 -t DLCO_STREAM
#netperf -H 192.85.4.22 -p 20000  -l 10 -t DLCO_RR
#netperf -H 192.85.4.22 -p 20000  -l 10 -t DLCL_STREAM
#netperf -H 192.85.4.22 -p 20000  -l 10 -t DLCL_RR
#netperf -H 192.85.4.22 -p 20000  -l 10 -t STREAM_STREAM
#netperf -H 192.85.4.22 -p 20000  -l 10 -t STREAM_RR
#netperf -H 192.85.4.22 -p 20000  -l 10 -t DG_STREAM
#netperf -H 192.85.4.22 -p 20000  -l 10 -t DG_RR
#netperf -H 192.85.4.22 -p 20000  -l 10 -t SCTP_STREAM
#netperf -H 192.85.4.22 -p 20000  -l 10 -t SCTP_STREAM_MANY
#netperf -H 192.85.4.22 -p 20000  -l 10 -t SCTP_RR
#netperf -H 192.85.4.22 -p 20000  -l 10 -t SCTP_RR_MANY
#netperf -H 192.85.4.22 -p 20000  -l 10 -t LOC_CPU
#netperf -H 192.85.4.22 -p 20000  -l 10 -t REM_CPU

#history
```

9.5.6.1.1.2 Verification with STC

DUT and Test Benches Setting

Set network ports connections



Figure 222. 2x1G-IPv4-STC

Set network ports connections as above, in a real case with NXP board farm, all boards and links connections look like below:

```
#bft map -m P2040RDB-1_RGMII1,T1040D4RDB-2_RGMII1,board -c
#bft map -m P2040RDB-1_RGMII2,T1040D4RDB-2_RGMII2,board -c
#bft map -m P2040RDB-1_SGMII1,Spirent_3:1,board -c --force
#bft map -m P2040RDB-1_SGMII2,Spirent_3:2,board -c --force
#bft map -m T1040D4RDB-2_SW-P3,Spirent_3:3,board -c --force
#bft map -m T1040D4RDB-2_SW-P4,Spirent_3:4,board -c --force

#bft port -l P2040RDB-1
 P2040RDB-1_RGMII1           T1040D4RDB-2_RGMII1
 P2040RDB-1_RGMII2           T1040D4RDB-2_RGMII2
 P2040RDB-1_SGMII1           Spirent_3:1
 P2040RDB-1_SGMII2           Spirent_3:2

#bft port -l T1040D4RDB-2
 T1040D4RDB-2_QSGMII1        MRV4_2:24
 T1040D4RDB-2_QSGMII2        T1040D4RDB-2_QSGMII1
 T1040D4RDB-2_RGMII1         P2040RDB-1_RGMII1
 T1040D4RDB-2_RGMII2         P2040RDB-1_RGMII2
 T1040D4RDB-2_SW-P3          Spirent_3:3
 T1040D4RDB-2_SW-P4          Spirent_3:4
```

Boot up DUT:

Boot up DUT into Linux with right binaries like dtb, kernel uImage and rootfs, then get the prompt “#”:

Commands in DUT

Check kernel version:

```
#uname -a
#cat /proc/version
```

Optional, kill irrelevant daemon

```
#killall -9 netserver
```

However, in order to avoid reset MAC table or STC ports mac addresses and reuse STC configuration file in other cases, below commands are plus, which can fit all DUT boards even they have different MAC addresses.

```
#ifconfig fm1-gb0 hw ether 00:e0:0c:00:81:00
#ifconfig fm1-gb1 hw ether 00:e0:0c:00:81:01
#ifconfig fm1-gb2 hw ether 00:e0:0c:00:81:02
#ifconfig fm1-gb3 hw ether 00:e0:0c:00:81:03
#ifconfig fm1-gb4 hw ether 00:e0:0c:00:81:04
```

Ensure L2 Switch driver enabled:

```
#lsmod
#insmod /lib/modules/xxx_kernel_version/extra/uio_seville.ko
#/etc/init.d/l2switch status
#/etc/init.d/l2switch help
#/etc/init.d/l2switch start
#/etc/init.d/l2switch status
#l2switch-cfg mac learn auto
```

Enable mode 4, (IEEE 802.1ax or the previous IEEE 802.3ad):

```
#cd /sys/class/net/bond0/bonding/
#echo 4 > mode
```

Enable IP Forwarding:

```
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Check which offline ports are probed:

```
#cat offline_ports
```

Attach one offline port to current bond as a hardware based traffic distribution helper:

```
#echo fman0-oh@2 > oh_needed_for_hw_distribution
```

Check whether they bind each other successfully or not:

```
#cat oh_needed_for_hw_distribution
```

Allow this offline port help this bond to do hardware based traffics distribution:

```
#cat oh_en
#echo 1 > oh_en
```

Check whether offline port agree to help the bond to do hardware based traffics distribution:

```
#cat oh_en
```

Add slave devices to the bond. if different version of file system is used, the name of the Ethernet may be different, maybe fmXX-gbYY or maybe fmXX-macYY, or maybe ethxxx:

```
#echo +fm1-gb3 >slaves
#echo +fm1-gb4 >slaves
#cat slaves
#cat xmit_hash_policy
```

Disable slave devices advanced features, which a bond is not aware or a bundle has not method to check, since FSL style LAG is bypassing slave devices driver:

```
#ethtool -K bond0 gso off sg off tx off
```

Configure the bonding interface and other network interface with IP address:

```
#ifconfig fm1-gb0 192.85.3.1/22
#ifconfig fm1-gb1 192.85.4.1/21
#ifconfig bond0 192.85.5.1/20
#route add default gw 192.85.3.1
#route
#arp -s 192.85.1.1 00:10:94:00:01:01
#arp -s 192.85.2.2 00:10:94:00:02:02
#free -l
#free -l -m
#route
#arp -a
#cat offline_port_xmit_statistics
#ps axu
#l2switch-cfg mac dump
```

Enable multi-cast forwarding:

```
#cd /proc/sys/net/ipv4
#ls -l icmp_*
#cat icmp_*
#cat icmp_echo_ignore_broadcasts
#echo 0 >icmp_echo_ignore_broadcasts
#cd -
```

Monitor frames from STC traffics generator:

```
#tcpdump -vvv -e -n
```

Utilize a develop board with multi network ports to create a bridge to emulate L2 Switch with full feature of LACP

```
Boot up into Linux with a bridge support then input below commands:
#ifconfig
#cat /proc/version
#cat /proc/cmdline
#ifconfig -a
#cd /sys/class/net/bond0/bonding/
#echo 4 > mode
#echo +fm1-gb3 > slaves
#echo +fm1-gb4 > slaves
```

```
        #brctl addbr brhelper
        #brctl addif brhelper fm1-gb0
        #brctl addif brhelper fm1-gb1
        #brctl addif brhelper bond0
        #echo "wait for another peer setting finish"
        #ifconfig brhelper up
        #ifconfig bond0 up
        #ifconfig fm1-gb0 up
        #ifconfig fm1-gb1 up
        #ifconfig
        #arp -a
        #route
        #brctl showmacs brhelper
```

Create traffic flows according above figure in this section:

```
4 traffic flows will be created in STC:
        STC1 --->STC3
        STC2 --->STC4
        STC3 --->STC1
        STC4 --->STC2

Above 4 traffic flows information look like below:
STC1<--->00:10:94:00:01:01,192.85.1.1 <---> STC3:192.85.3.3: 00:e0:0c:00:81:03 (neigh of the bridge
helper)
STC2<--->00:10:94:00:02:02,192.85.2.2 <---> STC4:192.85.4.4: 00:e0:0c:00:81:03 (neigh of the bridge
helper)
STC3<--->00:10:94:00:03:03,192.85.3.3 <---> STC1:192.85.1.1: 00:e0:0c:00:81:00 (L2 Switch FMan
port1)
STC4<--->00:10:94:00:04:04,192.85.4.4 <---> STC2:192.85.2.2: 00:e0:0c:00:81:01 (L2 Switch FMan
port2)
```

Verification

1.
```
Select Layer 4 type of frames among below, then send some frames, STC receiving port will receive
the same type of traffics as  expected:

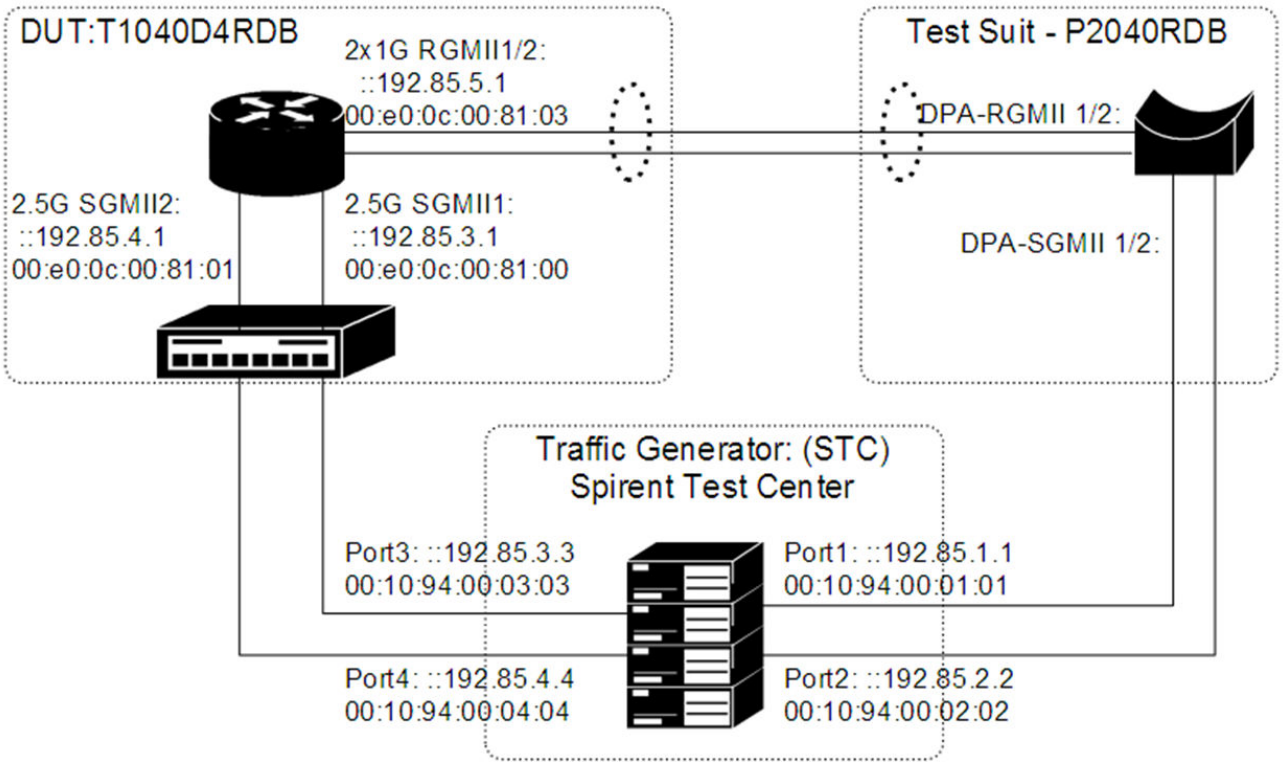
TCP/UDP/ ICMP Destination unreachable/ICMP Echo Reply/ICMP Echo request/ICMP Info Reply/ICMP
Info request/ICMP parameter problem/ICMP redirect/ICMP source quench/ICMP time exceeded/ICMP
timestamp reply/ICMP timestamp requestIGMP v1 message/IGMP v2 query message/IGMP v2 report
message/IGMP v3 query message/IGMP v3 report message/IPv6/IPv4 (My Test)
```
2. In the DUT terminal, tcpdump will also output the same type of traffics as expected.
3. Frames received in STC are also captured/recorded by STC software, which can be played in the 3rd party software like wireshark.

For more detailed information, please refer to Spirent Test Center user manual.

9.5.6.1.2 Traffics over IPv6 forwarding verification

9.5.6.1.2.1 Verification with software

DUT and Test Benches Setting

Set network ports connections



Figure 223. 2x1G-IPv6-software

Set hardware environment as above, in a real case with NXP board farm, all boards and links connections look like below:

```
T1040RDB-4_RGMII1           P2041RDB-2_RGMII1
T1040RDB-4_RGMII2           P2041RDB-2_RGMII2
T1040RDB-4_SGMII1           NW_2:39
T1040RDB-4_SW-P1            P1022DS-2_eTSEC1

P2041RDB-2_RGMII1           T1040RDB-4_RGMII1
P2041RDB-2_RGMII2           T1040RDB-4_RGMII2
P2041RDB-2_SGMII1           P2041RDB-2_SGMII2
P2041RDB-2_SGMII2           P2041RDB-2_SGMII1
P2041RDB-2_Slot1_PCIe_NIC   NW_2:41
```

Boot up DUT:

Boot up DUT into Linux with right binaries like dtb, kernel uImage and rootfs, then get the prompt “#”:

Commands in DUT

Check kernel version:

```
#uname -a
#cat /proc/version
```

Optional, kill irrelevant daemon

```
#killall -9 netserver
```

However, in order to avoid reset MAC table or STC ports mac addresses and reuse STC configuration file in other cases, below commands are plus, which can fit all DUT boards even they have different MAC addresses.

```
#ifconfig fm1-gb0 hw ether 00:e0:0c:00:81:00
#ifconfig fm1-gb1 hw ether 00:e0:0c:00:81:01
#ifconfig fm1-gb2 hw ether 00:e0:0c:00:81:02
#ifconfig fm1-gb3 hw ether 00:e0:0c:00:81:03
#ifconfig fm1-gb4 hw ether 00:e0:0c:00:81:04
```

Ensure L2 Switch driver enabled:

```
#lsmod
#insmod /lib/modules/xxx_kernel_version/extra/uio_seville.ko
#/etc/init.d/l2switch status
#/etc/init.d/l2switch help
#/etc/init.d/l2switch start
#/etc/init.d/l2switch status
#l2switch-cfg mac learn auto
```

Enable mode 4, (IEEE 802.1ax or the previous IEEE 802.3ad):

```
#cd /sys/class/net/bond0/bonding/
#echo 4 > mode
```

Enable IP Forwarding:

```
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Check which offline ports are probed:

```
#cat offline_ports
```

Attach one offline port to current bond as a hardware based traffic distribution helper:

```
#echo fman0-oh@2 > oh_needed_for_hw_distribution
```

Check whether they bind each other successfully or not:

```
#cat oh_needed_for_hw_distribution
```

Allow this offline port help this bond to do hardware based traffics distribution:

```
#cat oh_en
#echo 1 > oh_en
```

Check whether offline port agree to help the bond to do hardware based traffics distribution:

```
#cat oh_en
```

Add slave devices to the bond. if different version of file system is used, the name of the Ethernet may be different, maybe fmXX-gbYY or maybe fmXX-macYY, or maybe ethxxx:

```
#echo +fm1-gb3 >slaves
#echo +fm1-gb4 >slaves
#cat slaves
#cat xmit_hash_policy
```

Disable slave devices advanced features, which a bond is not aware or a bundle has not method to check, since FSL style LAG is bypassing slave devices driver:

```
#ethtool -K bond0 gso off sg off tx off
```

Configure the bonding interface and other network interface with IP addresses:

```
#ip -6 addr add  dev bond0 0::192.85.1.1/120
#ip -6 addr add  dev fm1-gb0 0::192.85.2.1/119
#ip link set bond0 up
#ip link set fm1-gb0 up
#ip -6 route add default via  0::192.85.1.2 dev bond0
#ip -6 route show
#ip -6 neigh show
#ip link show
#ip -6 addr show
#history
```

Wait till others test benches setting finish:

```
#echo "Wait till others test benches setting finish:"
```

Commands in test bench1 and in the test bench4 (network name space ns1 of test bench1):

```
#cd /sys/class/net/bond0/bonding/
#echo 4 >mode
#echo 1 >/proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
#echo +fm1-gb3 >slaves
#echo +fm1-gb4 >slaves
#cat slaves
#cat xmit_hash_policy
#ip -6 addr add 0::192.85.1.2/120 dev bond0
#ip -6 addr add 0::192.85.4.2/118 dev fm1-gb0
#ip link set bond0 up
#ip link set fm1-gb0 up
#ip -6 route add default via ::192.85.1.1 dev bond0
#ip link show
#ip -6 addr show
#ip -6 route show
#ip -6 neigh show
#killall -9 netserver
#netserver -6
#ps aux
#history
```

Wait till others test benches and DUT setting finish:

```
#echo "Wait till others test benches and DUT setting finish:"
```

Commands will be in network name space ns1 of test bench1:

```
#echo "=====below commands will be run in ns1======="
#ip netns help
#ip netns add ns1
#ip netns list
#ip link set fm1-gb1 netns ns1
```

```
#ip netns exec ns1 bash
```

Commands in test bench 4 (network name space ns1 of test bench1):

```
#ip link show
#ifconfig fm1-gb1 inet6 add 0::192.85.4.22/118 up
#echo 1 >/proc/sys/net/ipv6/conf/all/forwarding
#cat /proc/sys/net/ipv6/conf/all/forwarding
#ip -6 route add default via   ::192.85.4.2 dev fm1-gb1
#ip link show
#ip -6 addr show
#ip -6 route show
#ip -6 neigh show
#netserver -6 -p 20000
```

Commands in test bench 2:

```
#ip link show
#ip -6 addr add  dev eth0 0::192.85.2.11/119
#ip link set eth0 up
#ip -6 route add default via  0::192.85.2.1 dev eth0
#ip link show
#ip -6 addr show
#ip -6 route show
#ip -6 neigh show
#killall -9 netserver
#netserver -6
#ps aux
#history

#echo "=====wait for other testbench and DUT setting======"
```

Verification

Verification in ns1 of testbench1

```
#ping6 ::192.85.2.11 -c 5
#netperf -6 -H ::192.85.2.11 -l 10 -t TCP_STREAM
#ps aux
#history
#exit
```

Verification in test bench 2

```
#ping6 ::192.85.1.1 -c 5
#ping6 ::192.85.1.2 -c 5
#ping6 ::192.85.4.2 -c 5

#echo Verify shake hands between test bench 2 and test bench 1
#netperf -6 -H ::192.85.1.2 -l 10 -t TCP_STREAM
#netperf -6 -H ::192.85.1.2 -l 10 -t TCP_SENDFILE
#netperf -6 -H ::192.85.1.2 -l 10 -t TCP_MAERTS
#netperf -6 -H ::192.85.1.2 -l 10 -t TCP_RR
#netperf -6 -H ::192.85.1.2 -l 10 -t TCP_CRR
#netperf -6 -H ::192.85.1.2 -l 10 -t UDP_STREAM
```

```
#netperf -6 -H ::192.85.1.2 -l 10 -t UDP_RR
#netperf -6 -H ::192.85.1.2 -l 10 -t DLCO_STREAM
#netperf -6 -H ::192.85.1.2 -l 10 -t DLCO_RR
#netperf -6 -H ::192.85.1.2 -l 10 -t DLCL_STREAM
#netperf -6 -H ::192.85.1.2 -l 10 -t DLCL_RR
#netperf -6 -H ::192.85.1.2 -l 10 -t STREAM_STREAM
#netperf -6 -H ::192.85.1.2 -l 10 -t STREAM_RR
#netperf -6 -H ::192.85.1.2 -l 10 -t DG_STREAM
#netperf -6 -H ::192.85.1.2 -l 10 -t DG_RR
#netperf -6 -H ::192.85.1.2 -l 10 -t SCTP_STREAM
#netperf -6 -H ::192.85.1.2 -l 10 -t SCTP_STREAM_MANY
#netperf -6 -H ::192.85.1.2 -l 10 -t SCTP_RR
#netperf -6 -H ::192.85.1.2 -l 10 -t SCTP_RR_MANY
#netperf -6 -H ::192.85.1.2 -l 10 -t LOC_CPU
#netperf -6 -H ::192.85.1.2 -l 10 -t REM_CPU

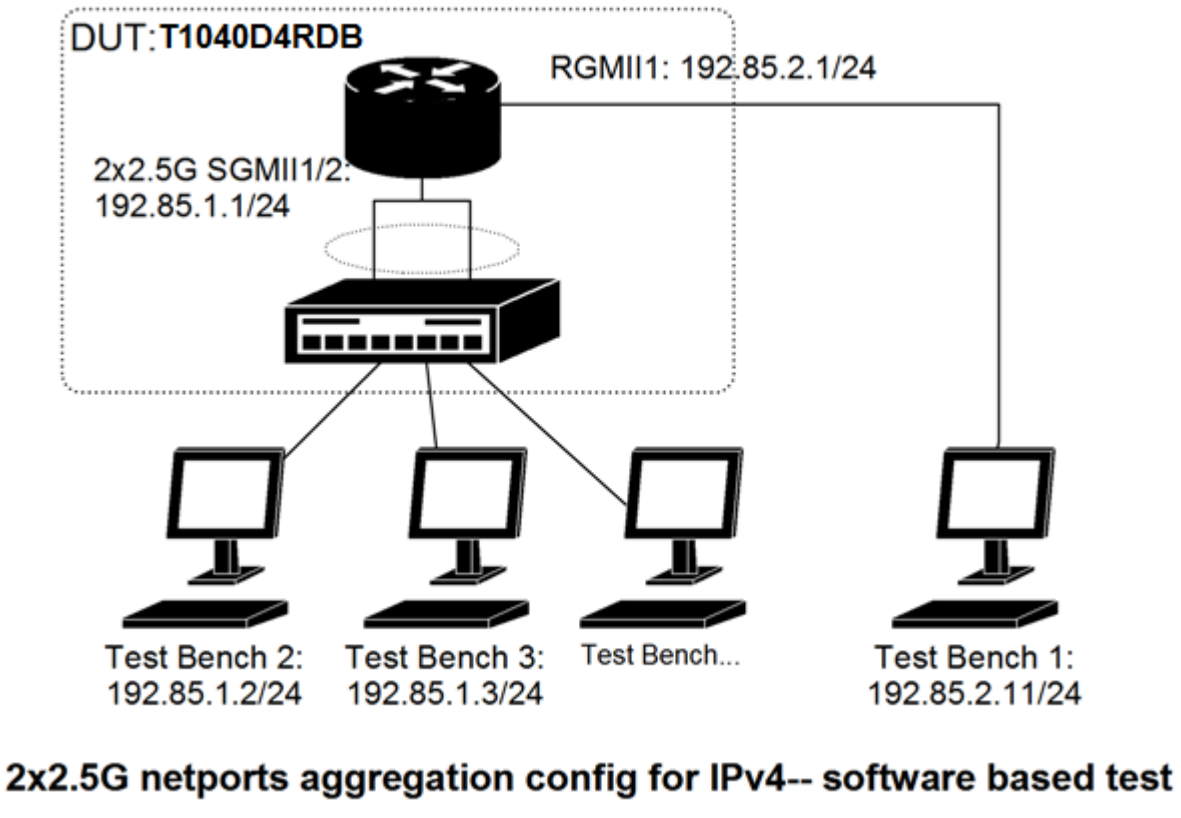
#echo Verify shake hands between test bench 2 and ns1 of test bench 1
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t TCP_STREAM
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t TCP_SENDFILE
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t TCP_MAERTS
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t TCP_RR
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t TCP_CRR
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t UDP_STREAM
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t UDP_RR
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t DLCO_STREAM
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t DLCO_RR
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t DLCL_STREAM
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t DLCL_RR
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t STREAM_STREAM
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t STREAM_RR
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t DG_STREAM
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t DG_RR
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t SCTP_STREAM
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t SCTP_STREAM_MANY
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t SCTP_RR
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t SCTP_RR_MANY
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t LOC_CPU
#netperf -6 -H ::192.85.4.22 -p 20000  -l 10 -t REM_CPU

#ps aux
#history
```

9.5.6.1.2.2 Verification with STC

DUT and Test Benches Setting

Set network ports connections



Figure 224. 2x1G-IPv6-STC

Set network ports connections as above, in a real case with NXP board farm, all boards and links connections look like below:

```
#bft map -m P2040RDB-1_RGMII1,T1040D4RDB-2_RGMII1,board -c
#bft map -m P2040RDB-1_RGMII2,T1040D4RDB-2_RGMII2,board -c
#bft map -m P2040RDB-1_SGMII1,Spirent_3:1,board -c --force
#bft map -m P2040RDB-1_SGMII2,Spirent_3:2,board -c --force
#bft map -m T1040D4RDB-2_SW-P3,Spirent_3:3,board -c --force
#bft map -m T1040D4RDB-2_SW-P4,Spirent_3:4,board -c --force

#bft port -l P2040RDB-1
 P2040RDB-1_RGMII1           T1040D4RDB-2_RGMII1
 P2040RDB-1_RGMII2           T1040D4RDB-2_RGMII2
 P2040RDB-1_SGMII1           Spirent_3:1
 P2040RDB-1_SGMII2           Spirent_3:2

#bft port -l T1040D4RDB-2
 T1040D4RDB-2_QSGMII1        MRV4_2:24
 T1040D4RDB-2_QSGMII2        T1040D4RDB-2_QSGMII1
 T1040D4RDB-2_RGMII1         P2040RDB-1_RGMII1
 T1040D4RDB-2_RGMII2         P2040RDB-1_RGMII2
 T1040D4RDB-2_SW-P3          Spirent_3:3
 T1040D4RDB-2_SW-P4          Spirent_3:4
```

Boot up DUT:

Boot up DUT into Linux with right binaries like dtb, kernel uImage and rootfs, then get the prompt “#”:

Commands in DUT

Check kernel version:

```
#uname -a
#cat /proc/version
```

Optional, kill irrelevant daemon

```
#killall -9 netserver
```

However, in order to avoid reset MAC table or STC ports mac addresses and reuse STC configuration file in other cases, below commands are plus, which can fit all DUT boards even they have different MAC addresses.

```
#ifconfig fm1-gb0 hw ether 00:e0:0c:00:81:00
#ifconfig fm1-gb1 hw ether 00:e0:0c:00:81:01
#ifconfig fm1-gb2 hw ether 00:e0:0c:00:81:02
#ifconfig fm1-gb3 hw ether 00:e0:0c:00:81:03
#ifconfig fm1-gb4 hw ether 00:e0:0c:00:81:04
```

Ensure L2 Switch driver enabled:

```
#lsmod
#insmod /lib/modules/xxx_kernel_version/extra/uio_seville.ko
#/etc/init.d/l2switch status
#/etc/init.d/l2switch help
#/etc/init.d/l2switch start
#/etc/init.d/l2switch status
#l2switch-cfg mac learn auto
```

Enable mode 4, (IEEE 802.1ax or the previous IEEE 802.3ad):

```
#cd /sys/class/net/bond0/bonding/
#echo 4 > mode
```

Enable IP Forwarding:

```
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Check which offline ports are probed:

```
#cat offline_ports
```

Attach one offline port to current bond as a hardware based traffic distribution helper:

```
#echo fman0-oh@2 > oh_needed_for_hw_distribution
```

Check whether they bind each other successfully or not:

```
#cat oh_needed_for_hw_distribution
```

Allow this offline port help this bond to do hardware based traffics distribution:

```
#cat oh_en
#echo 1 > oh_en
```

Check whether offline port agree to help the bond to do hardware based traffics distribution:

```
#cat oh_en
```

Add slave devices to the bond. if different version of file system is used, the name of the Ethernet may be different, maybe fmXX-gbYY or maybe fmXX-macYY, or maybe ethxxx:

```
#echo +fm1-gb3 >slaves
#echo +fm1-gb4 >slaves
#cat slaves
#cat xmit_hash_policy
```

Disable slave devices advanced features, which a bond is not aware or a bundle has not method to check, since FSL style LAG is bypassing slave devices driver:

```
#ethtool -K bond0 gso off sg off tx off
```

Configure the bonding interface and other network interface with IP address:

```
#ifconfig fm1-gb0 inet6 add 0::192.85.3.1/120 up
#ifconfig fm1-gb1 inet6 add 0::192.85.4.1/120 up
#ifconfig bond0   inet6 add 0::192.85.5.1/120 up
#ifconfig
#ip -6 route add default via 0::192.85.5.1 dev bond0
#route
#ip -6 neigh add 0::192.85.3.3 lladdr 00:10:94:00:03:03 dev fm1-gb0
#ip -6 neigh add 0::192.85.4.4 lladdr 00:10:94:00:04:04 dev fm1-gb1
#ip -6 neigh add 0::192.85.1.1 lladdr 00:10:94:00:01:01 dev bond0
#ip -6 neigh add 0::192.85.2.2 lladdr 00:10:94:00:02:02 dev bond0
#ip -6 neigh show
#free -l
#free -l -m
#route
#arp -a
#ip -6 route show
#ip -6 neigh show
#cat offline_port_xmit_statistics
#l2switch-cfg mac dump
#ps axu
```

Monitor frames from STC traffics generator:

```
#tcpdump -vvv -e -n
```

Utilize a develop board with multi network ports to create a bridge to emulate L2 Switch with full feature of LACP:

```
Boot up into Linux with a bridge support then input below commands:
#ifconfig
#cat /proc/version
#cat /proc/cmdline
#ifconfig -a
#cd /sys/class/net/bond0/bonding/
#echo 4 > mode
#echo +fm1-gb3 > slaves
#echo +fm1-gb4 > slaves
#brctl addbr brhelper
#brctl addif brhelper fm1-gb0
#brctl addif brhelper fm1-gb1
#brctl addif brhelper bond0
```

```
        #echo "wait for another peer setting finish"
        #ifconfig brhelper up
        #ifconfig bond0 up
        #ifconfig fm1-gb0 up
        #ifconfig fm1-gb1 up
        #ifconfig
        #arp -a
        #route
        #brctl showmacs brhelper
```

Create traffic flows according above figure in this section:

```
4 traffic flows will be created in STC:
        STC1 --->STC3
        STC2 --->STC4
        STC3 --->STC1
        STC4 --->STC2
Above 4 traffic flows information look like below:

STC1<--->00:10:94:00:01:01, ::192.85.1.1 <---> STC3: ::192.85.3.3: 00:e0:0c:00:81:03 (neigh of the
bridge helper)
STC2<--->00:10:94:00:02:02, ::192.85.2.2 <---> STC4: ::192.85.4.4: 00:e0:0c:00:81:03 (neigh of the
bridge helper)
STC3<--->00:10:94:00:03:03, ::192.85.3.3 <---> STC1: ::192.85.1.1: 00:e0:0c:00:81:00 (L2 Switch
FMan port1)
STC4<--->00:10:94:00:04:04, ::192.85.4.4 <---> STC2: ::192.85.2.2: 00:e0:0c:00:81:01 (L2 Switch
FMan port2)
```

Verification

1.
```
Select Layer 4 type of frames among below, then send some frames, STC receiving port will receive
the same type of traffics as  expected:

TCP/ UDP/ MLDv1 Listener Done/LMDv1 Listener Report/ My Test /ICMPv6 Dest Unreachable/ ICMPv6
Echo Reply/ICMPv6 Echo Request/ICMPv6 packet Too Big/ ICMPv6 Parameter Problem/ ICMPv6 Time
Exceeded/IPv4/ IPv6 Hop By Hop Header/IPv6 Authentication Header/IPv6 Authentication Header/IPv6
Routing Header /IPv6 Fragment Header/ IPv6 Encapsulation Header/ MLDv1/MLDv2 Query/MLDv2 Report /
Neighbor Advertisement/ Neighbor Solicitation /Redirect/ Router Advertisement/ Router
Solicitation/UDP-DHCP Ack/ UDP-DHCP Decline /UDP-DHCP Inform/ UDP-DHCP Discover/ UDP-DHCP
Inform /UDP-DHCP Nak/ UDP-DHCP Offer/UDP-DHCP Release /UDP-DHCP Request/ UDP-VxLAN
```

2. In the DUT terminal, tcpdump will also output the same type of traffics as expected.
3. Frames received in STC are also captured/recorded by STC software, which can be played in the 3rd party software like wireshark.

For more detailed information, please refer to Spirent Test Center user manual.

9.5.6.2 Aggregation on 2x2.5G Links

9.5.6.2.1 Traffics over IPv4 forwarding verification

9.5.6.2.1.1 Verification with software

DUT and Test Benches Setting

Set network ports connections



Figure 225. 2x2.5G-IPv4-software

Set hardware environment as above, in a real case with NXP board farm, all boards and links connections look like below:

```
T1040RDB-4_RGMII1          P1022DS-2_eTSEC1
T1040RDB-4_SW-P1           P2041RDB-2_SGMII1
T1040RDB-4_SW-P2           P2041RDB-2_SGMII2
```

Boot up DUT:

Boot up DUT into Linux with right binaries like dtb, kernel uImage and rootfs, then get the prompt "#":

Commands in DUT

Check kernel version:

```
#uname -a
#cat /proc/version
```

Optional, kill irrelevant daemon

```
#killall -9 netserver
```

However, in order to avoid reset MAC table or STC ports mac addresses and reuse STC configuration file in other cases, below commands are plus, which can fit all DUT boards even they have different MAC addresses.

```
#ifconfig fm1-gb0 hw ether 00:e0:0c:00:81:00
#ifconfig fm1-gb1 hw ether 00:e0:0c:00:81:01
#ifconfig fm1-gb2 hw ether 00:e0:0c:00:81:02
#ifconfig fm1-gb3 hw ether 00:e0:0c:00:81:03
#ifconfig fm1-gb4 hw ether 00:e0:0c:00:81:04
```

Ensure L2 Switch driver enabled:

```
#lsmod
#insmod /lib/modules/xxx_kernel_version/extra/uio_seville.ko
#/etc/init.d/l2switch status
#/etc/init.d/l2switch help
#/etc/init.d/l2switch start
#/etc/init.d/l2switch status

#l2switch-cfg set lag mode
#l2switch-cfg set lag no 1 members 8 9

#l2switch-cfg mac learn auto
```

Enable mode 2, (balance-xor 2 as default, IEEE 802.1ax or the previous IEEE 802.3ad while L2 Switch full feature stack supported):

```
#cd /sys/class/net/bond0/bonding/
#echo 2 > mode
```

Enable IP Forwarding:

```
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Check which offline ports are probed:

```
#cat offline_ports
```

Attach one offline port to current bond as a hardware based traffic distribution helper:

```
#echo fman0-oh@2 > oh_needed_for_hw_distribution
```

Check whether they bind each other successfully or not:

```
#cat oh_needed_for_hw_distribution
```

Allow this offline port help this bond to do hardware based traffics distribution:

```
#cat oh_en
#echo 1 > oh_en
```

Check whether offline port agree to help the bond to do hardware based traffics distribution:

```
#cat oh_en
```

Add slave devices to the bond. if different version of file system is used, the name of the Ethernet may be different, maybe fmXX-gbYY or maybe fmXX-macYY, or maybe ethxxx:

```
#echo +fm1-gb0 >slaves
#echo +fm1-gb1 >slaves
#cat slaves
#cat xmit_hash_policy
```

Disable slave devices advanced features, which a bond is not aware or a bundle has not method to check, since FSL style LAG is bypassing slave devices driver:

```
#ethtool -K bond0 gso off sg off tx off
```

Configure the bonding interface and other network interface with IP addresses:

```
#ifconfig bond0 192.85.1.1/24 up
#ifconfig fm1-gb3 192.85.2.1/23 up
#route add default gw 192.85.1.1
```

Enable multi-cast forwarding:

```
#cd /proc/sys/net/ipv4
#ls -l icmp_*
#cat icmp_*
#cat icmp_echo_ignore_broadcasts
#echo 0 > icmp_echo_ignore_broadcasts
#ip route add 224.0.0.0/4 dev bond0
#cd
```

Wait till others test benches setting finish:

```
#echo "Wait till others test benches setting finish:"
```

Commands in test bench 1:

```
#echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
#ifconfig eth0 192.85.2.11/23 up
#route add default gw 192.85.2.1
#killall -9 netserver
#netserver
#route

#echo "====wait for others testbench setting========="
```

Commands in test bench2 and in the net name space of the test bench2:

```
#ifconfig fm1-gb0 192.85.1.2/24 up
#route add default gw 192.85.1.1
#killall -9 netserver
#netserver

#ip netns add ns1
#ip netns list
#ip link set dev fm1-gb1 netns ns1
#ip netns exec ns1 ifconfig fm1-gb1 192.85.1.3/24 up
#ip netns exec ns1 route add default gw 192.85.1.1
#route
#ip netns exec ns1 bash
#netserver -p 20000
```

Wait till others test benches and DUT setting finish:

```
#echo "Wait till others test benches and DUT setting finish:"
```

Verification

Simple test in DUT:

```
#ping 224.0.0.1 -c 5
#ping -b 192.85.1.255 -c 5
#ping -b 192.85.3.255 -c 5
#ping -b 255.255.255.255 -c 5
#history
```

Status checking and logs recording in DUT:

```
#ifconfig
#l2switch-cfg mac dump
#arp -a
#route

#history
```

Multi-cast and broadcast verification in DUT:

```
please refer to above "Simple test in DUT"
```

Verification in testbench1:

```
#ping 192.85.1.1 -c 5
#ping 192.85.1.2 -c 5
#ps aux

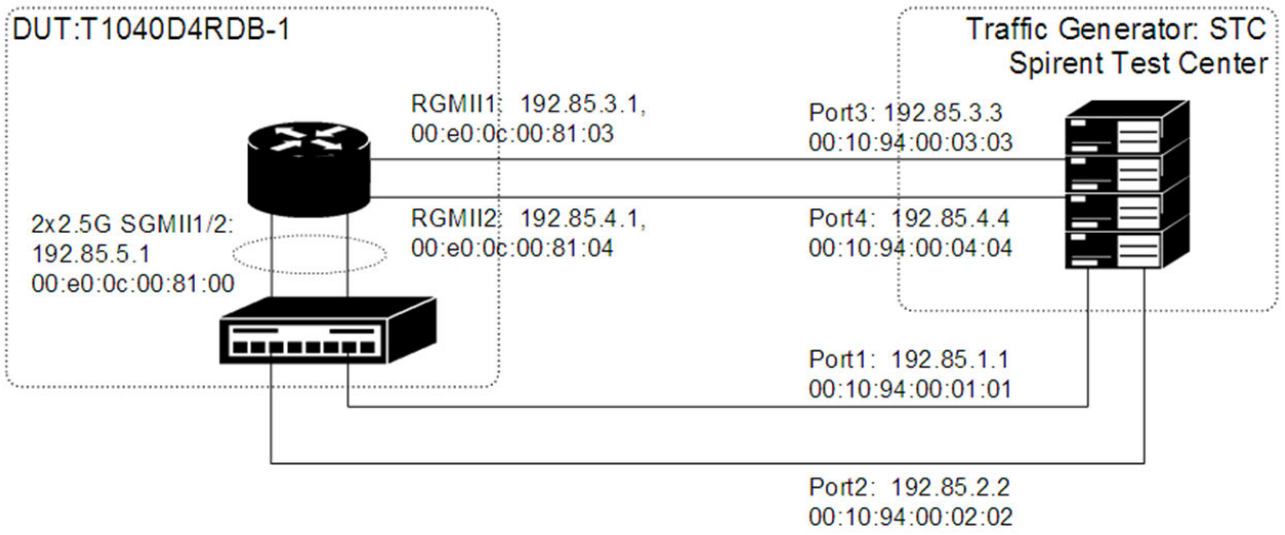
#echo Verify shake hands between test bench 1 and test bench 2
#netperf -H 192.85.1.2 -l 10 -t TCP_STREAM
#netperf -H 192.85.1.2 -l 10 -t TCP_SENDFILE
#netperf -H 192.85.1.2 -l 10 -t TCP_MAERTS
#netperf -H 192.85.1.2 -l 10 -t TCP_RR
#netperf -H 192.85.1.2 -l 10 -t TCP_CRR
#netperf -H 192.85.1.2 -l 10 -t UDP_STREAM
#netperf -H 192.85.1.2 -l 10 -t UDP_RR
#netperf -H 192.85.1.2 -l 10 -t DLCO_STREAM
#netperf -H 192.85.1.2 -l 10 -t DLCO_RR
#netperf -H 192.85.1.2 -l 10 -t DLCL_STREAM
#netperf -H 192.85.1.2 -l 10 -t DLCL_RR
#netperf -H 192.85.1.2 -l 10 -t STREAM_STREAM
#netperf -H 192.85.1.2 -l 10 -t STREAM_RR
#netperf -H 192.85.1.2 -l 10 -t DG_STREAM
#netperf -H 192.85.1.2 -l 10 -t DG_RR
#netperf -H 192.85.1.2 -l 10 -t SCTP_STREAM
#netperf -H 192.85.1.2 -l 10 -t SCTP_STREAM_MANY
#netperf -H 192.85.1.2 -l 10 -t SCTP_RR
#netperf -H 192.85.1.2 -l 10 -t SCTP_RR_MANY
#netperf -H 192.85.1.2 -l 10 -t LOC_CPU
#netperf -H 192.85.1.2 -l 10 -t REM_CPU

#echo Verify shake hands between test bench 1 and test bench3 (ns1 of test bench 2)
#netperf -H 192.85.1.3 -p 20000  -l 10 -t TCP_STREAM
#netperf -H 192.85.1.3 -p 20000  -l 10 -t TCP_SENDFILE
#netperf -H 192.85.1.3 -p 20000  -l 10 -t TCP_MAERTS
#netperf -H 192.85.1.3 -p 20000  -l 10 -t TCP_RR
#netperf -H 192.85.1.3 -p 20000  -l 10 -t TCP_CRR
#netperf -H 192.85.1.3 -p 20000  -l 10 -t UDP_STREAM
#netperf -H 192.85.1.3 -p 20000  -l 10 -t UDP_RR
```

```
#netperf -H 192.85.1.3 -p 20000  -l 10 -t DLCO_STREAM
#netperf -H 192.85.1.3 -p 20000  -l 10 -t DLCO_RR
#netperf -H 192.85.1.3 -p 20000  -l 10 -t DLCL_STREAM
#netperf -H 192.85.1.3 -p 20000  -l 10 -t DLCL_RR
#netperf -H 192.85.1.3 -p 20000  -l 10 -t STREAM_STREAM
#netperf -H 192.85.1.3 -p 20000  -l 10 -t STREAM_RR
#netperf -H 192.85.1.3 -p 20000  -l 10 -t DG_STREAM
#netperf -H 192.85.1.3 -p 20000  -l 10 -t DG_RR
#netperf -H 192.85.1.3 -p 20000  -l 10 -t SCTP_STREAM
#netperf -H 192.85.1.3 -p 20000  -l 10 -t SCTP_STREAM_MANY
#netperf -H 192.85.1.3 -p 20000  -l 10 -t SCTP_RR
#netperf -H 192.85.1.3 -p 20000  -l 10 -t SCTP_RR_MANY
#netperf -H 192.85.1.3 -p 20000  -l 10 -t LOC_CPU
#netperf -H 192.85.1.3 -p 20000  -l 10 -t REM_CPU

#history
```

Verification in test bench 2

```
#ping 192.85.1.3 -c 5
#ping 192.85.1.1 -c 5
#ping 192.85.2.1 -c 5
#ping 192.85.2.11 -c 5

#ps aux

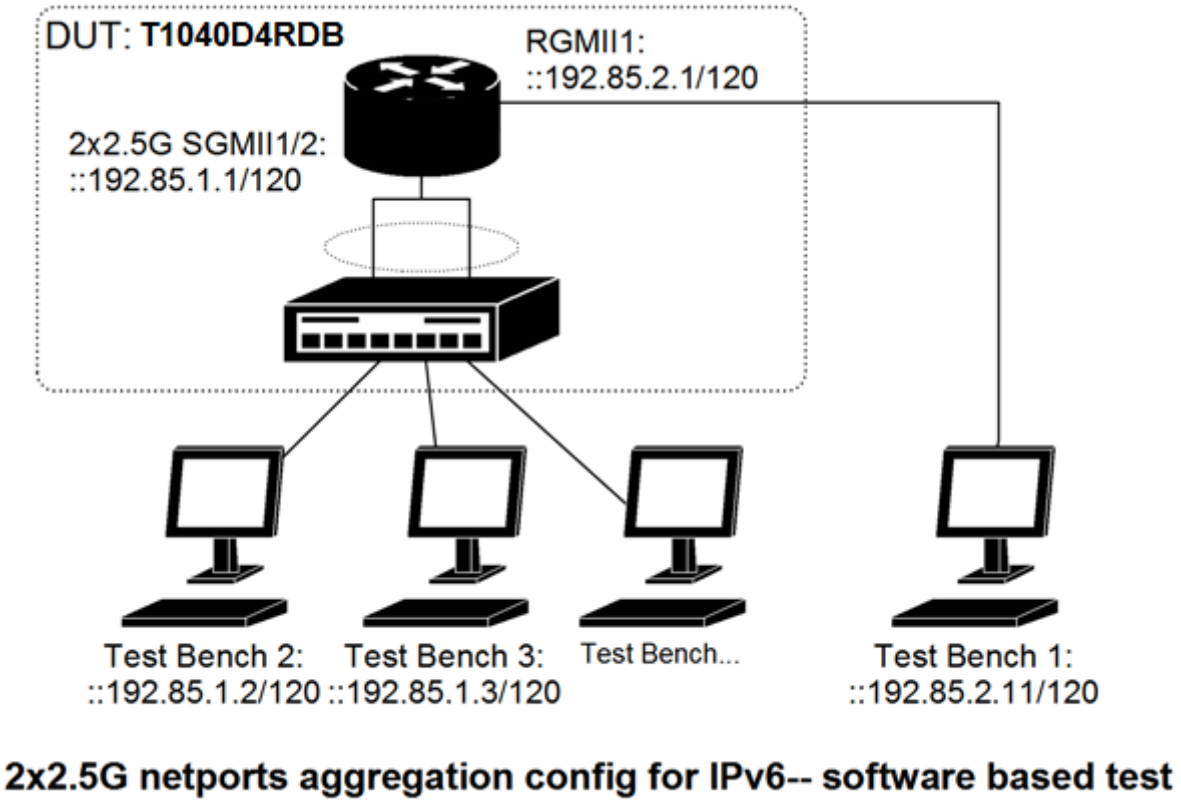
#echo Verify shake hands between test bench 2 and test bench 1
#netperf -H 192.85.2.11 -l 10 -t TCP_STREAM
#netperf -H 192.85.2.11 -l 10 -t TCP_SENDFILE
#netperf -H 192.85.2.11 -l 10 -t TCP_MAERTS
#netperf -H 192.85.2.11 -l 10 -t TCP_RR
#netperf -H 192.85.2.11 -l 10 -t TCP_CRR
#netperf -H 192.85.2.11 -l 10 -t UDP_STREAM
#netperf -H 192.85.2.11 -l 10 -t UDP_RR
#netperf -H 192.85.2.11 -l 10 -t DLCO_STREAM
#netperf -H 192.85.2.11 -l 10 -t DLCO_RR
#netperf -H 192.85.2.11 -l 10 -t DLCL_STREAM
#netperf -H 192.85.2.11 -l 10 -t DLCL_RR
#netperf -H 192.85.2.11 -l 10 -t STREAM_STREAM
#netperf -H 192.85.2.11 -l 10 -t STREAM_RR
#netperf -H 192.85.2.11 -l 10 -t DG_STREAM
#netperf -H 192.85.2.11 -l 10 -t DG_RR
#netperf -H 192.85.2.11 -l 10 -t SCTP_STREAM
#netperf -H 192.85.2.11 -l 10 -t SCTP_STREAM_MANY
#netperf -H 192.85.2.11 -l 10 -t SCTP_RR
#netperf -H 192.85.2.11 -l 10 -t SCTP_RR_MANY
#netperf -H 192.85.2.11 -l 10 -t LOC_CPU
#netperf -H 192.85.2.11 -l 10 -t REM_CPU

#history
```

9.5.6.2.1.2 Verification with STC

DUT and Test Benches Setting

Set network ports connections



Figure 226. 2x2.5G-IPv4-STC

Set network ports connections as above, in a real case with NXP board farm, all boards and links connections look like below:

```
#bft map -m T1040RDB-4_SW-P1,Spirent_3:1,board -c --force
#bft map -m T1040RDB-4_SW-P2,Spirent_3:2,board -c --force
#bft map -m T1040RDB-4_RGMII1,Spirent_3:3,board -c --force
#bft map -m T1040RDB-4_RGMII2,Spirent_3:4,board -c --force

#bft port -l t1040rdb-4
T1040RDB-4_RGMII1           Spirent_3:3
T1040RDB-4_RGMII2           Spirent_3:4
T1040RDB-4_SGMII1           NW_2:39
T1040RDB-4_SW-P1            Spirent_3:1
T1040RDB-4_SW-P2            Spirent_3:2
```

Boot up DUT:

Boot up DUT into Linux with right binaries like dtb, kernel uImage and rootfs, then get the prompt “#”:

Commands in DUT

Check kernel version:

```
#uname -a
#cat /proc/version
```

Optional, kill irrelevant daemon

```
#killall -9 netserver
```

However, in order to avoid reset MAC table or STC ports mac addresses and reuse STC configuration file in other cases, below commands are plus, which can fit all DUT boards even they have different MAC addresses.

```
#ifconfig fm1-gb0 hw ether 00:e0:0c:00:81:00
#ifconfig fm1-gb1 hw ether 00:e0:0c:00:81:01
#ifconfig fm1-gb2 hw ether 00:e0:0c:00:81:02
```

```
#ifconfig fm1-gb3 hw ether 00:e0:0c:00:81:03
#ifconfig fm1-gb4 hw ether 00:e0:0c:00:81:04
```

Ensure L2 Switch driver enabled:

```
#lsmod
#insmod /lib/modules/xxx_kernel_version/extra/uio_seville.ko
#/etc/init.d/l2switch status
#/etc/init.d/l2switch help
#/etc/init.d/l2switch start
#/etc/init.d/l2switch status
#l2switch-cfg set lag mode
#l2switch-cfg set lag no 1 members 8 9
#l2switch-cfg mac learn auto
```

Enable mode 2, (balance-xor 2 as default, IEEE 802.1ax or the previous IEEE 802.3ad while L2 Switch full feature stack supported):

```
#cd /sys/class/net/bond0/bonding/
#echo 2 > mode
```

Enable IP Forwarding:

```
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Check which offline ports are probed:

```
#cat offline_ports
```

Attach one offline port to current bond as a hardware based traffic distribution helper:

```
#echo fman0-oh@2 > oh_needed_for_hw_distribution
```

Check whether they bind each other successfully or not:

```
#cat oh_needed_for_hw_distribution
```

Allow this offline port help this bond to do hardware based traffics distribution:

```
#cat oh_en
#echo 1 > oh_en
```

Check whether offline port agree to help the bond to do hardware based traffics distribution:

```
#cat oh_en
```

Add slave devices to the bond. if different version of file system is used, the name of the Ethernet may be different, maybe fmXX-gbYY or maybe fmXX-macYY, or maybe ethxxx:

```
#echo +fm1-gb0 >slaves
#echo +fm1-gb1 >slaves
#cat slaves
#cat xmit_hash_policy
```

Disable slave devices advanced features, which a bond is not aware or a bundle has not method to check, since FSL style LAG is bypassing slave devices driver:

```
        #ethtool -K bond0 gso off sg off tx off
```

Configure the bonding interface and other network interface with IP address:

```
        #ifconfig fm1-gb3 192.85.3.1/22
        #ifconfig fm1-gb4 192.85.4.1/21
        #ifconfig bond0 192.85.5.1/20
        #ifconfig fm1-gb2 192.168.2.69/24
        #route add default gw 192.85.5.1
        #route
        #ping 192.168.2.1 -c 5
        #
        #arp -s 192.85.1.1 00:10:94:00:01:01
        #arp -s 192.85.2.2 00:10:94:00:02:02
        #arp -s 192.85.3.3 00:10:94:00:03:03
        #arp -s 192.85.4.4 00:10:94:00:04:04
        #
        #free -l
        #free -l -m
        #route
        #arp -a
        #cat offline_port_xmit_statistics
        #ps axu
        #l2switch-cfg mac dump
```

Enable multi-cast forwarding:

```
        #cd /proc/sys/net/ipv4
        #ls -l icmp_*
        #cat icmp_*
        #cat icmp_echo_ignore_broadcasts
        #echo 0 >icmp_echo_ignore_broadcasts
        #cd -
```

Monitor frames from STC traffics generator:

```
        #tcpdump -vvv -e -n
```

Create traffic flows according above figure in this section:

```
4 traffic flows will be created in STC:
        STC1 --->STC3
        STC2 --->STC4
        STC3 --->STC1
        STC4 --->STC2

Above 4 traffic flows information look like below:

STC1<--->00:10:94:00:01:01,192.85.1.1 <---> STC3:192.85.3.3: 00:e0:0c:00:81:00 (bond0 ports' MAC)
STC2<--->00:10:94:00:02:02,192.85.2.2 <---> STC4:192.85.4.4: 00:e0:0c:00:81:00 (bond0 ports' MAC)
STC3<--->00:10:94:00:03:03,192.85.3.3 <---> STC1:192.85.1.1: 00:e0:0c:00:81:03 (RGMII1 ports' MAC)
STC4<--->00:10:94:00:04:04,192.85.4.4 <---> STC2:192.85.2.2: 00:e0:0c:00:81:04 (RGMII2 ports' MAC)
```

Verification

1.
```
Select Layer 4 type of frames among below, then send some frames, STC receiving port will receive
the same type of traffics as  expected:

TCP/UDP/ ICMP Destination unreachable/ICMP Echo Reply/ICMP Echo request/ICMP Info Reply/ICMP
Info request/ICMP parameter problem/ICMP redirect/ICMP source quench/ICMP time exceeded/ICMP
timestamp reply/ICMP timestamp requestIGMP v1 message/IGMP v2 query message/IGMP v2 report
message/IGMP v3 query message/IGMP v3 report message/IPv6/IPv4 (My Test)
```

2. In the DUT terminal, tcpdump will also output traffics in the same type as expected.
3. Frames received in STC are also captured/recorded by STC software, which can be played in the 3rd party software like wireshark.

For more detailed information, please refer to Spirent Test Center user manual.

9.5.6.2.2 Traffics over IPv6 forwarding verification

9.5.6.2.2.1 Verification with software

DUT and Test Benches Setting

Set network ports connections



Figure 227. 2x2.5G-IPv6-software

Set hardware environment as above, in a real case with NXP board farm, all boards and links connections look like below:

```
T1040RDB-4_RGMII1           P1022DS-2_eTSEC1
T1040RDB-4_SW-P1            P2041RDB-2_SGMII1
T1040RDB-4_SW-P2            P2041RDB-2_SGMII2
```

Boot up DUT:

Boot up DUT into Linux with right binaries like dtb, kernel uImage and rootfs, then get the prompt "#":

Commands in DUT

Check kernel version:

```
#uname -a
#cat /proc/version
```

Optional, kill irrelevant daemon

```
#killall -9 netserver
```

However, in order to avoid reset MAC table or STC ports mac addresses and reuse STC configuration file in other cases, below commands are plus, which can fit all DUT boards even they have different MAC addresses.

```
#ifconfig fm1-gb0 hw ether 00:e0:0c:00:81:00
#ifconfig fm1-gb1 hw ether 00:e0:0c:00:81:01
#ifconfig fm1-gb2 hw ether 00:e0:0c:00:81:02
#ifconfig fm1-gb3 hw ether 00:e0:0c:00:81:03
#ifconfig fm1-gb4 hw ether 00:e0:0c:00:81:04
```

Ensure L2 Switch driver enabled:

```
#lsmod
#insmod /lib/modules/xxx_kernel_version/extra/uio_seville.ko
#/etc/init.d/l2switch status
#/etc/init.d/l2switch help
#/etc/init.d/l2switch start
#/etc/init.d/l2switch status

#l2switch-cfg set lag mode
#l2switch-cfg set lag no 1 members 8 9

#l2switch-cfg mac learn auto
```

Enable mode 2, (balance-xor 2 as default, IEEE 802.1ax or the previous IEEE 802.3ad while L2 Switch full feature stack supported):

```
#cd /sys/class/net/bond0/bonding/
#echo 2 > mode
```

Enable IP Forwarding:

```
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Check which offline ports are probed:

```
#cat offline_ports
```

Attach one offline port to current bond as a hardware based traffic distribution helper:

```
#echo fman0-oh@2 > oh_needed_for_hw_distribution
```

Check whether they bind each other successfully or not:

```
#cat oh_needed_for_hw_distribution
```

Allow this offline port help this bond to do hardware based traffics distribution:

```
#cat oh_en
#echo 1 > oh_en
```

Check whether offline port agree to help the bond to do hardware based traffics distribution:

```
#cat oh_en
```

Add slave devices to the bond. if different version of file system is used, the name of the Ethernet may be different, maybe fmXX-gbYY or maybe fmXX-macYY, or maybe ethxxx:

```
#echo +fm1-gb0 >slaves
#echo +fm1-gb1 >slaves
#cat slaves
#cat xmit_hash_policy
```

Disable slave devices advanced features, which a bond is not aware or a bundle has not method to check, since FSL style LAG is bypassing slave devices driver:

```
#ethtool -K bond0 gso off sg off tx off
```

Configure the bonding interface and other network interface with IP addresses:

```
#ip -6 addr add  dev bond0 0::192.85.1.1/120
#ip -6 addr add  dev fm1-gb3 0::192.85.2.1/119
#ip link set bond0 up
#ip link set fm1-gb3 up

#ip -6 route show
#ip -6 neigh show
#ip link show
#ip -6 addr show
#ps aux
#history
```

Wait till others test benches setting finish:

```
#echo "Wait till others test benches setting finish:"
```

Commands in test bench1:

```
#ip -6 addr add  dev eth0 0::192.85.2.11/119
#ip -6 route add default via 0::192.85.2.1 dev eth0
#ip link set eth0 up
#killall -9 netserver
#netserver -6
```

```
        #echo "====wait for others testbench setting========="
```

Commands in test bench2 and in the net name space of the test bench2:

```
        #ip -6 addr add  dev fm1-gb0 ::192.85.1.2/120
        #ip -6 route add default via  0::192.85.1.1 dev fm1-gb0
        #ip link set fm1-gb0 up
        #ip -6 route show
        #ip -6 neigh show
        #ip link show
        #ip -6 addr show
        #killall -9 netserver
        #netserver -6

        #ip netns add ns1
        #ip netns list
        #ip link set dev fm1-gb1 netns ns1
        #ip netns exec ns1 ip -6 addr add  dev fm1-gb1 0::192.85.1.3/119
        #ip netns exec ns1 ip -6 route add default via  0::192.85.1.1 dev fm1-gb1
        #ip netns exec ns1 ip link set fm1-gb1 up
        #netserver -6 -p 20000
```

Wait till others test benches and DUT setting finish:

```
        #echo "Wait till others test benches and DUT setting finish:"
```

Verification

Verification in testbench1

```
        #ping6 ::192.85.2.11 -c 3
        #ping6 ::192.85.2.1 -c 5
        #ping6 ::192.85.1.1 -c 5
        #ping6 ::192.85.1.2 -c 5
        #ping6 ::192.85.1.3 -c 5

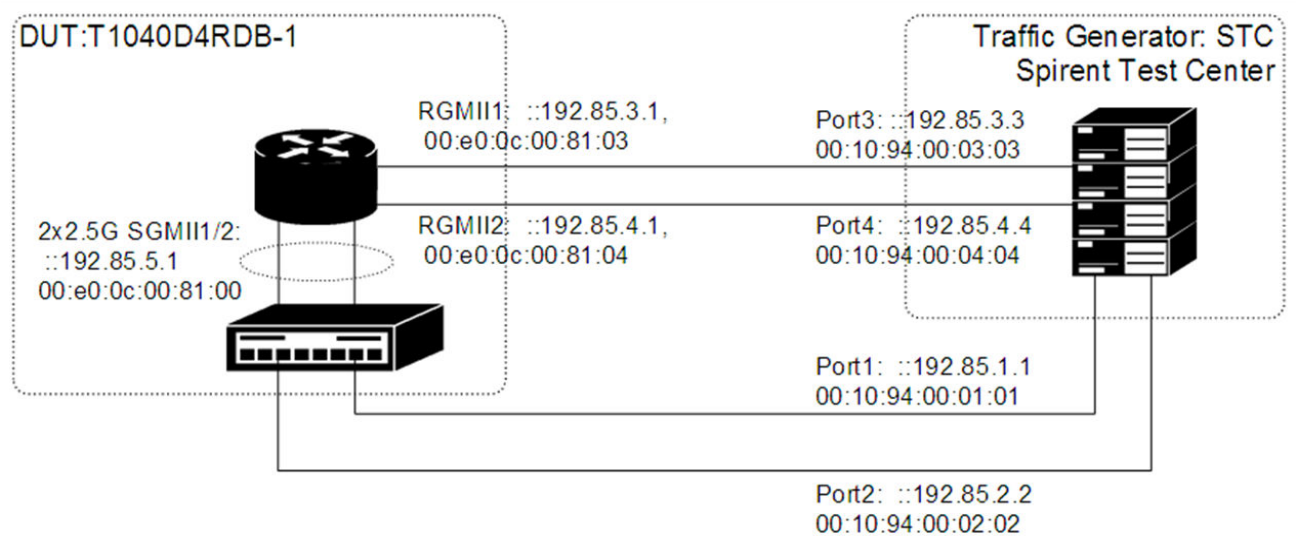
        #echo Verify shake hands between test bench 1 and test bench 2
        #netperf -H ::192.85.1.2 -l 10 -t TCP_STREAM
        #netperf -H ::192.85.1.2 -l 10 -t TCP_SENDFILE
        #netperf -H ::192.85.1.2 -l 10 -t TCP_MAERTS
        #netperf -H ::192.85.1.2 -l 10 -t TCP_RR
        #netperf -H ::192.85.1.2 -l 10 -t TCP_CRR
        #netperf -H ::192.85.1.2 -l 10 -t UDP_STREAM
        #netperf -H ::192.85.1.2 -l 10 -t UDP_RR
        #netperf -H ::192.85.1.2 -l 10 -t DLCO_STREAM
        #netperf -H ::192.85.1.2 -l 10 -t DLCO_RR
        #netperf -H ::192.85.1.2 -l 10 -t DLCL_STREAM
        #netperf -H ::192.85.1.2 -l 10 -t DLCL_RR
        #netperf -H ::192.85.1.2 -l 10 -t STREAM_STREAM
        #netperf -H ::192.85.1.2 -l 10 -t STREAM_RR
        #netperf -H ::192.85.1.2 -l 10 -t DG_STREAM
        #netperf -H ::192.85.1.2 -l 10 -t DG_RR
        #netperf -H ::192.85.1.2 -l 10 -t SCTP_STREAM
        #netperf -H ::192.85.1.2 -l 10 -t SCTP_STREAM_MANY
        #netperf -H ::192.85.1.2 -l 10 -t SCTP_RR
        #netperf -H ::192.85.1.2 -l 10 -t SCTP_RR_MANY
        #netperf -H ::192.85.1.2 -l 10 -t LOC_CPU
```

```
#netperf -H ::192.85.1.2 -l 10 -t REM_CPU

#echo Verify shake hands between test bench 1 and test bench3 (ns1 of test bench 2)
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t TCP_STREAM
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t TCP_SENDFILE
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t TCP_MAERTS
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t TCP_RR
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t TCP_CRR
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t UDP_STREAM
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t UDP_RR
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t DLCO_STREAM
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t DLCO_RR
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t DLCL_STREAM
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t DLCL_RR
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t STREAM_STREAM
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t STREAM_RR
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t DG_STREAM
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t DG_RR
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t SCTP_STREAM
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t SCTP_STREAM_MANY
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t SCTP_RR
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t SCTP_RR_MANY
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t LOC_CPU
#netperf -H ::192.85.1.3 -p 20000  -l 10 -t REM_CPU

#ip -6 route show
#ip -6 neigh show
#ip link show
#ip -6 addr show
#ps aux
#history
```

Verification in test bench 2

```
#ping6 ::192.85.1.1 -c 5
#ping6 ::192.85.1.3 -c 5
#ping6 ::192.85.2.1 -c 5
#ping6 ::192.85.2.11 -c 5

#echo Verify shake hands between test bench 2 and test bench 1
#netperf -H ::192.85.2.11 -l 10 -t TCP_STREAM
#netperf -H ::192.85.2.11 -l 10 -t TCP_SENDFILE
#netperf -H ::192.85.2.11 -l 10 -t TCP_MAERTS
#netperf -H ::192.85.2.11 -l 10 -t TCP_RR
#netperf -H ::192.85.2.11 -l 10 -t TCP_CRR
#netperf -H ::192.85.2.11 -l 10 -t UDP_STREAM
#netperf -H ::192.85.2.11 -l 10 -t UDP_RR
#netperf -H ::192.85.2.11 -l 10 -t DLCO_STREAM
#netperf -H ::192.85.2.11 -l 10 -t DLCO_RR
#netperf -H ::192.85.2.11 -l 10 -t DLCL_STREAM
#netperf -H ::192.85.2.11 -l 10 -t DLCL_RR
#netperf -H ::192.85.2.11 -l 10 -t STREAM_STREAM
#netperf -H ::192.85.2.11 -l 10 -t STREAM_RR
#netperf -H ::192.85.2.11 -l 10 -t DG_STREAM
#netperf -H ::192.85.2.11 -l 10 -t DG_RR
#netperf -H ::192.85.2.11 -l 10 -t SCTP_STREAM
#netperf -H ::192.85.2.11 -l 10 -t SCTP_STREAM_MANY
#netperf -H ::192.85.2.11 -l 10 -t SCTP_RR
#netperf -H ::192.85.2.11 -l 10 -t SCTP_RR_MANY
```

```
#netperf -H ::192.85.2.11 -l 10 -t LOC_CPU
#netperf -H ::192.85.2.11 -l 10 -t REM_CPU

#ip -6 route show
#ip -6 neigh show
#ip link show
#ip -6 addr show

#ps aux
#history
```

9.5.6.2.2.2 Verification with STC

DUT and Test Benches Setting

Set network ports connections



2x2.5G netports aggregation config for IPv6-- STC based test

Figure 228. 2x2.5G-IPv6-STC

Set network ports connections as above, in a real case with FSL board farm, all boards and links connections look like below:

```
#bft map -m T1040RDB-4_SW-P1,Spirent_3:1,board -c --force
#bft map -m T1040RDB-4_SW-P2,Spirent_3:2,board -c --force
#bft map -m T1040RDB-4_RGMII1,Spirent_3:3,board -c --force
#bft map -m T1040RDB-4_RGMII2,Spirent_3:4,board -c --force

#bft port -l t1040rdb-4
T1040RDB-4_RGMII1           Spirent_3:3
T1040RDB-4_RGMII2           Spirent_3:4
T1040RDB-4_SGMII1           NW_2:39
T1040RDB-4_SW-P1            Spirent_3:1
T1040RDB-4_SW-P2            Spirent_3:2
```

Boot up DUT:

Boot up DUT into Linux with right binaries like dtb, kernel uImage and rootfs, then get the prompt "#":

Commands in DUT

Check kernel version:

```
#uname -a
#cat /proc/version
```

Optional, kill irrelevant daemon

```
#killall -9 netserver
```

However, in order to avoid reset MAC table or STC ports mac addresses and reuse STC configuration file in other cases, below commands are plus, which can fit all DUT boards even they have different MAC addresses.

```
#ifconfig fm1-gb0 hw ether 00:e0:0c:00:81:00
#ifconfig fm1-gb1 hw ether 00:e0:0c:00:81:01
#ifconfig fm1-gb2 hw ether 00:e0:0c:00:81:02
#ifconfig fm1-gb3 hw ether 00:e0:0c:00:81:03
#ifconfig fm1-gb4 hw ether 00:e0:0c:00:81:04
```

Ensure L2 Switch driver enabled:

```
#lsmod
#insmod /lib/modules/xxx_kernel_version/extra/uio_seville.ko
#/etc/init.d/l2switch status
#/etc/init.d/l2switch help
#/etc/init.d/l2switch start
#/etc/init.d/l2switch status
#l2switch-cfg set lag mode
#l2switch-cfg set lag no 1 members 8 9
#l2switch-cfg mac learn auto
```

Enable mode 2, (balance-xor 2 as default, IEEE 802.1ax or the previous IEEE 802.3ad while L2 Switch full feature stack supported):

```
#cd /sys/class/net/bond0/bonding/
#echo 2 > mode
```

Enable IP Forwarding:

```
#echo 1 > /proc/sys/net/ipv4/ip_forward
#echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

Check which offline ports are probed:

```
#cat offline_ports
```

Attach one offline port to current bond as a hardware based traffic distribution helper:

```
#echo fman0-oh@2 > oh_needed_for_hw_distribution
```

Check whether they bind each other successfully or not:

```
#cat oh_needed_for_hw_distribution
```

Allow this offline port help this bond to do hardware based traffics distribution:

```
        #cat oh_en
        #echo 1 > oh_en
```

Check whether offline port agree to help the bond to do hardware based traffics distribution:

```
        #cat oh_en
```

Add slave devices to the bond. if different version of file system is used, the name of the Ethernet may be different, maybe fmXX-gbYY or maybe fmXX-macYY, or maybe ethxxx:

```
        #echo +fm1-gb0 >slaves
        #echo +fm1-gb1 >slaves
        #cat slaves
        #cat xmit_hash_policy
```

Disable slave devices advanced features, which a bond is not aware or a bundle has not method to check, since FSL style LAG is bypassing slave devices driver:

```
        #ethtool -K bond0 gso off sg off tx off
```

Configure the bonding interface and other network interface with IP address:

```
        #ifconfig fm1-gb3  inet6 add  ::192.85.3.1/120 up
        #ifconfig fm1-gb4  inet6 add  ::192.85.4.1/120 up
        #ifconfig bond0    inet6 add  ::192.85.5.1/120 up
        #ifconfig fm1-gb2 192.168.2.69/24 up
        #route -A inet6 add default gw ::192.85.5.1 dev bond0
        #ping 192.168.2.1 -c 5
        #l2switch-cfg mac dump
        #route
        #
        #arp -a
        #ip -6 route show
        #ip -6 neigh show
        #
        #free -l
        #free -l -m
        #cat offline_port_xmit_statistics
        #ps axu
        #history
```

Monitor frames from STC traffics generator:

```
        #tcpdump -vvv -e -n
```

Create traffic flows according above figure in this section:

```
4 traffic flows will be created in STC:
        STC1 --->STC3
        STC2 --->STC4
        STC3 --->STC1
        STC4 --->STC2

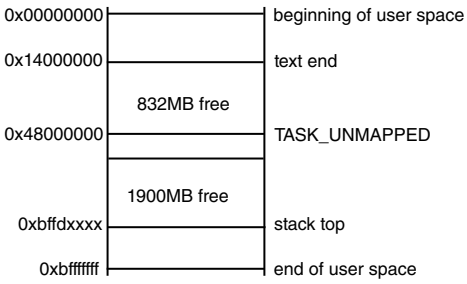
Above 4 traffic flows information look like below:
STC1<--->00:10:94:00:01:01, ::192.85.1.1 <---> STC3: ::192.85.3.3: 00:e0:0c:00:81:00 (bond0 ports'
```

```
MAC)
STC2<--->00:10:94:00:02:02, ::192.85.2.2 <---> STC4: ::192.85.4.4: 00:e0:0c:00:81:00 (bond0 ports'
MAC)
STC3<--->00:10:94:00:03:03, ::192.85.3.3 <---> STC1: ::192.85.1.1: 00:e0:0c:00:81:03 (RGMII1 ports'
MAC)
STC4<--->00:10:94:00:04:04, ::192.85.4.4 <---> STC2: ::192.85.2.2: 00:e0:0c:00:81:04 (RGMII2 ports'
MAC)
```

Verification

1.
```
Select Layer 4 type of frames among below, then send some frames, STC receiving port will receive
the same type of traffics as  expected:

TCP/ UDP/ MLDv1 Listener Done/LMDv1 Listener Report/ My Test /ICMPv6 Dest Unreachable/ ICMPv6
Echo Reply/ICMPv6 Echo Request/ICMPv6 packet Too Big/ ICMPv6 Parameter Problem/ ICMPv6 Time
Exceeded/IPv4/ IPv6 Hop By Hop Header/IPv6 Authentication Header/IPv6 Authentication Header/IPv6
Routing Header /IPv6 Fragment Header/ IPv6 Encapsulation Header/ MLDv1/MLDv2 Query/MLDv2 Report /
Neighbor Advertisement/ Neighbor Solicitation /Redirect/ Router Advertisement/ Router
Solicitation/UDP-DHCP Ack/ UDP-DHCP Decline /UDP-DHCP Inform/ UDP-DHCP Discover/ UDP-DHCP
Inform /UDP-DHCP Nak/ UDP-DHCP Offer/UDP-DHCP Release /UDP-DHCP Request/ UDP-VxLAN
```

2. In the DUT terminal, tcpdump will also output the same type of traffics as expected.
3. Frames received in STC are also captured/recorded by STC software, which can be played in the 3rd party software like wireshark.

For more detailed information, please refer to Spirent Test Center user manual.

9.5.6.3 Note

Note:

1. As default, valid Netperf testnames are not always compiled-in. For more detail Netperf testing, please refer to the Netperf UM.
2. When slow in lacp_rate as default, ping sometimes fail for a couple of seconds while a slave link quits the aggregation group from the other peer, which is normal phenomenon.
3. There are many free sshd programs, for example: dropbear, openssh and etc,. If the rootfs of current sdk version excludes dropbear, please refer to your sshd user manual accordingly for start or restart the ssh daemon.

9.5.7 Known Bugs, Limitations, or Technical Issues

1. Performance limitation: NXP style LAG performance depends on Offline port forwarding speed. By now, it is lower than the idea speed of FMan v3 version:3.75Mpps, need an enhancement in subsequent release.
2. Linux bonding is using XOR L2/L3/L4 header information to calculate slave device. But FMan supports neither XOR nor bi-directional SHIFT operator. NXP style LAG uses CRC-64 based hashing to instead.
3. Current design is bypassing DPA-Eth driver, so good features of the DPA-Eth driver are not available, such as SG-GSO, IEEE1588, etc..
4. Current NXP LAG only support 2 DPA slaves per bundle, this limitation dues to current Keygen hardware, which is requiring hashDistributionNumOfFqids" to be a power of 2, which implies the count of slave device has to be 2, 4, 8, 16.
5. Current NXP style LAG only support one bundle, other bundles are based on software, this limitation dues to lack of available count of Offline ports. By now, there are only 3 Offline ports in T104x/T102x. So far, OP 3# and OP 4# are reserved for oNIC and CAPWAP, OP 2# for LAG. This limitation will be eliminated in the future.

6. L3/4 header information have not been calculated in when xmit_hash_policy changes for current version, will be enhanced in the future version.
7. Non-reconfigurable: This release support attach/detach slave devices to/from a bundle only once, dues to design limitation. If a bond does not need offline port's help for traffics, below command maybe a workaround while a bundle is in inactive status : "echo 0 > /sys/class/net/bond0/bonding/ oh_en". NXP internal CRS: ENGR00326388, ENGR00326299, ENGR00326297 are tracking this.
8. Termination based functionalities and hardware L4 CSUM by offline ports are not available for this release. For forwarding path, L4 CSUM for xmit path is not necessary. L4 CSUM is only required in termination cases. These functionalities will be done in subsequent releases.
9. Jumbo Frame is not supported yet, DPA driver does this by FMan MAC ports. NXP style LAG will do this by FMan Offline ports while this is required.
10. The 3rd party L2 Switch protocol: For 2x2.5G L2 Switch FMan ports aggregation, the 3rd party full feature stack of L2 Switch is needed to support IEEE802.3ad/IEEE802.1ax LAG, because if the protocol stack is not available, 2.5G L2 Switch FMan ports can't reply LACP signalling. From SDK-v2.0, balance-xor mode Link Aggregation is provied.

9.6 Linux HugeTLBFS User Guide

9.6.1 Introduction

Instructions for using large pages for networking application performance improvement with Linux HugeTLBFS.

It is becoming customary for user-level applications to have large memory requirements in embedded systems. Even so, the default page size in Linux is 4KB while the amount of memory mappable in TLB0 with 4KB pages is small. Small TLBs may result in many TLB misses for large applications. This document illustrates, with examples, how to properly configure a QorIQ device for large page sizes.

9.6.2 Objectives

These are results expected for the method outlined:

- Baseline understanding of Linux HugeTLBFS.
- Identify any optimizations that have been discovered and ensure they are implemented.
- Investigate other changes that may improve performance.
- Compare these results with end product performance objectives.

9.6.3 TLBFS Basics

The amount of memory mappable in TLB0 with 4KB pages is small:

Table 214. Mappable Memory of e500 Versions

Processor	No. TLB Entries	Mappable by TLB0
e500 v1	256	1MB
• e500 v2 • e500MC • e5500	512	2MB

9.6.3.1 4KB TLB0 Miss Issues

The salient contributors to TLB0 miss issues:

- TLB0 is shared between multiple processes.
- TLB0 is used for mapping both instructions and data.
- Thrashing - A single MMU-hogging process in the system can have performance inpact on smaller processes running in the system.
- A single process may use significantly more than 2M.

For the case of conflict miss, TLB0 is 4-way set associative (2-way on e500v1), which has a higher potential for conflict miss than a fully-associative TLB cache. Another concern is latency issues caused by e500 reloads of TLB entries using software which has a latency of at least 100-200 cycles per TLB.

9.6.3.2 e500 Family TLB1

The e500 processor family provides a second software-visible TLB structure, the TLB1. This is sometimes referred to as the TLBCAM. Because the TLB1 is fully associative, new mappings may be placed in any entry.

TLB1 supports a number of large page sizes but has a small number of entries:

Table 215. e500 Family TLB1 Characteristics

Processor	TLB1 Entries	Page Size
e500 v1	16	4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB
e500 v2	16	Same as e500 v1, also, 1GB, 4GB
e500MC, e5500	64	Same as e500 v2

9.6.3.3 HugeTLB Basics

Hugetlbfs allows the Linux kernel to use large pages mapped by TLB1 for user processes:

- The kernel itself does not use “hugetlbfs”; it uses a few large TLB1 entries to map some amount of memory using large mappings
- Drivers that run as part of the kernel cannot use hugetlbfs (may be a short-term restriction)

The kernel will not allocate hugepages unless requested:

- This is intentional - the number of TLB1 entries is limited so they should be used with care.
- Hugepages can cause performance problems if misused.

Hugetlb for BookE supports 4MB, 16MB, 64MB, 256MB, and 1GB pages:

- This is a subset of the hardware page sizes supported by TLB1.
- We cannot easily support a page size less than 4MB due to the current page table architecture of the 32-bit Linux kernel.

Hugepages can be used for:

- program
- .text
- .data

- .bss
- `malloc()`
- `mmap()`
- `shmget()`

Using hugepages may or may not require application modification and/or rebuild.

9.6.3.4 Normal vs. Gigantic hugepages

Page sizes larger than 16MB are called **Gigantic** pages. This depends on `CONFIG_FORCE_MAX_ZONEORDER` in Linux and may vary on different kernels; this value is for the FSL BSP. There are differences in how normal hugepages and gigantic hugepages are allocated in the kernel:

Table 216. (Normal) Huge vs. Gigantic Pages

Type	Sizes	Allocation	Deallocation
Normal	4MB, 16MB	At boot, or while system is running	Allowed
Gigantic	64MB, 256MB, 1GB	Only at boot time	Not allowed

The table below lists several limiting cases for Gigantic pages:

Table 217. Gigantic Page Limitations

Limiting Case	Description
Gigantic pages should be allocated with care since they cannot be deallocated.	Has the effect of removing RAM from the system if the hugepages are not used.
Virtual and physical addresses in the TLB must be aligned to the huge page size requested.	• May not be possible for the kernel to allocate many gigantic pages because of alignment requirements • example: The user's virtual address space is limited to 3GB total on 32-bit Linux, so at most one or two 1GB pages and eight or so 256MB pages may be used by a single process

9.6.4 Building and Booting Linux with Hugetlb

Linux must be built with CONFIG_HUGETLBFS and CONFIG_HUGETLB_PAGE set; CONFIG_FORCE_MAX_ZONEORDER should be set to 13.

- Upcoming FSL BSPs have these options enabled
- The publicly available Linux trees do not yet contain the Hugetlb support; this update is pending publication.

Specify the default hugepage size to allocate hugepages via the kernel boot command line:

- `default_hugepagesz=16m hugepagesz=16m hugepages=25 hugepagesz=4m hugepages=25`
- Once allocated, hugepages are removed from the available memory pool and will only be used for huge mappings. Avoid allocating hugepages that you don't expect to use.

9.6.4.1 Booting Linux for HugeTLB

Once Linux is booted, follow these steps:

Table 218. Step 1: The hugetlbfs filesystem must be mounted by root.

Description	Action
Check that hugetlb is enabled:	• `grep hugetlbfs /proc/filesystems` • `grep HugePages_Total /proc/meminfo` (look for a non-zero number)
Mount the default huge page size:	`mount -t hugetlbfs none /mnt/hugetlbfs`
Mount a secondary page size:	`mount -t hugetlbfs none –opagesize=16m /mnt/hugetlbfs-16M`
To allow non-root to use hugepages:	mount point perms = 1777; or use `hugeadm` to create a user or group-specific mount point

Step 2: Build and install libhugetlbfs

- simple `make; make install`
- Will be included in the BSP.
- Specific version from NXP required.

Step 3: Use `hugeadm` utility to query/change huge page parameters:

- Change the number of non-gigantic huge pages available.
- Query the size of the current huge page pool.
- View supported page sizes.
- Create hugepage mount points.

Step 4: Use `HUGETLB_DEBUG=yes` in environment to help debug applications linked with the library when there are issues

9.6.4.2 Setting Up Mount Points with hugeadm

4 commands are provided for creating mount points:

Table 219. Rev. History Table

Command Line	Description
--create-mounts	Creates a mount point for each available huge page size on this system under `/var/lib/hugetlbfs`
--create-user-mounts [user]	Creates a mount point for each available hugepage size under `/var/lib/hugetlbfs/[user]` usable by [user] [user]
--create-group-mounts [group]	Creates a mount point for each available huge page size under `/var/lib/hugetlbfs/[group]` usable by group [group]
--create-global-mounts	Creates a mount point for each available huge page size under `/var/lib/hugetlbfs/global` usable by anyone

9.6.4.3 Running Hugepage mount/query Example

Each result of each action in the example is shown in the tables

```
-bash-3.2# mount -t hugetlbfs none /mnt/hugetlbfs
```

```
-bash-3.2# hugeadm --pool-list
```

Table 220. Pool List

Size	Minimum	Size	Minimum	Default
4194304	25	25	25	
16777216	25	25	25	*
67108864	0	0	0	
268435456	0	0	0	
1073741824	0	0	0	

```
-bash-3.2# hugeadm –pool-pages-min 4M:100
```

```
-bash-3.2# hugeadm --pool-list
```

Table 221. Pool List

Size	Minimum	Size	Minimum	Default
4194304	100	100	100	
16777216	25	25	25	*
67108864	0	0	0	
....				

9.6.5 Hugepage Allocation in User Programs

User applications can request hugepages several ways:

Table 222. Pool List

Method	Steps
Shared memory	• `shmget()` call specifies `MAP_HUGETLB` • Link application with libhugetlbfs to override all `shmget()` calls
Heap-based allocations	Causes all `malloc()` calls (and C++ `new` operator) to allocate in a huge page heap region.

Table continues on the next page...

Table 222. Pool List (continued)

Method	Steps
`mmap()`	• Anonymous `mmap() (MAP_ANONYMOUS)` with `MAP_HUGETLB mmap()` • `mmap()` file on the hugetlbfs mount point.
Back program segments with hugepages	.text, .data, and .bss segments

9.6.5.1 Using Shared Memory Hugepages: SHM_HUGETLB

Specify `SHM_HUGETLB` in `shmget()` flags inapplication code

- Requires application modification to add SHM_HUGETLB to shmget() arguments.
- Application must be rebuilt.
- Libhugetlbfs is not required to use this method (no special linking needed).
- Can specify exactly which shmget() calls use hugepages.
- Can only use the systemwide default hugepage size.
- The size requested must be a multiple of the huge page size.

9.6.5.2 Running the SHM_HUGETLB Example

The code below illustrates several of the concepts discussed so far:

```
    #define LENGTH 4 * 1024 * 1024

/* Create shared memory region */
if ((shmid = shmget(2, LENGTH, SHM_HUGETLB | IPC_CREAT | SHM_R | SHM_W)) < 0)
    exit(1);

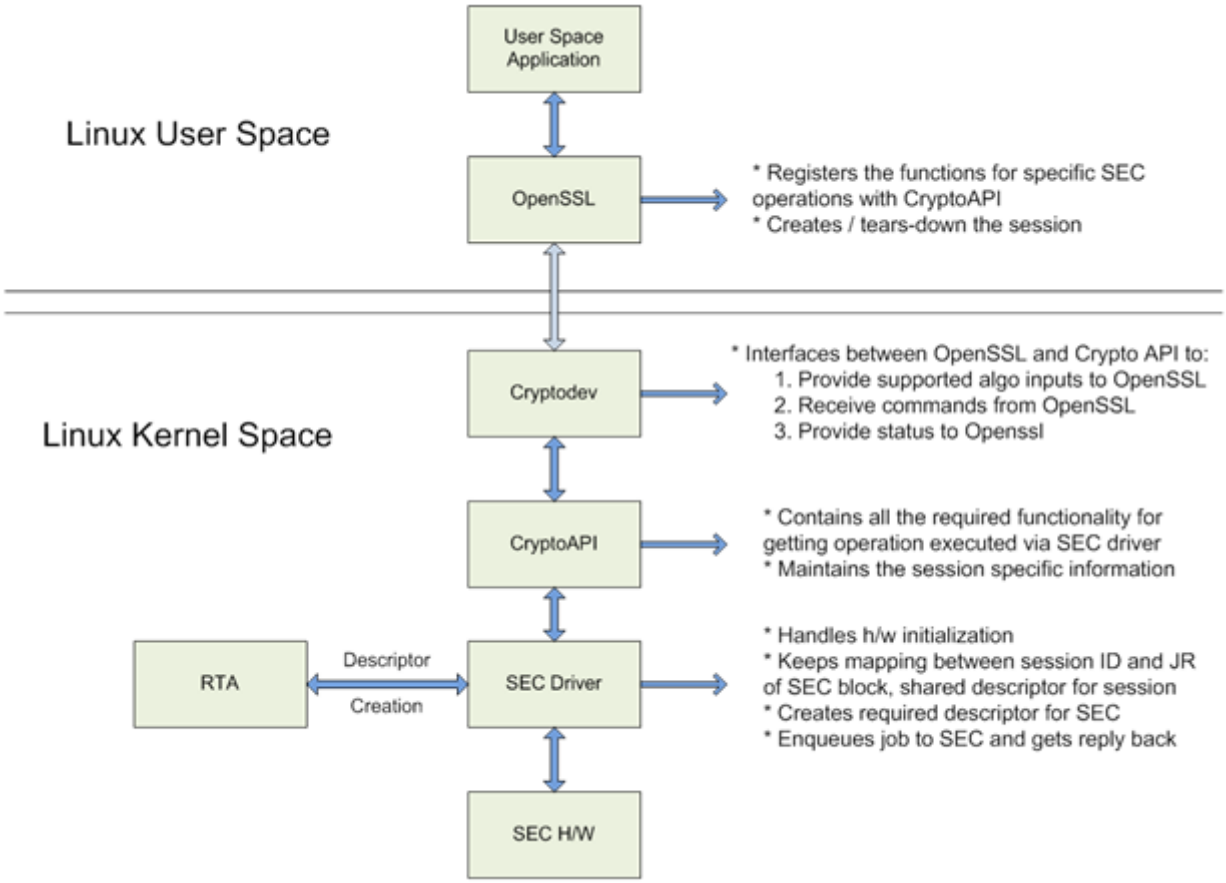
    shmaddr = shmat(shmid, 0, 0); /* Attach the region to the process */
if (shmaddr == (char *)-1) {
    shmctl(shmid, IPC_RMID, NULL);
    exit(2);
}

for (i = 0; i < LENGTH; i++)         /* Example write to the region */
    shmaddr[i] = (char)(i);

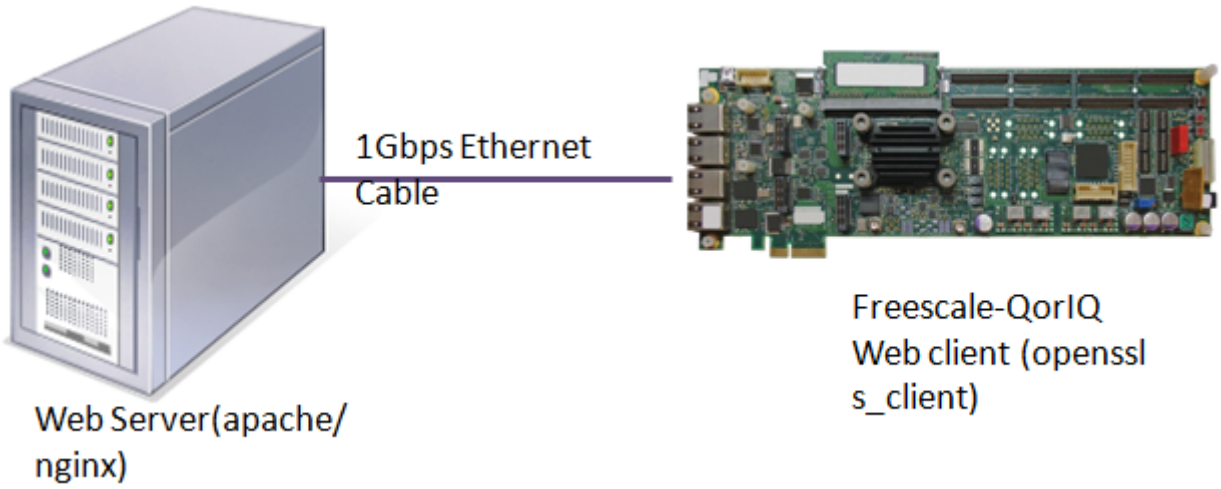
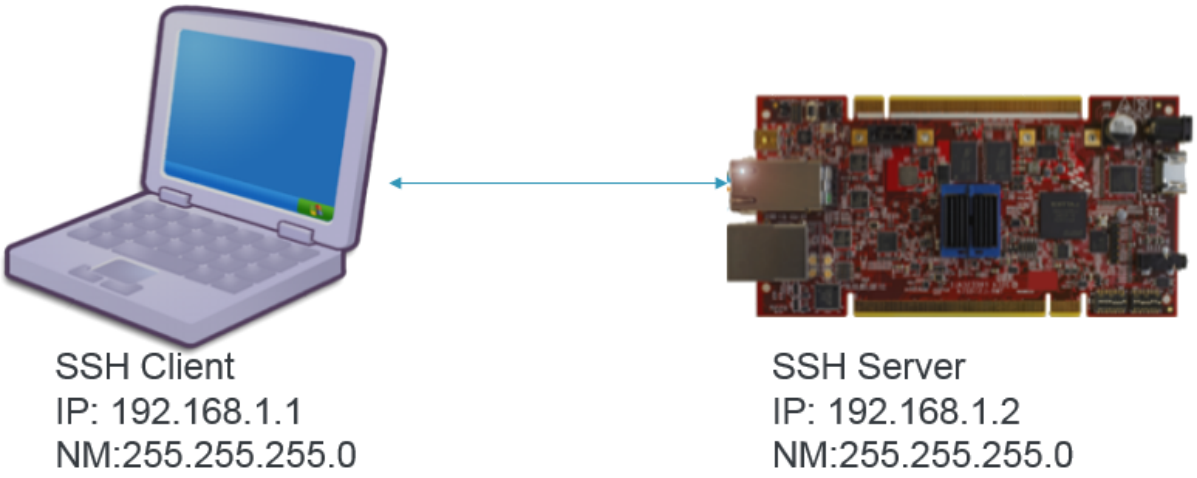
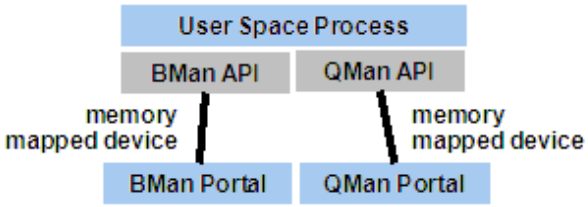
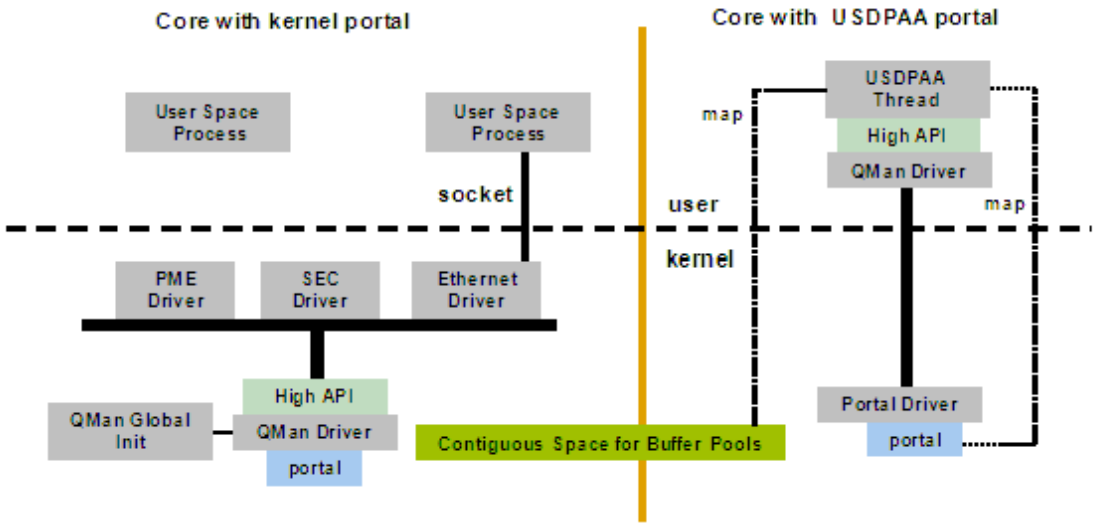
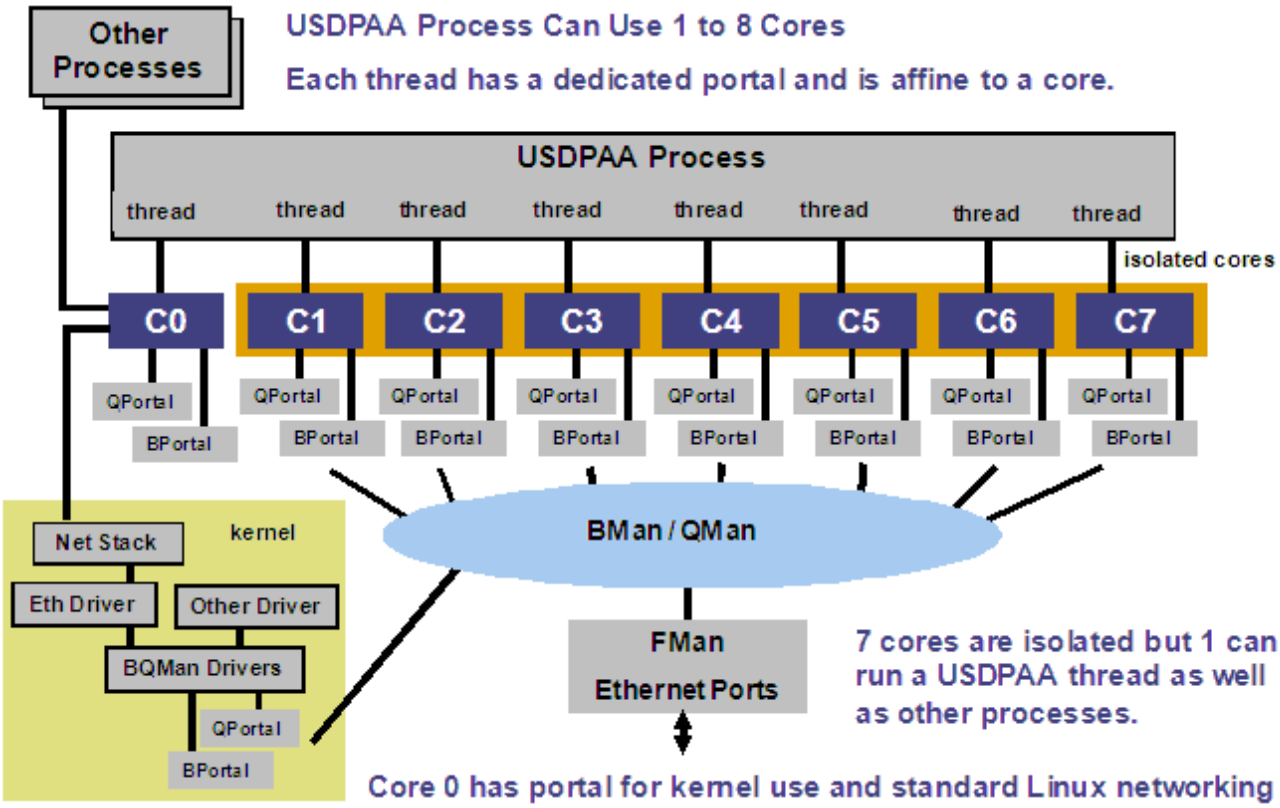
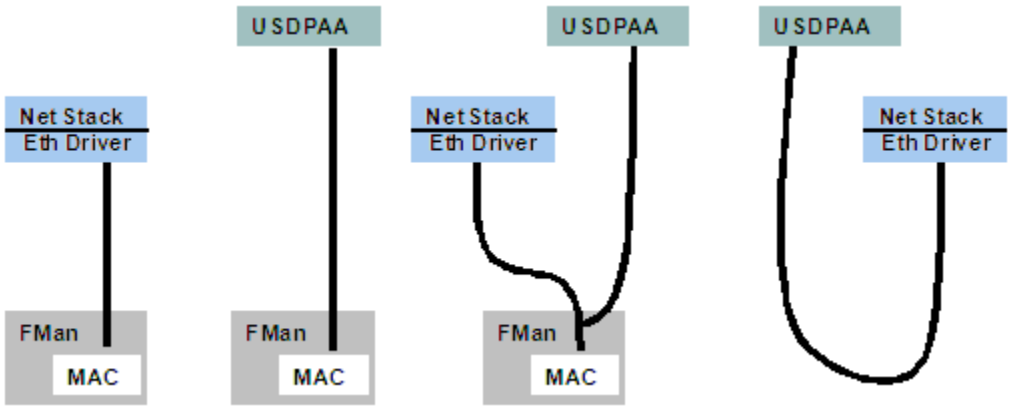
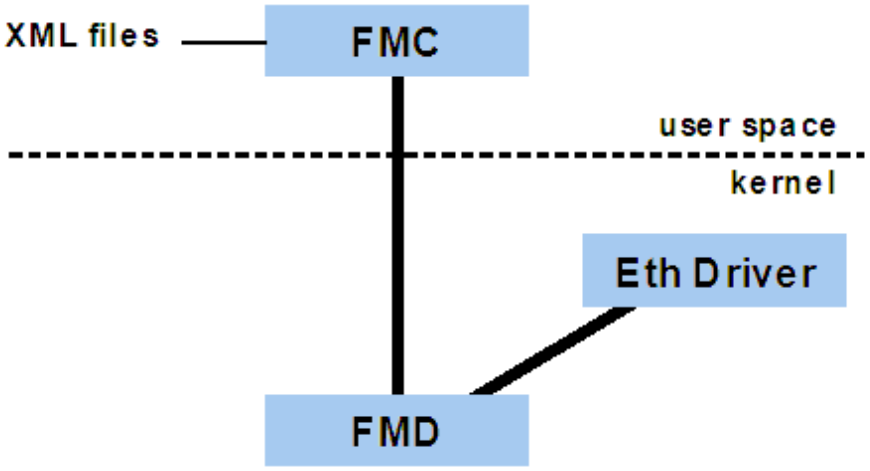
if (shmdt((const void *)shmaddr) != 0) { /* Detach the region */
    shmctl(shmid, IPC_RMID, NULL);
    exit(3);
}
shmctl(shmid, IPC RMID, NULL); /* Remove the shmid */
```

9.6.5.3 Using Shared Memory Hugepages: Link with libhugetlbfs

Instead of specifying `SHM_HUGETLB`, link application with libhugetlbfs to override all `shmget()` calls

- No application modification required.
- With dynamic linking, can use an existing binary.

- Can only use the systemwide default hugepage size.
- Simply invoke the application as follows (dynamic link):
  - `LD_PRELOAD=libhugetlbfs.so HUGETLB_SHM=yes [your app command line]`
- These variables can be set in the user's shell, but they will then apply to all binaries executed in this shell, which is usually not desired.
- The size requested will automatically be rounded up to the huge page size.

9.6.6 Manipulating Shared Memory System Tunables

Hugepage `shmget()` requests can fail due to limits placed on process shared memory by 4 system tunables:

- **shmmax**: Limit on shared memory size for a single process, in bytes.
- **shmall**: System-wide limit on total shared memory "pages"; in this context a "page" is 4k.
- **shmmin**: Minimum size of a shared memory segment.
- **shmmni**: System-wide maximum number of shared memory segments.

Low shmmax usually the cause of `shmget()` failure. It should be adjusted to the max size required by a single process on the system. For example, to allow any process to have 256MB of shared memory:

- `echo 268435456 > /proc/sys/kernel/shmmax`

shmall will rarely require change; the others need modification extremely rarely.

9.6.6.1 Allocating hugepages for Heap Memory

Link application with libhugetlbfs to use hugepages for all malloc() calls and the C++ new operator

- Requires no application modification.
- With dynamic linking, can use an existing binary.
- Any available and mounted hugepage size can be used via the HUGETLB_MORECORE environment variable.

All malloc()calls and the C++ new operator will use hugepages provided by kernel.

- C library malloc() code is the same as without libhugepages; the underlying interface to kernel (known as morecore()) is what is changed by libhugetlbfs
- Total amount of hugepages is limited by system configuration
- malloc() or new requests smaller than the hugepage size will be rounded up by the kernel; multiple small requests can end up being mapped by a single hugepage.

9.6.6.2 Invoking Applications with Huge Heap (mallock()/new)

To use the default hugepage size:

- `LD_PRELOAD=libhugetlbfs.so`
- `HUGETLB_MORECORE=yes` [your app command line]

To use a different hugepage size:

- `LD_PRELOAD=libhugetlbfs.so`
- `HUGETLB_MORECORE=16M` [your app command line]

If the libhugetlbfs library is not installed in the normal system library directory, use LD_LIBRARY_PATH to specify the location::

- `LD_PRELOAD=libhugetlbfs.so`
- `LD_LIBRARY_PATH=~/mylibdir`

- `HUGETLB MORECORE=yes` [your app command line]

9.6.6.3 Moving the Heap for Hugepage Allocations

By default, the heap is located after the program image and can continue until it runs into a mmap. The program image typically resides at text/data/bss. Addresses for mmap are normally chosen by the kernel, starting from TASK_UNMAPPED_BASE. The result is a large portion of address space, sometimes larger than the default heap, unused. The HUGETLB_MORECORE_HEAPBASE option can be used to start the heap after the mmap area, rather than before it, so as to provide more space for the heap.

There may not be enough address space at the default heap location for hugepage malloc() and new requests to succeed.

- The 32-bit kernel uses `0x48000000` as TASK_UNMAPPED_BASE, which causes failures when trying to map > about 832MB of memory

The library provides HUGETLB_MORECORE_HEAPBASE to allow the user to specify the heap location.

- Use pmap or set HUGETLB_DEBUG=yes to get more information on the process' heap allocations
- Find a hole in the map large enough for your allocation and set HUGETLB_MORECORE_HEAPBASE to this when you run.

Example:

- LD_PRELOAD=libhugetlbfs.so HUGETLB_MORECORE=yes HUGETLB_MORECORE_HEAPBASE=`0x4C000000` [your app command line]



Figure 229. Heap Example:

9.6.7 Using Hugepages for .text, .data, and/or .bss (gnu ld 2.17+)

Specify –hugetlbfs-align option to GNU ld.

- Requires no application modification.
- Application needs to be relinked.
- libhugetlbfs wrapper script and linker script must be installed.
- Quite flexible – can specify exactly which segment types should use hugepages.
- Causes segments to be properly aligned for hugepage use.

Hugeedit utility

- Change the default segment-specific behavior without relinking the program.
- Default behavior is no hugepage mapping.
- The settings are used when HUGETLB_ELFMAP is not specified.

Run application with HUGETLB_ELFMAP.

Table 223. Environment variable that controls which segments use hugepages

Variable	Description
HUGETLB_ELFMAP=R	Read-only segments (text)
HUGETLB_ELFMAP=W	`Writable segments (data/BSS)`
HUGETLB_ELFMAP=RW	All segments (text/data/BSS)
HUGETLB_ELFMAP=no	`No segments`

The HUGETLB_ELFMAP application can specify specific page sizes for the different segment types:

- HUGETLB_ELFMAP=[R[=[pagesize]]:[W[=[pagesize]]]

Example: program invocation (dynamic link):

- LD_PRELOAD=libhugetlbfs.so HUGETLB_ELFMAP=R=4M:W=16M [your app command line]

If you are **not** using HUGETLB_ELFMAP:

- The default settings for the binary (set via hugeedit) will be used.
- The system-wide hugepage size will be used.

9.6.8 Performing mmap on a Hugepage-backed File

The option exists for an alternative to hugetlb-backed memory. This option is appropriate for these cases:

- To have some things hugetlb-backed without making the heap huge-tlb backed.
- To Share huge-tlb backed memory.
- To make use of address space above TASK_UNMAPPED_BASE without moving heap and thereby losing use of the address space above TASK_UNMAPPED_BASE.

Create and mmap() files on the hugetlbfs mount point.

- Can use any mounted huge page size.
- The huge page size will be the page size supported by the mount point.
- Files on the hugetlb file system are not regular files – they represent memory. You cannot copy a regular file onto the hugetlb file system mount point.
- Do not specify MAP_HUGETLB for this type of mapping – files on the hugetlb mount point are automatically backed by hugepage memory
- Files on a hugetlb mount point should be created as a multiple of the huge page size for that particular mount point; otherwise mmap() fails.

9.6.8.1 Exercising the mmap() File Example

The following example demonstrates a 4M hugetlbfs mount point.

```
#define LENGTH 4*1024*1024
#define FILE_NAME "/var/lib/hugetlbfs/global/pagesize-4MB/my_hugefile"

/* Open/create the file on the 4m mountpoint */
fd = open(FILE_NAME, O_CREAT | O_RDWR, 1777);
if (fd < 0) exit(1);
```

```
/* map in the file and check that the mmap succeeded */
addr = mmap(NULL, LENGTH, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (addr == MAP_FAILED) {
perror("mmap");
unlink(FILE_NAME);
exit(1);

}
/* unmap file, check for error */
if (munmap(addr, LENGTH) < 0) {
perror("munmap");
exit(2);

}
close(fd);
unlink(FILE NAME);
```

9.6.8.2 Requesting Anonymous mmap() with Hugepages

The user can directly request hugepages with MAP_HUGETLB flag to anonymous (no file descriptor specified) mmap

- Libhugetlbfs is not required.
- This method can only use the default hugepage size
- No mount points required.

Example:

```
mmap(NULL, 4*1024*1024, PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_HUGETLB | MAP_SHARED, 0, 0);
```

NOTE

MAP_ANONYMOUS and MAP_HUGETLB are both required

9.6.8.3 Using HugeTLB mmap(): Arguments and Error Checking

If specified, the address argument passed to mmap() must be aligned to the huge page size.

- Address may be NULL; the system will pick one for you.

For non-anonymous mmap() calls, the "offset" should be a multiple of the huge page size, if specified as non-zero.

The length of a mmap() should be a multiple of the huge page size.

- mmap() calls that use a fd (i.e. non-anonymous) must follow this rule.
- This is not required for anonymous mmaps, but is recommended to avoid munmap() problems.

Always check mmap() return codes.

9.6.8.4 Using HugeTLB munmap(): Arguments and Error Checking

Exercise caution with this process step, munmap() calls must specify a length that is a multiple of the huge page size.

- This is not optional.
- The munmap() will fail if this rule is broken.

Programs should always check the return code of munmap().

- It is common coding practice to check the return code from mmap() but not from munmap(). This is incorrect. Check the return code of both mmap() and munmap() to properly catch errors.

9.6.9 Running the Test

After two boards boot up, login linux and run following commands:

Table 224. sRIO Test Procedure

Memory Allocation Type	Code Mod	Reuse Binary	Huge Page Size	Need Lib?	Notes
heap (malloc()/ new)	No	Yes*	Any>	Yes	* If dynamically linked.
anonymous (mmap()/new)	Yes	No	Default Only>	No	
file mmap()	Yes	No	Any*>	Yes	* Page size used is that supported by the mount
shared memory (shmget() with SHM_HUGETLB)	Yes	No	Default Only	No	
shared memory (shmget() without SHM_HUGETLB)	No	Yes*	Default Only	Yes	*If dynamically linked.
.text, .data, .bss	No	No	Any	Yes	

9.6.9.1 When to Use Hugepages

In order to calculate whether a program will benefit from using huge pages, count the L2 MMU miss rate using core performance monitors.

- If (# L2 MMU misses * 200 cycles) represents a large percentage of the program's total number of cycles, it is possibly a good candidate for hugepages

Use "top" or "ps" on a running process to determine its resource usage.

- For a long-running process, `/usr/bin/time -v` can be used to get the "Maximum resident set size", but it can return values that are 4x the actual usage.

Look at pattern of mmap()calls in a program trace.

- Large calls lend themselves more easily to using hugepages. Programs with lots of small calls can also benefit from code change to consolidate the calls into a multiple of the huge page size.
- Also look at munmap() calls – the program must be able to munmap() in multiples of the huge page size.

Generate an instruction trace of the program and look at load/store addresses

- If the top 20 bits of load/store addresses change often and have more than 512 distinct values, then the program may be using a lot of data and bss.
- Also see if the same is true for instruction addresses

Programs with malloc() or new requests whose sum is large may benefit.

- When the library sees one of these requests it will allocate a large heap region that is the size of the hugepage specified by the HUGETLB_MORECORE environment variable.
- Future malloc() or new requests will allocate from that same region until it is fully utliized.
- The smallest hugepage that encompasses all the requests should be used when possible.

Smaller heap requests can also benefit.

- If the number of hugepages needed in the system is smaller than the number of entries in TLB1, the application may see fewer TLB misses with a hugepage.

9.6.10 Matching Hugetlb Methods with Program Characteristics

Table 225. sRIO Protocol Efficiency

Program Characteristic	HugeTLB usage
Large `malloc()` or new requests.	HUGETLB_MORECORE = [hugepage size]
Large program segments (.text, hugetlbfs align .data, .bss)	--hugetlbfs-align, HUGETLB_ELFMAP=[specify segments/size]
Large MAP_ANONYMOUS `mmap()`	`mmap()` on hugetlb mount point.
Large shared memory	SHM_HUGETLB or link with library and specify HUGETLB_SHM.

9.6.11 Using Multiple Types of Hugepage Allocation Methods

HugeTLB allocation methods are not mutually exclusive.

- Example: HUGETLB_MORECORE + HUGETLB_ELFMAP
- Example: shmget(SHM_HUGETLB) + HUGETLB_MORECORE

All combinations are allowed; but care should be taken.

- Poor choice of options can result in less-than optimal hugepage allocations.
- Can run out of hugepages in the system.
- TLB1 can end up oversubscribed.
- Can waste process virtual address space and have failed program allocations.

9.6.12 Hugetlb Benefits and Limitations

Benefits:

- Fewer TLB misses in large processes.
- Other processes can benefit because their 4k pages aren't getting evicted by memory hog processes.
- Potential for significant performance gains.

Limitations:

- Limited number of TLB1 entries.
- Adds minor latency to critical kernel paths for 4k pages.
- Hugepages are more costly for the kernel to manage.
- Allocating hugepages removes them from the available memory pool, causing less memory to be available.

9.6.13 Understand the Differences Between Applications

Some applications convert easily to hugepages. In many cases, only slight modifications, or none at all, are required. Other applications may benefit, but with more work:

- Some huge page usage models can only operate on multiples of a huge page size.
- Applications that map and unmap large amounts of memory using a lot of small mmap() and/or munmap() calls will likely require significant modifications in order to use hugepages efficiently.
- Code that doesn't properly check system call return codes may have unexpected fails later in the program execution.

There are programs that will not benefit from huge pages. Either the memory footprint is too small or it may be the case that a low percentage of cycles are spent in TLB miss handling.

9.6.14 HugeTLB Status and Support

The current status of HugeTLB features and future goals for this effort includes these items:

- Some huge page usage models can only operate on multiples of a huge page size.
- Applications that map and unmap large amounts of memory using a lot of small mmap() and/or munmap() calls will likely require significant modifications in order to use hugepages efficiently.
- Code that doesn't properly check system call return codes may have unexpected fails later in the program execution.

There are programs that will not benefit from huge pages. Either the memory footprint is too small or it may be the case that a low percentage of cycles are spent in TLB miss handling.

9.6.15 HugeTLB Resources

4-part LWN.net series on HugeTLB

Running Applications with Libhugetlbfs

Libhugetlbfs HOWTO:

NOTE

These supporting documents don't have specifics for FSL Power Architecture such as page size but are included for general understanding.

9.7 OpenSSL Offload User's Guide

9.7.1 Overview

The Secure Socket Layer (SSL) protocol is the most widely deployed application protocol to protect data during transmission by encrypting the data using popular cipher algorithms such as AES, DES and 3DES.

Apart from encryption it also provides message authentication services using popular hash/digest algorithms such as SHA1 and MD5. SSL is widely used in application web servers (HTTP) and other applications such as SMTP POP3, IMAP, Proxy servers etc., where protection of data in transit is essential for data integrity.

There are various version of SSL protocol such as SSLv3, TLSv1.0, TLSv1.1, TLSv1.2, TLSv1.3 and DTLS (Datagram TLS). Of all the SSL protocol versions, TLSv1.0 and SSLv3 are in commonly in use, the other versions seeing more adoption as well.

This document introduces NXP SSL acceleration solution on QorIQ platforms using OpenSSL:

1. OpenSSL software architecture
2. Building OpenSSL with hardware offload support
3. Examples of OpenSSL Offloading

9.7.1.1 OpenSSL Software architecture

The SSL protocol is implemented as a library in OpenSSL - the most popular library distribution in Linux and BSD systems. The OpenSSL library has several sub-components such as:

1. SSL protocol library
2. Crypto library (Symmetric and Asymmetric cipher support, digest support etc.)
3. Certificate Management

The following figure presents the general interconnect architecture for OpenSSL. Each relevant layer is represented with a clear separation between Linux User Space and Linux Kernel Space.



Figure 230. OpenSSL interface with Linux kernel

9.7.1.1.1 OpenSSL's ENGINE Interface

OpenSSL Crypto library provides Symmetric and Asymmetric (PKI) cipher support that is used in a wide variety of applications such as OpenSSH, OpenVPN, PGP, IKE, XML-SEC etc. The OpenSSL Crypto library provides software support for:

1. Cipher algorithms
2. Digest algorithms
3. Random number generation
4. Public Key Infrastructure

Apart from the software support, the OpenSSL can offload these functions to hardware accelerators via the ENGINE interface. The ENGINE interface provides callback hooks that integrate hardware accelerators with the crypto library. The callback hooks provide the glue logic to interface with the hardware accelerators. Generic offloading of cipher and digests algorithms through Linux kernel is possible with cryptodev engine.

9.7.1.1.2 NXP solution for OpenSSL hardware offloading

The following layers can be observed in NXP's solution for OpenSSL hardware offloading:

- OpenSSL (user space) - implements the SSL protocol
- cryptodev-engine (user space) - implements the OpenSSL ENGINE interface; talks to cryptodev-linux (/dev/crypto) via ioctls, offloading cryptographic operations in kernel
- cryptodev-linux (kernel space) - Linux module that translates ioctl requests from cryptodev-engine into calls to Linux Crypto API
- Linux Crypto API (kernel space) - Linux kernel crypto abstraction layer
- CAAM driver (kernel space) - Linux device driver for the CAAM (Cryptographic Acceleration and Assurance Module) crypto engine

The following are offloaded in hardware in current SDK:

- protocols: TLS v1.0
- cipher modes:
  - one-shot (a single ioctl for both encryption and authentication): AES128-SHA, AES256-SHA
  - two-shot (two ioctls - one for encryption, the other for authentication): all combinations of AES with SHA1, all combinations of DES and 3DES with SHA1

9.7.2 Building OpenSSL with hardware offloading support

1. Build the `fsl-image-core` rootfs for P4080DS (similar for other platforms); this will automatically deploy OpenSSL with cryptodev-engine support:

```
$ bitbake fsl-image-core
```

2. Boot the board and load the cryptodev kernel module:

```
root@p4080ds:~# modprobe cryptodev
cryptodev: driver 1.8 loaded.
```

3. Check that OpenSSL detected cryptodev:

```
root@p4080ds:~# openssl engine
(cryptodev) BSD cryptodev engine
(dynamic) Dynamic engine loading support
```

9.7.3 Nginx offloading with OpenSSL

In this section, a client-server example will be presented. There are two options of testing and validating the OpenSSL TLS 1.0 Offloading:

- Web Server running on the NXP QorIQ Board and client (e.g. HTTPS browser) on another equipment
- Client running on the NXP QorIQ board and the web server on another equipment

The examples below use "nginx" web server and "openssl s_client" utility as client.

Building a custom OpenSSL and a web server

Both openssl and nginx are build by default with NXP SDK full image. These packages can be added to other images with minor modifications of he image recipes. For example, to add nginx package to the fsl-minimal image add this line to fsl-image-minimal.bb file:

```
IMAGE_INSTALL += "nginx"
```

Nginx does not use any openssl engines by default. Cryptodev must be explicitely mentioned in nginx configuration file to enable hardware offloading:

`/etc/nginx/nginx.conf:`

```
ssl_engine cryptodev;
worker_processes 4;
worker_cpu_affinity 0001 0010 0100 1000; #for 4 Core CPU; For 2 Core CPU worker_cpu_affinity 01
10;
...
    # HTTPS server
    #
    server {
        listen       443;
        server_name  localhost;

        ssl                  on;
        ssl_certificate      server.crt;
        ssl_certificate_key  server.key;
        ssl_session_timeout  5m;
        ssl_protocols  TLSv1;
        ssl_ciphers AES128-SHA:AES256-SHA;
        ssl_prefer_server_ciphers   on;

        location / {
            root   /var/www/localhost/html;
            index  index.html index.htm;
        }
    }
...
```

Worker processes and affinity should be set according to the number of CPU cores available on the platform. Refer to nginx documentation for more details.

Insert cryptodev module and run basic checks

Cryptodev must be probed into the kernel before running crypto acceleration code. Both nginx and s_client will notice cryptodev presence only during their initialization phase.

```
$ modprobe cryptodev
cryptodev: driver 1.8 loaded.
```

```
$ openssl version
OpenSSL (...)
$ openssl engine
(cryptodev) BSD cryptodev engine
(dynamic) Dynamic engine loading support
```

If cryptodev driver is not loaded, openssl will report only dynamic engine support and all operations will be done in software.

Linux kernel can check and report ciphers availability with the help of tcrypt module. After probing tcrypt, crypto algorithms will be listed in /proc/crypto:

```
$ modprobe tcrypt
$ grep tls /proc/crypto
name         : tls10(hmac(sha1),cbc(aes))
driver       : tls10-hmac-sha1-cbc-aes-caam
```

Offloading operations can be monitored with the interrupt counters for CAAM JR and QI interfaces:

```
root@p4080ds:~# cat /proc/interrupts | grep -i qman
108: 0 0 0 0 0 0 0 7508   OpenPIC   108 Level     QMan portal 7
110: 0 0 0 0 0 0 7524 0   OpenPIC   110 Level     QMan portal 6
112: 0 0 0 0 0 7542 0 0   OpenPIC   112 Level     QMan portal 5
114: 0 0 0 0 7565 0 0 0   OpenPIC   114 Level     QMan portal 4
116: 0 0 0 7576 0 0 0 0   OpenPIC   116 Level     QMan portal 3
118: 0 0 7524 0 0 0 0 0   OpenPIC   118 Level     QMan portal 2
120: 0 7535 0 0 0 0 0 0   OpenPIC   120 Level     QMan portal 1
122: 7521 0 0 0 0 0 0 0   OpenPIC   122 Level     QMan portal 0
470: 0 0 0 0 0 0 0 0      OpenPIC   2006 Edge      qman-err
```

The interrupt counters are also incremented on networking operations. Further analysis is required to understand the source of their modification. Some algorithms implemented in CAAM use only the JR interface. For example, these are the JR counters after "openssl speed aes-128-cbc" when using cryptodev:

```
root@p4080ds:~# cat /proc/interrupts | grep jr
 88: 0 0 0 0 26 0 0 0   OpenPIC    88 Level     ffe301000.jr
 89: 0 0 0 0 0 1117204 0 0   OpenPIC    89 Level     ffe302000.jr
 90: 0 0 0 0 0 0 24 0   OpenPIC    90 Level     ffe303000.jr
 91: 0 0 0 0 0 0 0 24   OpenPIC    91 Level     ffe304000.jr
```

Testing TLS

To verify TLS record offload, the minimum setup requires a testing board and a web server with support for TLS1.0, either nginx, apache, lighttpd. We will run the SSL client on the NXP QorIQ board and and download one 100MB file over the network from a web-server.



Figure 231. Test setup

Openssl s_client command will be used on the NXP board to make the connection with the server:

```
$ openssl s_client
```

The command can be scripted and run without further intervention:

```
$ echo GET /index.html | openssl s_client -connect <server_ip>:443 -cipher AES128-SHA -tls1 -quiet
```

The option "-quiet" can be removed to see more details about the TLS session.

OpenSSL will transparently use HW acceleration if cryptodev module is loaded in the kernel. If cryptodev module is removed, openssl will not initialize cryptodev engine and will use software implementation.

9.7.4 OpenSSH offloading with OpenSSL

This section describes the steps to enable OpenSSH offloading:

- Configure OpenSSH for QorIQ devices
- Test OpenSSH with OpenSSL Acceleration

Configure OpenSSH for QorIQ devices

By default the SSH server used in NXP images is Dropbear. Dropbear does not have support for hardware acceleration of cryptographic operations. For this reason, a customized QorIQ image have to be built in which we replace Dropbear with OpenSSH. Add the following code to the bitbake file for your image of choice (e.g. fsl-image-minimal.bb):

```
DEPLOY_PKGS = " \
    fm-ucode \
    hv-cfg \
    rcw \
    hypervisor \
    hypervisor-partman \
    openssl \
    openssh \
    openssh-ssh \
    openssh-sftp-server \
    openssh-sshd \
"
IMAGE_INSTALL_append = " ${DEPLOY_PKGS}"
```

```
IMAGE_INSTALL_append = " cryptodev-linux"
IMAGE_INSTALL_append = " cryptodev-module"
```

The bitbake recipe for OpenSSH and the server configuration file must be modified as well:

poky/meta/recipes-connectivity/openssh/openssh_7.1p1.bb

```
EXTRA_OECONF = "'LOGIN_PROGRAM=${base_bindir}/login' \
                ${@base_contains('DISTRO_FEATURES', 'pam', '--with-pam', '--without-pam', d)} \
                --without-zlib-version-check \
                --with-privsep-path=/var/run/sshd \
                --sysconfdir=${sysconfdir}/ssh \
                --with-xauth=/usr/bin/xauth \
                --with-ssl-engine"
```

poky/meta/recipes-connectivity/openssh/openssh/sshd_config

```
Protocol 2
Ciphers aes128-cbc,aes256-cbc
MACs hmac-sha1
```

After completing these steps the customized boot image can be created:

```
$ bitbake fsl-image-minimal
```

Test OpenSSH with OpenSSL Acceleration



Figure 232. OpenSSH Test setup

Operations to be executed on the QorIQ device:

```
root@p4080ds:~# modprobe cryptodev
cryptodev: driver 1.8 loaded.
root@p4080ds:~# openssl version
OpenSSL (...)
root@p4080ds:~# openssl engine
(cryptodev) BSD cryptodev engine
(dynamic) Dynamic engine loading support
```

If cryptodev driver is not correctly loaded, OpenSSH will not be able to use the hardware acceleration. Cryptographic operations will be executed using the OpenSSL software implementation.

Start the OpenSSH server on the QorIQ device after loading cryptodev kernel module. The OpenSSH server will automatically use cryptodev if available, without any further configuration:

```
root@p4080ds:~# /usr/sbin/sshd -f /etc/ssh/sshd_config
```

Test connectivity between computer and board.

```
[user@host temp] ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.234 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.104 ms
```

Initial connection to QorIQ Board IP 192.168.1.2.

```
[user@host temp] ssh root@192.168.1.2
The authenticity of host '192.168.1.2 (192.168.1.2)' can't be established.
ECDSA key fingerprint is dc:83:0d:91:d8:e4:bb:f4:76:66:54:6d:52:03:c7:a3.
Are you sure you want to continue connecting (yes/no)?yes
```

Create a test file for SCP on SSH client and transfer the file to board:

```
[user@host temp] dd if=/dev/zero of=file.out bs=1MB count=500
500+0 records in
500+0 records out
500000000 bytes (500 MB) copied, 11.9799 s, 41.7 MB/s
[user@host temp]scp file.out root@192.168.1.2:/var/volatile
file.out            100%  477MB  68.1MB/s   00:07
```

Special Case: If the board gets rebooted, the server key will change. For this reason you will have to delete the initial key that have been stored on the client. Delete the line and restart the SSH Connection:

```
[user@host temp] vi .ssh/known_hosts
[...]
[user@host temp] ssh root@192.168.1.2
The authenticity of host '192.168.1.2 (192.168.1.2)' can't be established.
ECDSA key fingerprint is dc:83:0d:91:d8:e4:bb:f4:76:66:54:6d:52:03:ff:ff.
Are you sure you want to continue connecting (yes/no)?yes
root@p4080ds:~#
```

Offloading operations can be monitored with the interrupt counters for CAAM JR and QI interfaces:

```
root@p4080ds:~# cat /proc/interrupts | grep -i qman
108: 0 0 0 0 0 0 0 7508   OpenPIC   108 Level     QMan portal 7
110: 0 0 0 0 0 0 7524 0   OpenPIC   110 Level     QMan portal 6
112: 0 0 0 0 0 7542 0 0   OpenPIC   112 Level     QMan portal 5
114: 0 0 0 0 7565 0 0 0   OpenPIC   114 Level     QMan portal 4
116: 0 0 0 7576 0 0 0 0   OpenPIC   116 Level     QMan portal 3
118: 0 0 7524 0 0 0 0 0   OpenPIC   118 Level     QMan portal 2
120: 0 7535 0 0 0 0 0 0   OpenPIC   120 Level     QMan portal 1
122: 7521 0 0 0 0 0 0 0   OpenPIC   122 Level     QMan portal 0
470: 0 0 0 0 0 0 0 0      OpenPIC   2006 Edge      qman-err
```

The interrupt counters are also incremented on networking operations. Further analysis is required to understand the source of their modification. Some algorithms implemented in CAAM use only the JR interface. For example, these are the JR counters after "openssl speed aes-128-cbc" when using cryptodev:

```
root@p4080ds:~# cat /proc/interrupts | grep jr
88: 0 0 0 0 26 0 0 0           OpenPIC    88 Level     ffe301000.jr
89: 0 0 0 0 0 1117204 0 0   OpenPIC    89 Level     ffe302000.jr
90: 0 0 0 0 0 0 24 0           OpenPIC    90 Level     ffe303000.jr
91: 0 0 0 0 0 0 0 24           OpenPIC    91 Level     ffe304000.jr
```

9.7.5 TLS Ciphersuites vs TLS protocol version

Table 226. OpenSSL CipherSuite Compatibility

CipherSuite	TLS Protocol Version
SSL_RSA_WITH_NULL_MD5	SSL3.0
SSL_RSA_WITH_NULL_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_RC4_40_MD5	SSL3.0
SSL_RSA_WITH_RC4_128_MD5	SSL3.0
SSL_RSA_WITH_RC4_128_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	SSL3.0
SSL_RSA_WITH_IDEA_CBC_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_DSS_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DHE_DSS_WITH_DES_CBC_SHA	SSL3.0
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DHE_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	SSL3.0

Table continues on the next page...

Table 226. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
SSL_DH_anon_WITH_RC4_128_MD5	SSL3.0
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_anon_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_NULL_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA	SSL3.0
TLS_RSA_WITH_NULL_MD5	TLS1.0
TLS_RSA_WITH_NULL_SHA	TLS1.0
TLS_RSA_EXPORT_WITH_RC4_40_MD5	TLS1.0
TLS_RSA_WITH_RC4_128_MD5	TLS1.0
TLS_RSA_WITH_RC4_128_SHA	TLS1.0
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	TLS1.0
TLS_RSA_WITH_IDEA_CBC_SHA	TLS1.0
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_DES_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	TLS1.0
TLS_DH_anon_WITH_RC4_128_MD5	TLS1.0

Table continues on the next page...

Table 226. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_AES_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_SEED_CBC_SHA	TLS1.0

Table continues on the next page...

Table 226. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_DHE_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_SEED_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_NULL_SHA	TLS1.0
TLS_ECDH_RSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_NULL_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_NULL_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_NULL_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_NULL_SHA	TLS1.0
TLS_ECDH_anon_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_AES_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_NULL_SHA256	TLS1.2
TLS_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_RSA_WITH_AES_128_GCM_SHA256	TLS1.2

Table continues on the next page...

Table 226. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS1.2

Table continues on the next page...

Table 226. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_anon_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_256_GCM_SHA384	TLS1.2

9.8 USDPAA

9.8.1 USDPAA User Guide

9.8.1.1 Introduction

The NXP Data Path Acceleration Architecture comprises a set of hardware components which are integrated via a hardware queue manager and use a common hardware buffer manager. Software accesses the DPAA via hardware components called "software portals". These directly provide queue and buffer manager operations such as enqueues, dequeues, buffer allocations, and buffer releases and indirectly provide access to all of the other DPAA hardware components via the queue manager.

This document describes the User Space Datapath Acceleration Architecture (USDPAA) software. USDPAA is a software framework that permits Linux user space applications to directly access the DPAA queue and buffer manager software portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

9.8.1.1.1 Intended audience

This document is intended for software developers and system architects who work with the USDPAA framework.

The material is technical in nature. The reader is assumed to be familiar with

- General Linux software development, operation, and configuration-- for Power architecture devices in particular.
- The concept of device drivers and their role in operating systems.
- Linux network administration and use.
- The NXP Linux SDK for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.

The P4080 was the first NXP SoC to incorporate the DPAA. As such, it is used in many examples. However, USDPAA is intended to apply in the same manner to all DPAA-based SoCs.

9.8.1.1.2 USDPAA overview

As mentioned above, USDPAA is an environment that allows the development of Linux user space applications that have direct access to software portals for the DPAA buffer manager (BMan) and the DPAA queue manager (QMan).

Software portals are memory mapped hardware components. The items within them (message ring, dequeue ring, etc.) have specific addresses in the SoC's physical address map just like regular memory does. USDPAA works by permitting the physical address ranges that correspond to software portals to be mapped into the virtual (technically "effective" in the Power architecture nomenclature) address space of user space processes.Thus, the user space application can use normal load and store instructions to perform operations on the portals.

Portals must be accessed using correct instruction sequences, and also the memory range for the portals must be mapped using the correct cache attributes. For this reason, the USDPAA software includes both the user space driver for the portals and also an access and control API packaged as a user space library.



Figure 233. C-Language APIs access memory-mapped devices

The USDPAA framework assumes the context of a single SMP Linux instance on an SoC such as the P4080 . There is no dependence on the NXP Embedded Hypervisor, and this document assumes that it is not used.

Specifically, the SMP Linux instance may contain many user spaces processes. These processes can be (and normally are) multi-threaded using the pthreads facility of Linux. Define a "USDPAA thread" as a thread that has been allocated a BMan and a QMan software portal for its use via direct access. A "USDPAA process" is a Linux user space process that contains at least one USDPAA thread.

Clearly, the number of USDPAA threads is limited by the number of software portals that the particular SoC provides. The P4080 provides 10 QMan and 10 BMan software portals. The available portals must be divided into two sets: portals for use by the Linux kernel and portals for use by USDPAA threads.

9.8.1.1.3 USDPAA and legacy Linux software

This section describes the relationship between USDPAA and various types of legacy software.

9.8.1.1.3.1 Legacy user space applications

USDPAA provides user space processes access to DPAA functionality via user space device drivers and device-oriented C-language APIs layered upon them. Legacy applications must be modified in order to use these APIs. USDPAA's primary goal is to provide these APIs.

As an example, it is possible to offload cryptographic operations to the DPAA Security Engine (SEC) using USDPAA drivers, but this does not imply that all applications that use a cryptography-related legacy Linux facility are automatically accelerated. Software someplace (in user space) must explicitly use the USDPAA functions.

It is possible that some legacy user space facilities could be accelerated in a manner that automatically applies to legacy user space applications if means other than or in addition to USDPAA are used. This topic is, however, beyond the scope of the document.

9.8.1.1.3.2 Relationship to conventional kernel-based drivers

Conventional device drivers reside in the Linux kernel and are accessed via system calls such as read and write. These often (but not always) involve copying data between the application's address space and the kernel's address space. These types of drivers are desirable in many situations.

The existence of USDPAA neither implies nor excludes the existence of conventional drivers. It also implies nothing about whether or not conventional drivers copy data. Mostly, the topic of conventional drivers is beyond the scope of this document. Details will vary among DPAA IP blocks. A few points follow:

- SEC has a job queue interface that can be used concurrently with its QMan interface. The DPAA SDK contains kernel code that does this.
- As figure QMan driver overview on page 1481 shows, The DPAA's Buffer Manager and Queue Manager in-kernel portal drivers support access by multiple entities such as the DPAA kernel ethernet driver. It is possible to create custom conventional kernel drivers that layer on top of the in-kernel portal drivers. These can provide traditional kernel-mediated device access.

9.8.1.2 USDPAA assumptions and use cases

The primary purpose of USDPAA is application performance. User space drivers can provide substantial performance improvement over traditional kernel-mediated access to devices.

- No system calls are needed to do I/O.
- This implies no need to switch into and out of the kernel's execution context.
- User space applications directly access data buffers, thus providing zero copy I/O in all cases.

Because the goal is performance, USDPAA provides simple but low-level access to I/O functionality. It does not employ complex and costly abstractions. Applications deal directly with QMan and BMan via their software APIs. These form a thin but helpful layer between the application and the hardware.

9.8.1.2.1 Assumptions

The USDPAA software makes certain assumptions and supports multiple use-cases. These are detailed in the sections that follow.

9.8.1.2.1.1 General assumptions

- USDPAA threads should be made affine to a core. This is due to support for stashing to per-core caches. (Nothing breaks if this is not the case, but performance will be sub-optimal.)
- Every USDPAA thread has its own BMan and QMan software portals. No other thread or entity accesses these portals. Within the drivers, these are stored as thread-local variables and the locking assumptions depend on them being thread-private.
- USDPAA threads may make use of arbitrary Linux system calls. For example, they can do file I/O.
- USDPAA threads are compatible with general Linux services such as debuggers like gdb.
- Threading is via standard Linux pthreads and the standard Linux GNU C library.

9.8.1.2.2 Use cases

USDPAA is intended to support multiple use-cases, and the supported use-case set will grow with time.

9.8.1.2.2.1 Run-to-completion

The full run-to-completion use case is defined by the following characteristics:

- The number of USDPAA threads per core must not exceed the number of QMan and BMan portals assigned to that core (and reserved for USDPAA use) within the device-tree. Note however that testing with more than one thread per core has been minimal. Not all cores need have a USDPAA thread, however.
- Cores hosting USDPAA threads may be configured to run nothing in user space other than that core's USDPAA thread. For example the kernel parameter "isolcpus" could be used - see CPU isolation on page 1485 for more on this topic.
- The /proc/irq mechanism may be used to limit interrupts for miscellaneous peripherals to non-isolated cores. Again, USDPAA does not require this architecturally, but it is often done.
- In run-to-completion, USDPAA threads poll their portals. Polling generally implies that the USDPAA threads will always be running or ready to run. It would be unusual to have other threads scheduled on the same core when USDPAA threads are in this "non-interactive" mode.

- Often, one thinks of the USDPAA threads as "workers" able to process any form of "work" delivered via QMan messages. In this model, the QMan scheduler can be used as a general "work" scheduler rather than relying on the Linux scheduler for this purpose.

9.8.1.2.2.2 Interrupt-driven

USDPAA supports an interrupt-driven model that allows benefits such as cooperating with other threads on the same core, potential power-saving by not polling all of the time, and so on. This model introduces more operating system overheads and thus trades performance and latency for these benefits.

- The user space drivers for BMan and QMan software portals are implemented using a Linux character device. It permits USDPAA threads to await interrupts from software portals by doing file operations (like "select()") using a file descriptor associated with the user space device. See section QMan and BMan drivers and C API. on page 1481 for more details.
- USDPAA threads process dequeued frames from portals (as much as they like) after an interrupt indicates that data are available. When dequeue processing is complete, the thread re-enables the interrupt.
- This interrupt mechanism allows USDPAA threads to sleep awaiting I/O. Thus, other threads can execute while a particular USDPAA thread sleeps. In this model, it is normal it have multiple threads, USDPAA or otherwise, execute on the same core.
- This use-case is compatible with SCHED_FIFO Linux scheduling. USDPAA is independent of scheduling policy. It is also independent of real-time policy and patches such as PREEMPT_RT.

Application developers may chose the model, run-to-completion or interrupt-driven that best fits their need. The PPAC-based example applications in USDPAA (eg. reflector) implement a hybrid of both modes, where an interrupt-driven mode is used whenever the packet-processing has been idle for a short period of time, and switching back to run-to-completion once processing resumes.

9.8.1.3 USDPAA components

USDPAA consists of several components which may used in the context of standard SMP Linux on the NXP family of DPAA-based SoCs. These components are described below.

9.8.1.3.1 Device-tree handling

Many configuration and resource details are defined within the "device-tree" used to boot Linux. The kernel itself uses this data structure to determine the system's devices and attributes, and USDPAA applications are dependent on the same information. As will be seen, the QMan and BMan portals that are available to USDPAA (and their attributes such as channel IDs) are declared within the device-tree, as are the ethernet interfaces and related attributes from the FMan side of things.

The USDPAA "of" driver[16] parses the device-tree information from the procfs filesystem exported by the Linux kernel (as found at /proc/device-tree/) and constructs an internal data-structure representation of the tree, including interrelationships between device-nodes (eg. portals and CPUs, pool-channels and portals, etc). This information is then used by driver and application-configuration layers where required to obtain device information.

9.8.1.3.1.1 Device-tree initialization requirements

In the current "of" driver implementation, it is the application's responsibility to initialize this driver layer prior to initializing or using any other driver layers that may be dependent on it. This is achieved by calling the of_init() API. To satisfy leak-checkers or to support the use of persistent/reusable processes, an application can also undo all allocations and driver state by calling of_finish().

The "Queue Manager, Buffer Manager API Reference Manual" cover these APIs in more detail, if the above description (and the <usdpaa/of.h> header) are insufficient.

[16] "OF" refers to Open Firmware, the specification behind device-trees, a subject beyond the scope of this document.

9.8.1.3.2 QMan and BMan drivers and C API.

The Queue Manager (QMan) and Buffer Manager (BMan) drivers are the heart of USDPAA. The two drivers are structurally similar in the broadest sense. This discussion will focus on QMan, but the material also applies to BMan.

9.8.1.3.2.1 QMan driver overview

The QMan driver serves two roles:

1. Initialization and management of all aspects of QMan that are not per-portal, i.e. are global.
2. Initialization and management of per software portal QMan operations, i.e. operations that are local to the system component to which the portal is dedicated.

It is reasonable to think of the QMan driver as being split into parts: a QMan global driver and QMan software portal driver. There is only one instance of the global driver per SoC. It is always a kernel-level driver. There is an instance of the software portal driver for every QMan software portal that exists physically on the SoC. For example, P4080 provides 10 QMan software portals. Thus, there are 10 instances of the portal driver available.



Figure 234. QMan Driver Architecture

As the figure above shows, the software portals (and hence their driver instances) may be dedicated for kernel or or user space use. Essentially the same QMan portal API is presented for both the kernel and the user space instances. Indeed, the same software is used in user space and the kernel.

The kernel driver instances have existing "users" in the kernel. For example, the ethernet driver, the PME driver and (though not in the current release) the SEC driver all use the QMan API to share access to the same physical portal on a core. They all enqueue and queue on the same portal using the API associated with the portal's driver instance. Kernel developers are free to add additional users in the form of additional kernel-level components that share the same portal.

As an aside, the SEC is a special case because it has two interfaces that software can use: the job queue interface, and the QMan interface. With current versions of the DPAA SDK, kernel software uses the job queue interface. User space processes can use the SEC via QMan using their portals.

User space processes access QMan using software portals that are dedicated to user space. To a great extent the QMan API abstracts it, but this access is implemented using standard Linux character devices for user space drivers. It provides /dev entries for devices to be accessed from user space. A user space process opens such a device and requests that the device be mapped directly into the process's address space. The assumption is that the physical device is a conventional memory mapped peripheral - as QMan software portals are. At this point, the process can access the device hardware via normal load and store instructions.

QMan's hardware design requires that portals be mapped with the correct caching attributes for each part of the portal. In addition, careful instruction sequences are needed to ensure that portal operations are carried out correctly. This is due to the hardware's cache stashing support, which provides low-latency access to portals and data dequeued from portals.

The QMan API mentioned above abstracts this and provides software with a convenient method to perform portal operations such as enqueues and dequeues. The QMan API in user space is a library that is layered on top of the USDPAA character device infrastructure. For the user space driver instances, there is

- In user space:
  - The QMan API
  - Direct access to the portal hardware
- In the kernel:
  - The character device code to establish mappings
  - Handle interrupts

This C-callable API is documented in the "Queue Manager, Buffer Manager API Reference Manual" that is distributed with the DPAA SDK software.

One difference in usage of portals in user space versus the kernel is that kernel portals are expected to be persistent. They are brought into service and remain in service so upper layer kernel services such as the ethernet driver can assume that they are there. User space portals are, in contrast, dedicated to an application thread and need to be in service only as long as that thread is running. Thus, the time during which the portal is in service is determined by the application and all uses of the portal are determined by the application.

9.8.1.3.2.2 QMan portals and the Linux device-tree

Since QMan software portals can be dedicated to either the Linux kernel or user space, there must be a mechanism to indicate how each portal will be used.

Linux device trees describe physical resources that are available to Linux and provide information that allows Linux to select drivers for those devices. As such, there are entries in the device tree for all QMan software portals.

The device tree property "fsl,usdpaa-portal" indicates that a given portal is to be made available to user space; the absence of this property implies that the portal will be used only within the Linux kernel.

A listing of two QMan software portal device tree nodes follows. The first portal is used by the kernel. The second is made available to user space.

```
qportal0: qman-portal@0 {
        cell-index = <0x0>;
        compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
        reg = <0x0 0x4000 0x100000 0x1000>;
        cpu-handle = <&cpu0>;
        interrupts = <104 0x2 0 0>;
        fsl,qman-channel-id = <0x0>;
        fsl,qman-pool-channels = <&qpool1 &qpool2 &qpool3>;
};
qportal1: qman-portal@4000 {
        cell-index = <0x1>;
        compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
        fsl,usdpaa-portal;
        reg = <0x4000 0x4000 0x101000 0x1000>;
        cpu-handle = <&cpu1>;
        interrupts = <106 0x2 0 0>;
        fsl,qman-channel-id = <0x1>;
        fsl,qman-pool-channels = <&qpool4 &qpool5 &qpool6
                                  &qpool7 &qpool8 &qpool9
                                  &qpool10 &qpool11 &qpool12
                                  &qpool13 &qpool14 &qpool15>;
};
```

There is no mechanism provided to determine what, if anything, user space software will do with a given user space portal. Initially, these portals are present but not initialized. Each user space portal has a /dev entry. It is a USDPAA driver decision as to which portals a given process or thread will use. A portal is placed into use when a user space process or thread requests it via the QMan API.

9.8.1.3.2.3 Note on the current implementation

In the current implementation, portals are bound to cores via the cpu-handle in the portal device tree node. Most commonly, a thread will make itself affine to its portal's core, because stashing for the portal will be targeting that core's cache. Functionality will not break if the thread executes on a different core, because coherency will be maintained, but at the expense of sub-optimal performance (and increased contention between the cores).

Expect more flexibility in this respect in later releases of the QMan software.

9.8.1.3.2.4 Portal initialization requirements

As described in Device-tree initialization requirements on page 1480, the USDPAA "of" driver must be initialised prior to trying to initialize or use QMan or BMan. Once this is done, a pthread that wishes to be initialized with access to a QMan portal should call the qman_thread_init() API, which is defined in the <usdpaa/fsl_usd.h> header along with all other USDPAA-specific QMan/BMan APIs.

Please consult the "Queue Manager, Buffer Manager API reference" for more information.

9.8.1.3.2.5 Buffer Manager (BMan)

The BMan driver and its software support is very similar to QMan's. BMan software portals may be used in the kernel or in user space just like QMan portals. Just as with QMan, the same BMan software API is available in both the kernel and in user space. This API also is defined in the "Queue Manager, Buffer Manager API Reference Manual."

9.8.1.3.2.6 Raw Portal APIs

Raw portal APIs are available to allow USDPAA process to allocate QMan and BMan portals on behalf of another processor. The portals allocated by these APIs are not configured, it is the responsibility of the allocator to do any configuration of the portal that may be needed. The APIs are as follows:

- int qman_allocate_raw_portal(struct usdpaa_raw_portal *portal) - Allocates an unconfigured QMan Portal
- int qman_free_raw_portal(struct usdpaa_raw_portal *portal) - Releases a raw QMan portal that was previously allocated
- int bman_allocate_raw_portal(struct usdpaa_raw_portal *portal) - Allocates an unconfigured BMan portal
- int bman_free_raw_portal(struct usdpaa_raw_portal *portal) - Releases a raw BMan portal that was previously allocated

For detailed information on then usdpaa_raw_portal struct refer to the fsl_usd.h header file.

9.8.1.3.3 DMA memory management

The NXP DPAA hardware provides several peripherals such as FMan, SEC, and PME that read and write memory directly using DMA. Buffers used for peripherals' DMA must be allocated from memory with special properties:

- The memory must be physically contiguous since the peripheral does not access memory via the core's MMU (or any page-mode I/O MMU).
- The memory must be addressable by the peripheral.
- The memory must not be swapped out by Linux while the device is accessing it.
- For user space drivers, there must be an efficient mechanism to convert the physical addresses used by the peripheral hardware to and from the effective addresses used in a user space process or thread's address space.
- For BMan and DPAA usage, it is often convenient for software to allocate memory from physically contiguous regions that are quite large.

- It is desirable to use the Power core's TLB1 mechanism to map large physically contiguous memory regions of this sort. This increases performance by reducing the number of MMU-related interrupts that must be processed. A single TLB0 entry can map only a single 4K page. A single TLB1 entry, in contrast, can map a large (but variable-sized) page. One TLB1 entry can do the work of many TLB0 entries in circumstances such as this one.

Memory that meets the criteria above is called DMA memory. Drivers for all DMA-capable devices must use DMA memory for buffers. This is true for both conventional in-kernel drivers and user space drivers. It is a hardware implication of peripherals' relationships to cores and MMUs.

9.8.1.3.3.1 Current USDPAA solution

The USDPAA Linux kernel reserves a contiguous region of memory very early in the kernel boot process for use as DMA memory. This is done prior to full initialization of the kernel's memory-management subsystem that subsequently inhibits such allocations. This memory reservation is of a size and alignment that is hard-coded into the kernel via the Kconfig option "CONFIG_FSL_USDPAA_SHMEM". Within the kernel's "make menuconfig" interface, this can be found under "Device Drivers" -> "Misc devices" -> "NXP USDPAA shared memory driver". The default is for a 64MB memory reservation.

This same USDPAA kernel driver exposes the reserved memory range via a "/dev/fsl_usdpaa_shmem" character device, which allows the USDPAA application to mmap() the physically-contiguous memory range to a corresponding contiguous range in its virtual address space, thus facilitating trivial virtual/physical address conversions. Additionally, a hook is placed into the memory-management code to "catch" page faults within this memory range and ensure that they are resolved by a single TLB1 mapping that spans the entire memory reservation (rather than the usual 4KB TLB0 mappings). That is, once the USDPAA application has accessed any address within the DMA memory range, no more page faults should occur for any other access within the entire region. Given that USDPAA datapaths of less than 200 processor cycles per packet have been demonstrated, for traffic rates that can consume over 1MB of buffer space in less than a millisecond, we consider that incurring the overhead of page-fault handlers in the kernel when iterating across thousands of pages of memory would likely be too high a price to pay for high performance applications.

The 'fsl_usdpaa_shmem' driver does not automatically constrain the mmap() alignment, so due to the alignment requirements of TLB1 entries, the USDPAA application has to propose virtual base addresses to the kernel rather than letting it allocate them. This will be fixed in a future release. However all of this is encapsulated within the USDPAA "dma_mem" driver. This driver is initialized via the dma_mem_setup() API, which handles the loading and mapping of the DMA memory region. Furthermore, a hard-coded subset of the DMA memory is reserved for buffer pool usage in the USDPAA demo applications, and the rest of it is exposed for general purpose DMA-able memory allocation via the dma_mem_memalign() and dma_mem_free() APIs.

Implementation Note

The USDPAA QMan and BMan APIs always refer to buffers by physical address, as these are what are passed to and from DMA-enabled devices within the Datapath Architecture. So ultimately, any mechanism can be used to provide "DMA memory" to a USDPAA application so long as it allows the application easy access to the memory and an efficient mechanism for converting to and from physical addresses.

Performance considerations will probably also require that page-faults be minimized or eliminated in performance-critical scenarios. The USDPAA "dma_mem" driver (and underlying character device and TLB1 kernel hook) is just one way to achieve this. In future USDPAA releases, it is likely that this will be deprecated in favour of a HugeTLB-based mechanism. In particular, this would help eliminate a limitation of the current USDPAA release, that prevents more than one application instance running at once-- the DMA memory region can only be safely mapped into one process at a time.

9.8.1.3.3.2 DMA memory API

See the <usdpaa/dma_mem.h> header for a simple user space DMA memory API.

- dma_mem_memalign - dynamic allocation of DMA memory from within the large physically contiguous region.
- dma_mem_free - free allocated memory.
- dma_mem_ptov - convert physical to virtual (effective) address within user space process address space.
- dma_mem_vtop - convert virtual (effective) address to physical address.

Expect this API to be expanded in future software releases.

9.8.1.3.4 Network configuration

The USDPAA QMan and BMan drivers do not, in and of themselves, dictate which resources such as frame queues or buffer pools are used. In some cases, they can be dynamically allocated. In other cases, the specific resource is determined by a factor that is external to the USDPAA application itself.

The most common examples relate to the frame manager (FMan) and are discussed in section Relationship to SDK Linux ethernet subsystem on page 1486.

The PPAC-based USDPAA applications ("reflector" and "ipfwd") use the usdpaa_netcfg_acquire() APIto determine the specific resources needed for a network interface. This configuration information is collected from several external sources:

- FMC policy file
- FMC configuration file
- the device-tree

9.8.1.3.4.1 Network configuration initialisation requirements

As described in Device-tree initialization requirements on page 1480, the USDPAA "of" driver must be initialised prior to network configuration being extracted from it. Once this is done, the usdpaa_netcfg_acquire() API, as defined in the <usdpaa/usdpaa_netcfg.h> header, can be used to extract the network configuration. In addition to initialisation of the USDPAA "of" driver layer, this API also requires the path to the two FMC XML files described above (policy and configuration), which are passed as parameters.

To satisfy leak-checkers or to support the use of persistent/reusable processes, an application can also undo all allocations and state by calling usdpaa_netcfg_release(), passing the "info" structure previously returned from usdpaa_netcfg_acquire() as a parameter.

9.8.1.3.5 CPU isolation

USDPAA provides no non-standard Linux features to provide CPU isolation, but the topic is important enough to merit some discussion. Standard Linux features may be use to achieve CPU isolation.

Especially in the run-to-completion use case, it can make sense to dedicate an entire core (which Linux would call a "cpu") to a single USDPAA process or thread. In other words, one wishes to reduce or eliminate any other software use of that particular core.

First, consider user space processes and threads. Linux provides a helpful kernel parameter called "isolcpus". This parameter indicates to the Linux kernel which cores should not have any user space process or thread scheduled onto them by default. The only way that a user space process or thread can execute on an isolated core is by explicit request.

For example, booting the kernel with "isolcpus=1-7" will isolate cores 1 through 7 as shown in the figure below. Note that this isolation is not architecturally required for functionality.



Figure 235. Portal allocation, affinity, and core-isolation

The "isolcpus" kernel parameter is documented (along with others) in the kernel source documentation directory, Documentation/kernel-parameters.txt.

The most convenient way to cause a USDPAA thread to be scheduled on an isolated to core is to use the function pthread_setaffinity_np(). This is a standard Linux pthreads function and "man pthread_setaffinity_np" should give documentation on it. (The "_np", for "non-portable", simply implies that it is not necessarily available in other pthreads implementations on other operating systems.)

The USDPAA example applications also provide an example of this function's use. It is used not only to allow the USDPAA thread to be executed on an isolated core, it makes the thread affine to that core. Recall that affinity is recommended when using the user space QMan and BMan drivers.

Isolation is also a kernel topic. Users who wish to maximize isolation should use the standard Linux interrupt core binding mechanism, /proc/irq to bind interrupts to cores other than those that are isolated. Of course, interrupts associated with portals bound to cores should be bound to that core.

9.8.1.4 Relationship to SDK Linux ethernet subsystem

The DPAA SDK FMan and Linux kernel ethernet driver software is defined as external to the USDPAA software for this release. However, USDPAA, the ethernet driver in the kernel, and the FMan software are all interrelated. The figure below shows four use cases involving the three components.



Figure 236. USDPAA, FMan, ethernet use cases

1. QMan connects FMan MAC only to the kernel ethernet driver, which exchanges frames with the Linux network stack.
2. QMan connects an FMan MAC only to a USDPAA application.
3. QMan connects an FMan MAC to both the kernel driver and to a USDPAA application. On ingress, FMan selects the destination for each frame and enqueues it onto a particular frame queue accordingly. Different frame queues make the connections between the MAC and the ethernet driver and the MAC and the USDPAA application. This use case can be generalized by assuming multiple USDPAA applications, multiple ethernet driver instances, or both.
4. QMan connects an ethernet driver to a USDPAA application. FMan is not involved. Note that it is possible to accomplish this via the standard Linux facility TUN/TAP rather than using QMan.

The current release of USDPAA demonstrates cases 1 and 2.

9.8.1.4.1 Selecting ethernet interfaces for USDPAA

As will be discussed below, the ethernet driver is always involved in configuring FMan. This is true even for the second use case above in which the ethernet driver does not directly use FMan. Instead, the ethernet driver is creating an interface for another entity to use. Historically in the SDK that other entity was the LWE. In the current release, that other entity is a USDPAA application. The FMan and ethernet subsystem is actually unchanged from the standard DPAA SDK.

It is ethernet-related Linux device tree entries that determine the use case. This is documented in the "P4080 DPAA Device Bindings" distributed with the DPAA SDK. The sub-topic is "Data-Path Acceleration Assist".

The following device tree snippet shows a Linux private interface and also an interface used privately by USDPAA.

```
ethernet@0 {
   compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
   fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
   fsl,qman-channel = <&qpool4>;
   fsl,qman-frame-queues-rx = <0x50 1 0x51 1>;
   fsl,qman-frame-queues-tx = <0x70 1 0x71 1>;
   fsl,fman-mac = <&enet0>;
};
ethernet@1 {
   compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";
   fsl,qman-channel = <&qpool1>;
   fsl,fman-mac = <&enet1>;
};
```

The first ethernet is used by USDPAA. The second is used by the Linux ethernet driver.

Table 227. P4080DS ethernet interfaces

Deice Tree Name	U-boot Name	U-Boot MAC Environment Variable	Linux Name (udev)	SerDes 0xe Physical Position
ethernet@0	FM1@DTSEC1	ethaddr	fm1-gb0	not used
ethernet@1	FM1@DTSEC2	eth1addr	fm1-gb1	Motherboad RGMII
ethernet@2	FM1@DTSEC3	eth2addr	fm1-gb2	not used
ethernet@3	FM1@DTSEC4	eth3addr	fm1-gb3	not used
ethernet@4	FM1@TGEC1	eth4addr	fm1-10g	slot 5 XAUI
ethernet@5	FM2@DTSEC1	eth5addr	fm2-gb0	not used
ethernet@6	FM2@DTSEC2	eth6addr	fm2-gb1	not used
ethernet@7	FM2@DTSEC3	eth7addr	fm2-gb2	slot 3 SGMII 2nd closest to motherboard
ethernet@8	FM2@DTSEC4	eth8addr	fm2-gb3	slot 3 SGMII closest to motherboard
ethernet@9	FM2@TGEC1	eth9addr	fm2-10g	slot 4 XAUI

The DPAA SDK uses different names for different physical ethernet interfaces in different contexts. This is due to long-standing decisions on naming conventions, many not made by NXP. It can be confusing. See table above for a list of all of the ethernet interfaces on the P4080 and their names in all of the important contexts. For P4080, no single SerDes protocol number provides access to all of the ethernets at the same time. This is due to pin multiplexing reasons.

The example defaults assume the use of the SerDes 0xe because it gives a total of 23 Gbps of ethernet connectivity on the P4080DS. The table shows which ethernet interfaces are used in SerDes 0xe.

9.8.1.4.2 FMC, FMD, and the ethernet Driver

The Frame Manager (FMan) software in the DPAA SDK is divided into a user space component (FMC) and a kernel component (FMD). The latter is the FMan driver in the conventional sense. FMD provides a capable API to support FMan configuration.



Figure 237. FMC, FMD, and ethernet

By itself, the ethernet driver uses FMD to configure only very simple FMan use cases. Basically, it configures only the default and error frame queues for ingress and egress. Doing more requires running the "fmc" user space application.

This application reads xml files as input and can establish more complex use-cases such as parsing incoming frames and hashing them into a set of frame queues for distribution to multiple cores (meaning USDPAA threads in the context of this document).

Various USDPAA example applications provide XML files and assume that fmc will be run. This usage is described in the application user guides.

9.8.1.5 Supported hardware platforms

This USDPAA release is tested in the following environments:

9.8.1.5.1 P4080DS

- P4080 revision 2 SoC. (Revision 1 is not supported.)
- The main test configuration
  - SGMII card in slot 3 with two interfaces used.
  - XAUI card in slot 4 (but the reflector example will work without it)
  - XAUI card in slot 5 (but the reflector example will work without it)
- SerDes protocol 0xe is the main test case. It provides
  - 1 x 1G ethernet port (via RGMII) used by the Linux kernel network stack.
  - 2 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 2 x 10G ethernet ports (via XAUI) used by USDPAA.
- Testing is done with P4080 high bin clocking:
  - 1500 MHz core, 800 MHz platform, 1300 MHz DDR rate, 600 MHz FMan and PME
- Memory, 4 GB (but 8 GB should work also).
- Linux 8-way SMP.
- All caches (core-private L1/L2 and shared L3) are enabled. No locking or cache partitioning is used.

9.8.1.5.2 P3041DS

- SerDes protocol 0x36 (RR_HXAPNSP_0x36) is the main test case. It provides
  - 2 x 1G ethernet port (via RGMII) used by USDPAA.
  - 3 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 2 x 10G ethernet ports (via XAUI) used by USDPAA.
- no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_15g_1500mhz:
  - 1500 MHz core, 750 MHz platform, 1333 MHz DDR rate, 583 MHz FMan and 375 MHz PME

9.8.1.5.3 P5020DS

- SerDes protocol 0x36 (RR_HXAPNSP_0x36) is the main test case. It provides
  - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
  - 3 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 1 x 10G ethernet port (via XAUI) used by USDPAA.
  - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_15g_2000mhz:
  - 2000 MHz core, 800 MHz platform, 1333 MHz DDR rate, 600 MHz FMan and 400 MHz PME

9.8.1.5.4 P5040DS

- SerDes protocol 0x02 (RR_XXSNSpP_0x02) is the main test case. It provides
  - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
  - 4 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 2 x 10G ethernet port (via XAUI) used by USDPAA.
  - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_26g_2267mhz.bin:
  - 2267 MHz core, 800 MHz platform, 1600 MHz DDR rate, 600 MHz FMan and 400 MHz

9.8.1.5.5 P2041RDB

- SerDes protocol 0x09 (RR_PX_0x09) is the main test case. It provides
  - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
  - 2 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 1 x 10G ethernet port (via XAUI) used by USDPAA.
  - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_14g_1500mhz:
  - 1500 MHz core, 750 MHz platform, 1333 MHz DDR rate, 583 MHz FMan and 400 MHz PME

9.8.1.5.6 B4860QDS

- SerDes protocol 2a and 49 (N_NNSS_0x2A_0x49) is the main test case. It provides
  - 4 x 1G ethernet port (via SGMII) used by USDPAA.

- 1 x 1G ethernet port (via SGMII) used/available for the Linux kernel network stack.

- Testing is done with the rcw_5sgmii_1600mhz.bin:
  - 1600 MHz core, 666.667 MHz platform, 800 MHz DDR rate, 666.667 MHz FMan and 333.333 MHz QMan.

9.8.1.5.7 T4240QDS

- SerDes protocol RR_XXXXPRPR_1_1_6_6 is the main test case. It provides
  - 2 x 1G ethernet ports (via SRIO) used by USDPAA.
  - 4 x 10G ethernet port (via XAUI) used by USDPAA.
  - 2 x 1G ethernet ports (via PCIe) used/available for the Linux kernel network stack.
- Testing is done with the rcw_1_1_6_6_1666MHz.bin:
  - 1666.667 MHz core, 666.667 MHz platform, 800 MHz DDR rate, 400 MHz FMan and 333.333 MHz PME

9.8.1.6 Example applications

USDPAA will be distributed with a set of example applications that is expected to grow over time.

The current set is

- A packet reflector application, "reflector".
- An IP forwarding performance demonstration, "ipfwd".
- A cryptographic accelerator example, "simple_crypto".
- A pattern-matching accelerator example, "pme_loopback_test".
- An IPFwd application based upon Longest Prefix Match methodology, "lpm_ipfwd".
- An application to route IPv4 packets after performing encryption/decryption, "IPSecfwd".
- A non-PPAC based stand-alone application, "hello_reflector".

Each example has its own user manual.

9.8.1.7 USDPAA installation and execution

The USDPAA software is contained within the SDK release, and should be compiled and installed by default by following the general instructions accompanying the release. Those instructions will not be reproduced here in their entirety. All that should be required to bring up and run the USDPAA environment, compared to booting the "standard" Linux system from the SDK, is to use the USDPAA-specific device tree. Eg. on the P4080DS system, one should use "uImage-p4080ds-usdpaa.dtb" instead of "uImage-p4080ds.dtb". Other than that, please follow the steps prescribed in the SDK installation notes. The following sections reproduce some of these steps (for the P4080DS case only), but primarily to ensure that USDPAA-relevant information is correctly identified.

9.8.1.7.1 Files needed to boot Linux on the P4080DS system

To run the USDPAA software, one must first boot Linux with the correct files;

- R_PPSXX_0xe/**rcw_2sgmii_1500mhz.bin**

  Reset configuration word (rcw) that selects the SerDes Protocol, suitable to program into the P4080DS NOR flash.

- **u-boot-P4080DS.bin**

  U-Boot bootloader image suitable to program into the P4080DS NOR flash.

- **fsl_fman_ucode.bin**

  FMan microcode for appropriate P4080 silicon suitable to program into the P4080DS NOR flash.

- **uImage-p4080ds.bin**

  Linux kernel supporting USDPAA.

- **uImage-p4080ds-usdpaa.dtb**

  Device tree file configured for USDPAA usage.

- **fsl-image-core-p4080ds.ext2.gz.u-boot**

  File system (used in RAM disk) that contains the needed user space files including USDPAA example application binaries. Linux must be booted using this file system.

All six of the files listed above are needed to run USDPAA. The first three must be programmed into the P4080DS NOR flash. The last three may be programmed into the NOR flash, but also may be loaded into RAM by u-boot using the tftp protocol.

U-boot is also capable of loading files into RAM via tftp and then programming them into the NOR flash. In all cases, you must have access to a tftp server, ideally on your Linux development host.

Copy the six files listed above to a directory from which they can be accessed via your tftp server. U-boot on the P4080DS must use tftp to access them. Details of installing and configuring a tftp server on your development host are specific to your host Linux distribution.

9.8.1.7.2 About U-Boot and network interfaces

U-boot is a bootloader. It is very flexible. but its main job is to do quite a bit of system configuration and then to load an operating system image (mainly Linux) into RAM and transfer control to it.

One of the simplest ways of transferring images (and other files) to the P4080DS running u-boot is to use tftp. For this to work, you must configure a network interface in u-boot.

Specifically, we assume that the 1 Gbps ethernet interface on the P4080DS motherboard will be used, and that the RCW file used causes this interface to correspond to the 2nd DTSEC in the P4080's first FMan instance.

At this point, we list for reference the text that U-boot should print when it runs. Note that there can be some variation in what U-boot prints.

```
U-Boot 2011.06-rc1-00037-g2bc0243 (May 27 2011 - 11:01:49)

CPU0:  P4080E, Version: 2.0, (0x82080020)
Core:  E500MC, Version: 2.0, (0x80230020)
Clock Configuration:
       CPU0:1499.985 MHz, CPU1:1499.985 MHz, CPU2:1499.985 MHz, CPU3:1499.985 MHz,
       CPU4:1499.985 MHz, CPU5:1499.985 MHz, CPU6:1499.985 MHz, CPU7:1499.985 MHz,
       CCB:799.992 MHz,
       DDR:649.994 MHz (1299.987 MT/s data rate) (Asynchronous), LBC:99.999 MHz
       FMAN1: 599.994 MHz
       FMAN2: 599.994 MHz
       PME:   599.994 MHz
L1:    D-cache 32 kB enabled
       I-cache 32 kB enabled
Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0b, vBank: 4
36-bit Addressing
Reset Configuration Word (RCW):
       00000000: 105a0000 00000000 1e1e181e 0000cccc
       00000010: 3842440c 3c3c2000 fe800000 e1000000
       00000020: 00000000 00000000 00000000 008b6000
       00000030: 00000000 00000000 00000000 00000000
SERDES Reference Clocks: Bank1=100MHz Bank2=125MHz Bank3=125MHz
I2C:   ready
SPI:   ready
DRAM:  Initializing....using SPD
Detected UDIMM EBJ21EE8BAFA-DJ-E
```

```
Detected UDIMM EBJ21EE8BAFA-DJ-E
CS2 is disabled.
CS3 is disabled.
CS2 is disabled.
CS3 is disabled.
2 GiB left unmapped
    DDR: 4 GiB (DDR3, 64-bit, CL=9, ECC on)
       DDR Controller Interleaving Mode: cache line
       DDR Chip-Select Interleaving Mode: CS0+CS1
Testing 0x00000000 - 0x7fffffff
Testing 0x80000000 - 0xffffffff
Remap DDR 2 GiB left unmapped

POST memory PASSED
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 2048 KB enabled
SRIO1: disabled
SRIO2: disabled
MMC:  FSL_ESDHC: 0
EEPROM: Invalid ID (ff ff ff ff)
PCIe1: Root Complex, x1, regs @ 0xfe200000
  01:00.0     - 1095:3132 - Mass storage controller
PCIe1: Bus 00 - 01
PCIe2: disabled
PCIe3: Root Complex, no link, regs @ 0xfe202000
PCIe3: Bus 02 - 02
In:    serial
Out:   serial
Err:   serial
Net:   Fman1: Uploading microcode version 101.8.0
FM1@DTSEC2 connected to Vitesse VSC8244
FM1@TGEC1 connected to Teranetics TN2020
Fman2: Uploading microcode version 101.8.0
FM2@DTSEC3 connected to Vitesse VSC8234
FM2@DTSEC4 connected to Vitesse VSC8234
FM2@TGEC1 connected to Teranetics TN2020
FM1@DTSEC2, FM1@TGEC1, FM2@DTSEC3, FM2@DTSEC4, FM2@TGEC1
```

In the above text, notice the final line;

```
FM1@DTSEC2, FM1@TGEC1, FM2@DTSEC3, FM2@DTSEC4, FM2@TGEC1
```

It lists the available network interfaces, which is determined by the RCW file and the SerDes protocol that it selects. This document assumes that SerDes 0xe is used. It provides the interfaces above. They will be used as follows.

- FM1@DTSEC2, U-boot MAC "eth1addr"

  This is the 1 GBps ethernet interface on the P4080DS motherboard. It will be used by U-Boot to transfer images and also is available in Linux as a standard ethernet interface. The "ifconfig" command will show it as "fm1-gb1" because the "gb's" are counted from zero.

- FM1@TGEC1 (slot 5 XAUI), U-boot MAC "eth4addr"
- FM2@DTSEC3 (slot 3 SGMII, 2nd closest to motherboard), U-boot MAC "eth7addr"
- FM2@DTSEC4 (slot 3 SGMII, closest to motherboard), U-boot MAC "eth8addr"
- FM2@TGEC1 (slot 4 XAUI), U-boot MAC "eth9addr"

  These other 4 interfaces, 2 x 1 GB and 2 x 10 GB, are dedicated to user space access via USDPAA.

This provides the USDPAA application with 22 Gbps of full-duplex ethernet capacity, assuming that an SGMII card is in P4080DS slot 3, and 10G XAUI cards are in slots 4 and 5. The USDPAA "reflector" application will function if the XAUI cards are not present, but only two GB of capacity will be available.

The P4080 has many ethernet interfaces. See Selecting ethernet interfaces for USDPAA on page 1487 for a complete summary of the names of these interfaces in different software contexts and for a statement of their use by the SerDes 0xe protocol on the P4080DS.

Returning to u-boot, selection of the network interface and networking parameters is done via u-boot environment variables. These are set using the "setenv" command that can be entered at the u-boot prompt. The network parameters include MAC addresses for the interfaces. Set them for all ten of the P4080 interfaces even though only a subset will be used.

Here is a complete list of the needed setenv commands. You will have to adjust the IP addresses and netmask to match your network. The addresses shown below are examples only.

```
setenv ethact FM1@DTSEC2
setenv ethaddr 00:04:9F:77:4E:00
setenv eth1addr 00:04:9F:77:4E:01
setenv eth2addr 00:04:9F:77:4E:02
setenv eth3addr 00:04:9F:77:4E:03
setenv eth4addr 00:04:9F:77:4E:04
setenv eth5addr 00:04:9F:77:4E:05
setenv eth6addr 00:04:9F:77:4E:06
setenv eth7addr 00:04:9F:77:4E:07
setenv eth8addr 00:04:9F:77:4E:08
setenv eth9addr 00:04:9F:77:4E:09
setenv ipaddr 10.82.146.151
setenv serverip 10.82.146.150
setenv gatewayip 10.82.146.150
setenv netmask 255.255.255.0
saveenv
```

In summary, U-boot will use the motherboard ethernet and will give it IP address 10.82.146.151. The tftp server has IP address 10.82.146.150. The "saveenv" saves all environment variable values into flash so they are retained after a reboot.

After entering the values above, you can test the U-boot network connection via ping and a trial tftp transfer. If this does not work, check the variables and network cables. This must work. Here is a text capture showing a successful test. You may need to adjust the path usdpaa/u-boot.bin per your tftp server configuration.

```
=> ping $serverip
Using FM1@DTSEC2 device
host 10.82.146.150 is alive
=> tftpboot 01000000 usdpaa/u-boot.bin
Using FM1@DTSEC2 device
TFTP from server 10.82.146.150; our IP address is 10.82.146.151
Filename 'usdpaa/u-boot.bin'.
Load address: 0x1000000
Loading: ###################################
done
Bytes transferred = 524288 (80000 hex)
```

9.8.1.7.3 P4080DS NOR flash banks

The P4080DS board has a feature that uses address swizzling to make it appear that the NOR flash is divided into multiple parts-- this document will assume two. The parts are called "bank 0" and "bank 4".

When you power-on or reset, u-boot will boot from bank 0. U-boot in bank 0 can program images into bank 4. Then, you can enter "pixis altbank" from the bank 0 u-boot prompt to boot into bank 4.

It is very wise to leave the bank 0 images alone and simply use them to program images into bank 4. This is to ensure that you always have working images in bank 0. This document will assume that you have u-boot flashed in bank 0. Use it only to program bank 4.

Look at the u-boot output in About U-Boot and network interfaces on page 1492. The line

```
Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0b, vBank: 4
```

shows the bank from which u-boot was booted. In this case it was bank 4 (per vBank).

9.8.1.7.4 Programming the P4080DS NOR flash bank 4

First, boot from bank 0 by doing a reset or power-on. Check that you see "vBank: 0" since this is very important. Then, set network parameters using the u-boot environment variables described in section About U-Boot and network interfaces on page 1492.

The following U-boot commands will flash all six of the needed files into bank 4. Remember that you may have to adjust the paths in the tftpboot commands per your tftp server.

```
# BE BOOTED FROM BANK 0; WE WILL FLASH THE ALT BANK, WHICH WILL BE BANK 4

# rcw altbank
tftpboot 0x01000000 usdpaa/rcw_2sgmii_1500mhz.bin
protect off 0xec000000 +$filesize && erase 0xec000000 +$filesize && cp.b 0x01000000 0xec000000
$filesize

# u-boot altbank
tftpboot 0x01000000 usdpaa/u-boot.bin
protect off 0xebf80000 +$filesize && erase 0xebf80000 +$filesize && cp.b 0x01000000 0xebf80000
$filesize

# Delete altbank the u-boot env-- use default
protect off 0xebf60000 +0x20000 && erase 0xebf60000 +0x20000

# FMan u-code altbank
tftpboot 0x01000000 usdpaa/fsl_fman_ucode.bin
protect off 0xeb000000 +$filesize && erase 0xeb000000 +$filesize && cp.b 0x01000000 0xeb000000
$filesize

# device tree alt bank
tftpboot 0x01000000 usdpaa/p4080ds-usdpaa.dtb
protect off 0xec800000 +$filesize && erase 0xec800000 +$filesize && cp.b 0x01000000 0xec800000
$filesize

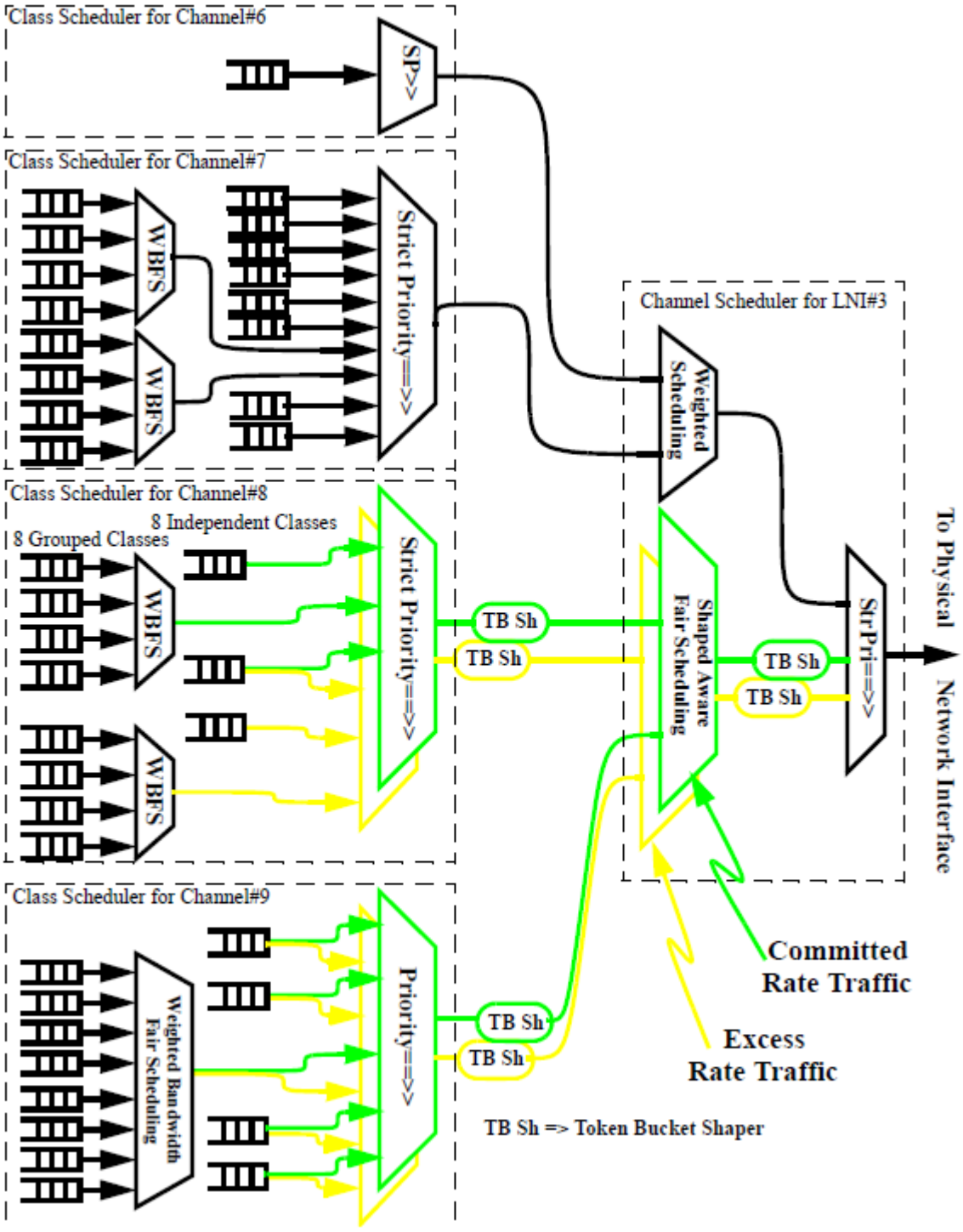
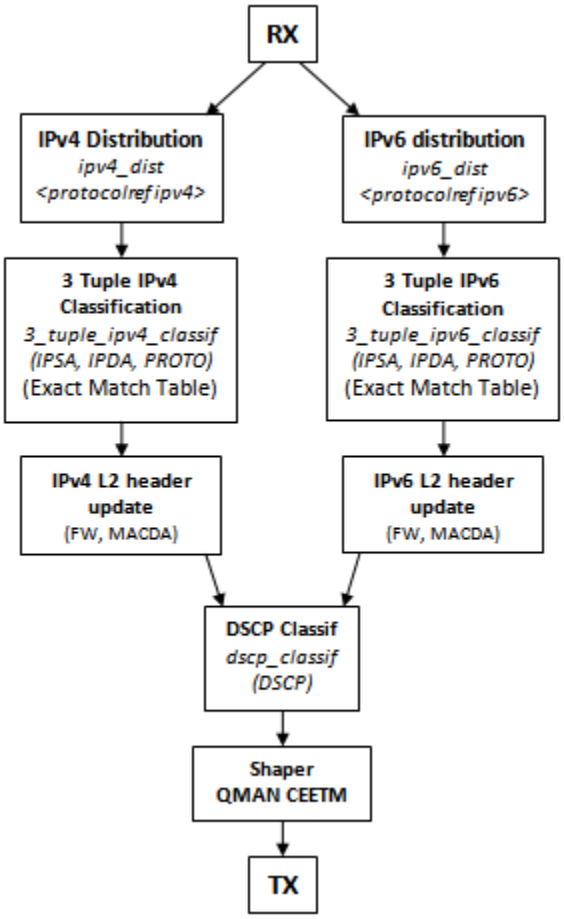
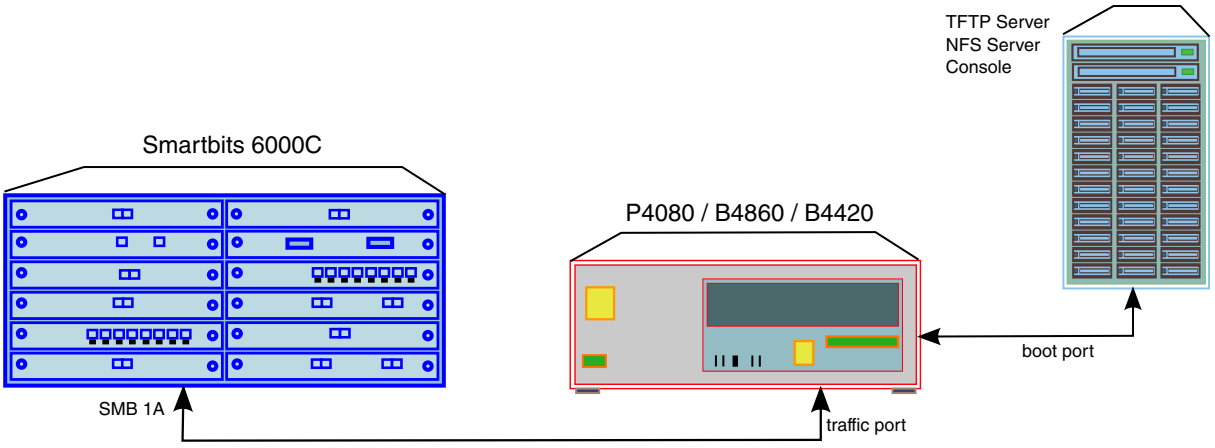
# kernel altbank
tftpboot 0x01000000 usdpaa/uImage
protect off 0xec020000 +$filesize && erase 0xec020000 +$filesize && cp.b 0x01000000 0xec020000
$filesize

# rootfs altbank
tftpboot 0x01000000 usdpaa/initramfs.cpio.gz.uboot
protect off 0xed300000 +$filesize && erase 0xed300000 +$filesize && cp.b 0x01000000 0xed300000
$filesize
```

9.8.1.7.5 Boot into bank 4 and set more variables

Next, enter the command "pixis altbank" to boot into bank 4 and press "any key" to stop the boot. Check that u-boot prints "vBank: 4". It is important that it does. If not, there is probably a mistake in the previous steps.

At this point, you must enter all of the networking u-boot environment variables (see section About U-Boot and network interfaces on page 1492) again and also set variable bootcmd. The latter is shown below along with a "saveenv" to save the values.

```
setenv bootcmd setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M
bportals=s0-1 qportals=s0-1 \; bootm 0xe8020000 0xe9300000 0xe8800000
saveenv
```

It is important that you type the backslash (\) because semicolon (;) is a command separator in u-boot. Here is a "printenv bootcmd" in a larger font and with line wrap showing the correct value. Note that the printenv does not show the backslash.

```
=> printenv bootcmd
bootcmd=setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M bportals=s0-1
qportals=s0-1 ; bootm 0xe8020000 0xe9300000 0xe8800000
```

U-boot has an environment variable "bootdelay" that controls the number of seconds u-boot counts down before automatically running command "boot". If you prefer to run "boot" manually, set bootdelay to -1. This will cause u-boot to go directly to a command prompt. You can set bootdelay to whatever you want.

```
setenv bootdelay -1
saveenv
```

9.8.1.7.6 Environment variable hwconfig and optical 10G

At this point , enter the u-boot command "printenv hwconfig" as shown below.

```
=> printenv hwconfig
hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

If you see the value above, and you plan to use the 10G copper interfaces (or no 10G at all), then all is well. You may skip to the next section.

If you plan to use optical 10G interfaces, you must add information to hwconfig. This is important. The optical interface will operate erratically without it. It is easiest to do this using the u-boot "editenv" command. Whatever you type will be appended to hwconfig. The text you need to append is

```
;fsl_fm2_xaui_phy:xfi;fsl_fm1_xaui_phy:xfi
```

The leading semicolon is needed. Do another "printenv hwconfig" to ensure that the value is correct and then a saveenv as shown below.

```
=> printenv hwconfig
hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;fsl_fm2_xaui_phy:xfi;fsl_fm1_xaui_ph
y:xfi
=> saveenv
```

9.8.1.7.7 Booting Linux

At this point, you need to reset once again into bank 4 (u-boot command "pixis altbank") and run the u-boot "boot" command, either manually or by letting it happen automatically after the count-down.

Linux will boot and give you a login prompt.

Login as user "root" with password "root".

An "ifconfig -a" should show only one FMan (fm) ethernet interface, fm1-gb1. This is the P4080DS motherboard 1G ethernet interface. You can set its IP address and use it as an ordinary Linux network interface if you wish.

If you cat /root/SOURCE_THIS, you will see the commands needed to run the "reflector" example application. This is the first application you should examine. Please see its user manual for more information.

```
[root@p4080 /root]# cat /root/SOURCE_THIS
cd /usr/etc
fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst.xml -a
reflector
```

9.8.1.7.8 Using tftp for the kernel, device-tree, and file-system

It can be slow to reflash the kernel, device tree, and file system every time you run ltib to change a USDPAA application (see section Known limitations on page 1498). As an alternative, you can use the following u-boot commands (from bank 4) to boot Linux.

```
tftpboot 01000000 usdpaa/uImage
tftpboot 02000000 usdpaa/p4080ds-usdpaa.dtb
tftpboot 02100000 pusdpaa/initramfs.cpio.gz.uboot
boot
```

However, in this case the addresses in the "bootm" run by bootcmd need to be different. So, set bootcmd as follows.

```
setenv bootcmd setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M
bportals=s0-1 qportals=s0-1 \; bootm 01000000 02100000 02000000
saveenv
```

9.8.1.8 Using configurations other than SerDes 0xe

Most NXP testing, examples, and discussion of examples in documentation assumes the use of SerDes 0xe and 22 Gbps of ethernet connectivity to USDPAA in the form of 2 x 1 Gbps + 2 x 10 Gbps.

It is possible to use USDPAA in other configurations and some examples will be summarized below.

See the document "NXP DPAA SDK <version>: Selecting Ethernet Interfaces" for background information. It is distributed with the NXP DPAA SDK.

9.8.1.8.1 SGMII (4 x 1 Gbps) card and one XAUI (10 Gbps) card

Goal (using Linux names for network interfaces, see Selecting ethernet interfaces for USDPAA on page 1487):

- fm1-gb1 used by Linux kernel
- fm2-gb2 used by USDPAA
- fm2-gb3 used by USDPAA
- fm2-10g used by USDPAA

Method:

- SGMII card goes in P4080DS slot 3.
- XAUI card goes in P4080DS slot 4.
- Continue to use SerDes 0xe and RCW file R_PPSXX_0xe/rcw_2sgmii_1500mhz.bin in the P4080DS NOR flash.
- Boot into bank 4 (assuming you are using bank 4 for USDPAA).
- Add text ";serdes:fsl_srds_lpd_b3=0xf" to u-boot environment variable hwconfig, saveenv, and reset the system again into bank 4. This disables fm1-10g. The leading ";" separates the information added to hwconfig from what was already there.

- Delete references to fm1-10g from the fmc configuration file you will use. For example, delete the following text from it:

```
<engine name="fm0">
          <port type="10G" number="0" policy="hash_ipv4_policy_4"/>
</engine>
```

Run the USDPAA application.

9.8.1.8.2 SGMII (4 x 1 Gbps) card and no XAUI (10 Gbps) card

Goal (using Linux names for network interfaces, see table Selecting ethernet interfaces for USDPAA on page 1487):

- fm1-gb1 used by Linux kernel
- fm2-gb0 used by USDPAA
- fm2-gb1 used by USDPAA
- fm2-gb2 used by USDPAA
- fm2-gb3 used by USDPAA

Method:

- SGMII card goes in P4080DS slot 3.
- Use SerDes 0x10 and RCW file: R_PPSXN_0x10/**rcw_5g_1500mhz.bin** in the P4080DS NOR flash. This is a different RCW file than the one discussed above.
- Boot into bank 4 (assuming you are using bank 4 for USDPAA).
- Add text ";serdes:fsl_srds_lpd_b2=0xf" to u-boot environment variable hwconfig, saveenv, and reset the system again into bank 4. This disables fm2-10g. The leading ";" separates the information added to hwconfig from what was already there. Note that fm1-10g is never available in SerDes 0x10.
- Delete references to any of the 10 Gbps interfaces and ensure references to all USDPAA 1 Gbps interfaces are present in the fmc configuration file. For example, it should like the following after you edit it:

```
<cfgdata>
    <config>
        <engine name="fm1">
                <port type="1G" number="0" policy="hash_ipv4_policy_5"/>
                <port type="1G" number="1" policy="hash_ipv4_policy_6"/>
                <port type="1G" number="2" policy="hash_ipv4_policy_7"/>
                <port type="1G" number="3" policy="hash_ipv4_policy_8"/>
        </engine>
    </config>
</cfgdata>
```

Run the USDPAA application.

9.8.1.9 Known limitations

- Interrupts for QMan and BMan portals used in user-space by USDPAA threads are not necessarily affine to the CPU to which the portal is assigned (ie. to the portal where stashing is performed and where the threads are advised to be affine). This is a current limitation of the UIO interface in the kernel which does not give the QMan/BMan drivers explict control over interrupt affinity. However given that interrupts are generally used to implement sleeping/blocking semantics (eg. when idle), this is not expected to have a significant impact. As a workaround, if the need arises, the user can manually override the interrupt-affinity via the procfs controls available at /proc/irq.
- Only 1 Gbps full-duplex operation is supported on 1 Gbps ethernet links. It is also true that 10 Gbps links may only be used at 10 Gbps, but in this case the reason is a P4080DS board-level hardware limitation.

- Present release does not permit working with all four SGMII 1 Gbps ethernet ports and a XAUI 10 Gbps ethernet port at the same time. This would appear possible using SerDes 0x10, but it is not due to FMan buffer size constraints and support for jumbo frames. This is, at heart, a P4080 SoC hardware limitation, but future releases will provide greater flexibility.

9.8.1.10 Document history

Table 228. Document history

Rev	Notes
1.0	Initial version.
1.1	FQID detail and more.
1.2	Description of proof-of-concept added.
1.3	Add TX FQIDs and discussion of application timing
1.4	Documentation of configuration files.
1.5	Instructions on running on the P4080DS. This version describes the first release of the proof-of-concept.
1.6	Corrected reserving 512 bytes in buffers.
1.7	Updated performance table for case when FMan automatically releases buffers.
1.8	Updated for 10G XAUI + 4x1G SGMII support and some optimizations.
1.9	Changed document title. Updated for phase 0 release.
1.10	Updated for USDPAA phase 1 release.
1.11	Late updates for phase 1 release including advice on non-22Gbps runs.
1.12	Add discussion of use cases beyond SerDes 0xe and 22 Gbps of ethernet connectivity for USDPAA.
1.13	Fixed minor typographical errors and made other small improvements.
1.14	Updated for SDK v1.0.
1.15	Updated for SDK v1.3.
1.16	Added T4 & B4 platforms for SDK v1.3.1.

9.8.2 USDPAA Multiple Processor Support User Guide

9.8.2.1 USDPAA Multiple Process Support

Describes the modifications done in the QorIQ SDK1.2 to provide USDPAA multiple process support.

The changes introduced in the SDK V1.2 release and are described in this document as a set of changes between SDK V1.1 and the SDK V1.2 releases. Concurrently, a set of interim releases (IRn) are being developed to provide DPAA Offload support

in the SDK. The IR2 release integrates support for multiple USDPAA processes. As such, the delta between SDK V1.1 and SDK V1.2 described in this document also applies as the delta between IR1 and IR2 releases.

9.8.2.2 USDPAA User/Kernel Device Interface

For SDK 1.1 most USDPAA resources were hard-coded into the USDPAA applications, there being only a single USDPAA process. For SDK 1.2 each process opens the “/dev/fsl_usdpaa” device once and uses this process driver for all that process's resource management.

SDK 1.1 QMan/BMan resources

Prior to the SDK 1.2 release, a simplifying assumption of a single process, together with assumptions about unused resources in the device tree led to hard-coded resources. This was the case for buffer pools, congestion groups, pool channels and frame queues.

SDK 1.1 DMA Memory

The only USDPAA resource that was kernel-managed was the mapping of DMA memory, via the

```
/dev/fsl_usdpaa_shmem
```

device. This also had the simplifying assumption that the entire memory region reserved by the kernel would always be mapped in its entirety by the unique USDPAA process, meaning that the functions in the USDPAA “dma_mem” API did not need any parameter to specify which DMA region was implied as there was only one region.

SDK 2.1 Process Driver

The “fsl_usdpaa_shmem” device has been renamed to simply “/dev/fsl_usdpaa” and is also referred to as the “process” device (within USDPAA, this is handled via the “process” driver), because the intention is for each process to open this device once and to use it for all that process's resource management.

SDK 2.1 QMan and BMan Resources

As of the 1.2 release, the process device supports ioctl() commands for (de)allocating resources to (and from) the kernel. So multiple USDPAA processes as well as any datapath logic in the kernel will all be using the same allocator for resources.

SDK 2.1 DMA Memory

The DMA mapping of the device remains but is enhanced to allow multiple regions and sub-regions to be allocated and mapped out of the total memory reservation. These regions are mapped independently and can be allocated by USDPAA apps, via the “dma_mem” driver, on the fly. The allocated regions have the same size and alignment limitations that come from the use of TLB1 entries to map them for fault-less access during datapath operations, but the kernel management of these regions is maximally optimal. Ie. any combination of regions that can conceivably fit within the total memory reservation will always be obtainable, independent of the order in which the USDPAA processes request the allocation of those regions.

Other features of the allocation and mapping of DMA regions:

- They can be process-private (“unnamed”) or shareable (“named”). Named regions can be mapped by multiple processes.
- For shared regions, the process driver in the kernel provides a sleep-based locking scheme that the USDPAA dma_mem driver uses to synchronise buffer allocations within a sub-region across multiple processes.

SDK 2.1 Resource Tracking

Because USDPAA processes open the /dev/fsl_usdpaa device and perform all resource management through that file-descriptor, the kernel device driver can track what resources are allocated and deallocated by those process. When such a process exits (intentionally or otherwise) the file-descriptor is cleaned up and this allows the device driver to check which resources had not been explicitly released by the application. If there are unreleased frame queues, buffer pools, pool channels or congestion groups, the kernel driver will issue leak warnings to the kernel log (and/or the serial console).Examples:

```
USDPAA process leaking 10 FQIDs
USDPAA process leaking 4 QPOOLs
```

Leaked resources are not automatically returned to the allocators, because the current drivers do not yet support automatic clean up and recovery of resources that are left in an undefined (and possibly volatile) state.

Even when applications explicitly deallocate resources back to the kernel-managed allocators (subsequently described here in the "Multi-process PPAC Applications" topic), there is no protection against applications that fail to first put those resources back into their expected “power-on” states. As such, an application that does not correctly clean up resources can, for the current version of the SDK (1.2), pollute the allocators with resources that will be allocated out to other users and lead to undefined results.

9.8.2.3 USDPAA Resource Management

Describes QMan and BMan resource availability and how to declare these resources at system initialization.

QMan and BMan Portals

In SDK 1.1, QMan and BMan portals were declared in the device-tree with properties that pre-determined whether they were for use in the kernel or USDPAA (the latter were marked by the “fsl,usdpaa-portal” property) as well as a pre-determined CPU-affinity (the “cpu-handle” property links to a CPU node).

For SDK 1.2 and later versions, portals are declared in the device-tree simply as hardware resources, with no specification of what the portals will be used for nor which CPU they will be used from. The kernel parses all portals into an internal list, from(/to) which they can be (de)allocated as required.

The QMan driver will, by default, try to allocate a distinct portal for each core and initialise it for kernel use. This behaviour can be influenced by the use of the “qportals” boot argument, to use fewer portals and share them between cores. The mechanism by which a portal can be shared involves cores that do not have their own portals being “slaves” to a core that does have its own portal. Portal processing (interrupts, polling, changing dequeue masks, [etc]) is still performed only on the core to which the portal has been assigned, but software running on slave cores can perform software-initiated commands (enqueues, management commands, [etc]) on the shared portal due.

Once the kernel has initialised portals for its own use, it will allocate and export all the remaining portals as UIO devices for USDPAA use. When a USDPAA application thread initialises a portal for use, the opening of the UIO device triggers kernel logic to configure the portal as affine to the CPU the USDPAA thread is executing on.

QMan Frame Queues (FQs) in SDK 1.1

The allocation of FQs was not coordinated between user-space and kernel-space. Kernel-space FQ allocations would, by default, acquire FQIDs from buffer pool zero, which was statically seeded (via device-tree entries) with a range of values from 0x100 (255) to 0x1ff (511) inclusive, though this behaviour could be overridden by the presence of a “fsl,fqid-range” node in the device-tree which would bypass the buffer pool allocator and instead implemented a software allocator using the given range.

User-space FQ allocations on the other hand always used a software allocator implementation whose range was hard-coded (in source code) to be from 0x200 (512) to 0x3ff (1023).

QMan FQs in SDK 1.2

In kernal-space, support for using buffer pool zero as a special case for FQ allocations has been entirely removed. Now, FQ allocations are only possible if the device-tree contains a “fsl,fqid-range” node. There are device-tree include files (arch/powerpc/boot/dts/fsl/qoriq-dpaa-res*.dtsi) that declare default allocation ranges.

User-space FQ allocations are now always handled by using the resource allocation ioctl() commands in the USDPAA “process” driver. Ie. user-space FQ allocations are sourced from the allocator residing in kernel-space, and so multiple user-space processes and kernel code are using a common allocator.

QMan Congestion Group Records (CGRs) for SDK 1.1

There was no facility at all for providing allocation of CGRs, and indeed there was no knowledge on the part of kernel or user-space as to how many CGRs were physically present in the hardware – any CGR-dependent software would simply be making its own assumptions about what CGRIDs were safe to use relative to the hardware and any other CGR-dependent software.

QMan CGRs for SDK 1.2

The kernel now implements a CGRID allocator which is seed by a “fsl,cgrid-range” node in the device-tree. The user-space has the same API, and its allocations are routed in the kernel via the “process” driver in the same way as was mentioned for frame queues.

QMan Pool Channels in SDK 1.1

There was no facility for providing allocations of pool-channels, however the device-tree did represent how many pool-channels were present in hardware (this was primarily to allow network devices to be statically configured to use particular pool-channels via device-tree linkage). The kernel-space driver could then enforce its knowledge of what pool-channels were available, but did not coordinate the resource so independent entities of software would need to avoid conflicts via its own means. The user-space driver also used the device-tree to determine the physically-available pool-channels, and provided no coordination of the resources.

QMan Pool Channels in SDK 1.2

The use of device-tree linkage between network nodes and pool-channels in previous versions is gone. The kernel network driver has been adjusted to dynamically allocate its pool-channel instead. As with the other resource types, the USDPAA driver now has the same pool-channel allocation API as the kernel, with user-space allocation operations going via the “process” driver to be handled by the allocator in the kernel.

BMan Buffer Pools in SDK 1.1

The kernel used to determine the number of buffer pools available by looking at the SoC version. Some buffer pools would be represented by device-tree nodes in order to support device-tree linkage with network nodes and in doing so could optionally be seed with ranges of values specified by node properties. The kernel would, by default, implement a BPID allocator that would automatically include all physically available buffer pools that were not explicitly mentioned in device-tree nodes (ie. device-tree nodes acted as reservations against being allocated). An optional “fsl,bpool-range” node could be used to override this behaviour, implementing a software allocator in the same way as “fsl,fqid-range” does.

BMan Buffer Pools in SDK 1.2

If the device-tree contains a “fsl,bpid-range” node (previously named “fsl,bpool-range”). There are device-tree include files

```
arch/powerpc/boot/dts/fsl/qoriq-dpaa-res*.dtsi
```

that declare default allocation ranges. User-space FQ allocations are now always handled by using the resource allocation ioctl() commands in the USDPAA “process” driver. Ie. user-space FQ allocations are sourced from the allocator residing in kernel-space, and so multiple user-space processes and kernel code are using a common allocator.

Changes for FMan Resources

The only change to FMan resource management caused by the SDK 1.2 changes for USDPAA multi-process support is the removal of statically-assigned pool channels for ethernet interfaces. The “dpaa_eth” kernel driver now dynamically allocates a pool-channel during initialisation, and uses it for all the network interfaces it instantiates.

USDPAA DMA Memory for SDK 1.1

The kernel driver used an early-boot hook to reserve a memory region of a hard-coded size (configurable at the expense of a kernel recompile), and the USDPAA “dma_mem” driver would open the “/dev/fsl_usdpaa_shmem” device on behalf of the unique application process and mmap() all the of reserved memory. It was not possible to map less memory than that, it was not possible to create named/shared mappings, and so this was one of the reasons it was not possible to run multiple USDPAA processes.

USDPAA DMA Memory for SDK 1.2

The kernel driver now requires a boot-argument (“usdpaa_mem=<size>[,<num_tlb1>”) to trigger the reservation of USDPAA memory early during the kernel boot, otherwise no memory is reserved. The reservation of multiple TLB1 indices for use by USDPAA is also possible by passing a comma-separated argument. Using multiple TLB1 indices allows simultaneous mappings from USDPAA processes to DMA regions without any risk of fault handling overheads (note that if two processes map the same shared region, that requires 2 TLB1 indices). See section 2.2.3 for information on the device that exposes this memory to mapping from USDPAA applications.

9.8.2.4 BMan and QMan API

SDK 1.2 USDPAA processes performs resource management through the /dev/fsl_usdpaa file-descriptor .

BMan Modifications - fsl_bman.h

The changes described in this section apply to the BMan API in kernel-space and user-space unless otherwise specified. For more information on specific APIs (eg. the “partial” parameter to allocation functions or API return values) please see the comments in the header file that declares the interface (or consult the API reference manual).

Removal of “recovery” API

bman_recovery_cleanup_bpid() and bman_recovery_exit() have been removed, as they were never more than non-functional stubs and were conflicting with ongoing development.

New BPID allocation API

```
int bman_alloc_bpid_range(u32 *result, u32 count, u32 align, int partial);
static inline int bman_alloc_bpid(u32 *result)
{
       int ret = bman_alloc_bpid_range(result, 1, 0, 0);
       return (ret > 0) ? 0 : ret;
}

void bman_release_bpid_range(u32 bpid, unsigned int count);
static inline void bman_release_bpid(u32 bpid)
{
       bman_release_bpid_range(bpid, 1);
}
```

QMan Modifications - fsl_qman.h

The changes described here apply to the QMan API in kernel-space and user-space unless otherwise specified.

Removal of "recovery" API for SDK 1.2

qman_recovery_cleanup_fq() and qman_recovery_exit() have been removed, as they were never more than non-functional stubs and were conflicting with ongoing development.

Removal of Buffer-Pool Based FQ Allocator API

qm_fq_new(), qm_fq_free(), and the QM_FQ_FREE_* flags have been removed, as they were rendered unnecessary and awkward, in particular as they used to support “wait” options that were useful when deallocating FQIDs to a buffer pool but have no sane interpretation with the software-implemented allocator.

Removal of 'struct qman_portal_config::has_stashing' for SDK 1.2

Stashing can now be assumed as enabled in all environments (kernel-space, user-space, with Linux running natively, under the topaz hypervisor, or under KVM), so support for stashing-disabled operation has been removed for the sake of optimisation.

Removal of 'struct qman_fq_cb::dc_ern' for SDK 1.2

The callbacks associated with a frame queue object no longer support a DC_ERN handler (as these never worked properly because the concept is fundamentally unworkable). The lowest level at which DC_ERN messages can meaningfully be handled is at the portal level.

New API for handling DC_ERNs in SDK 1.2

It is possible to register a handler for DC_ERN messages with the portal affine to the running CPU, or as a global fallback for any portals that don't have their own handler.

```
void qman_set_dc_ern(qman_cb_dc_ern handler, int affine);
```

Removal of “NULL FQ” API in SDK 1.2

qman_get_null_cb(), qman_set_null_cb(), and QMAN_INITFQ_FLAG_NULL flag have been removed, as this functionality was considered marginal, had no known use-case, and was conflicting with ongoing development.

New API to Obtain Portal Channel for SDK 1.2

As portals are dynamically allocated, initialised, and assigned to CPUs during boot up, it became necessary for a user to be able to determine the channel ID for the portal associated with a given CPU, eg. in order to schedule frame queues such that dequeues would be handled on that CPU core.

```
enum qm_channel qman_affine_channel(int cpu);
```

New Pool-Channel Allocation API for SDK 1.2

For QMan pool-channel allocation:

```
int qman_alloc_pool_range(u32 *result, u32 count, u32 align, int partial);
static inline int qman_alloc_pool(u32 *result)
{
       int ret = qman_alloc_pool_range(result, 1, 0, 0);
       return (ret > 0) ? 0 : ret;
}

void qman_release_pool_range(u32 id, unsigned int count);
static inline void qman_release_pool(u32 id)
{
       qman_release_pool_range(id, 1);
}
```

New QMan CGR allocation API for SDK 1.2

```
int qman_alloc_cgrid_range(u32 *result, u32 count, u32 align, int partial);
static inline int qman_alloc_cgrid(u32 *result)
{
       int ret = qman_alloc_cgrid_range(result, 1, 0, 0);
       return (ret > 0) ? 0 : ret;
}

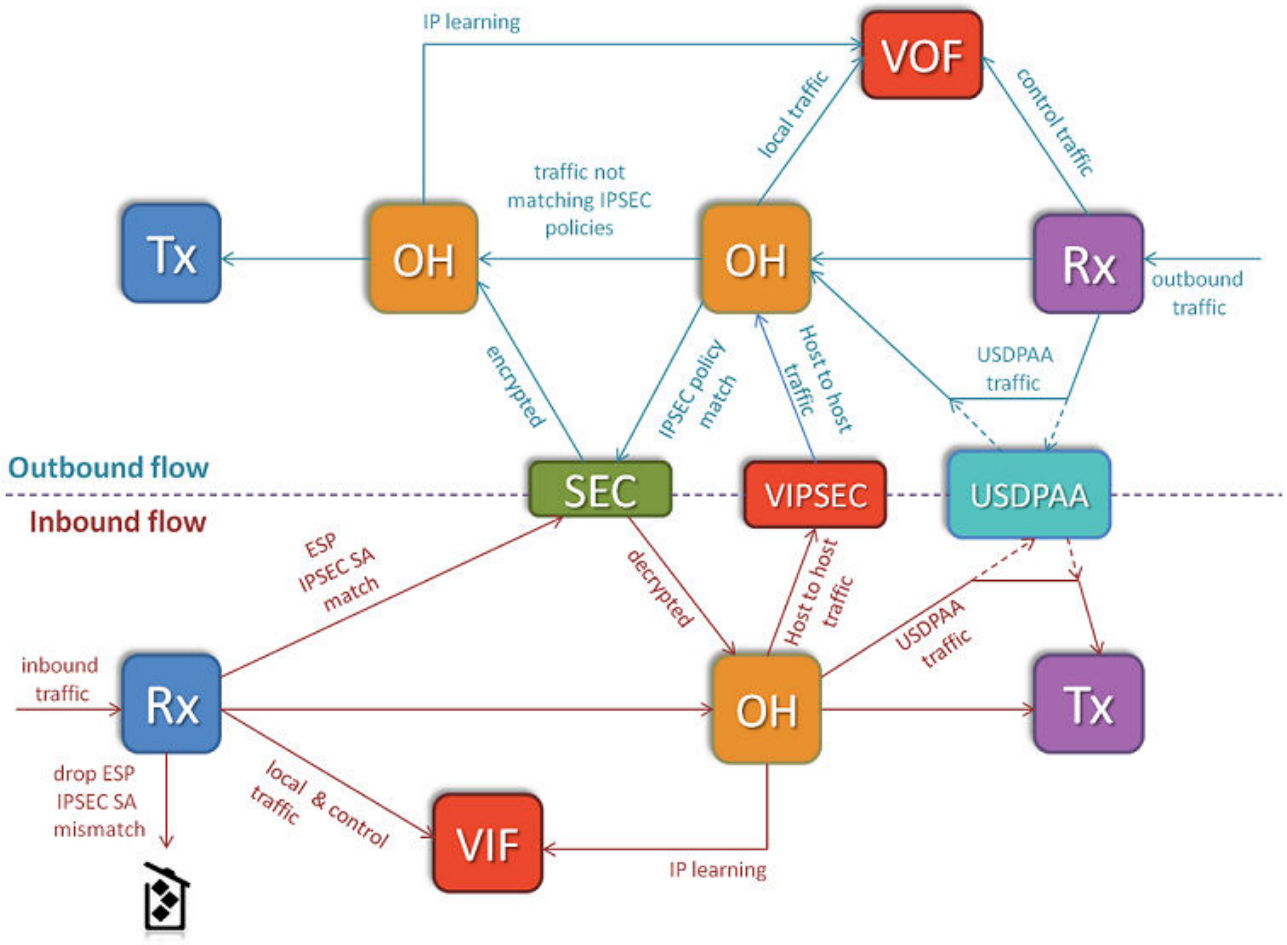
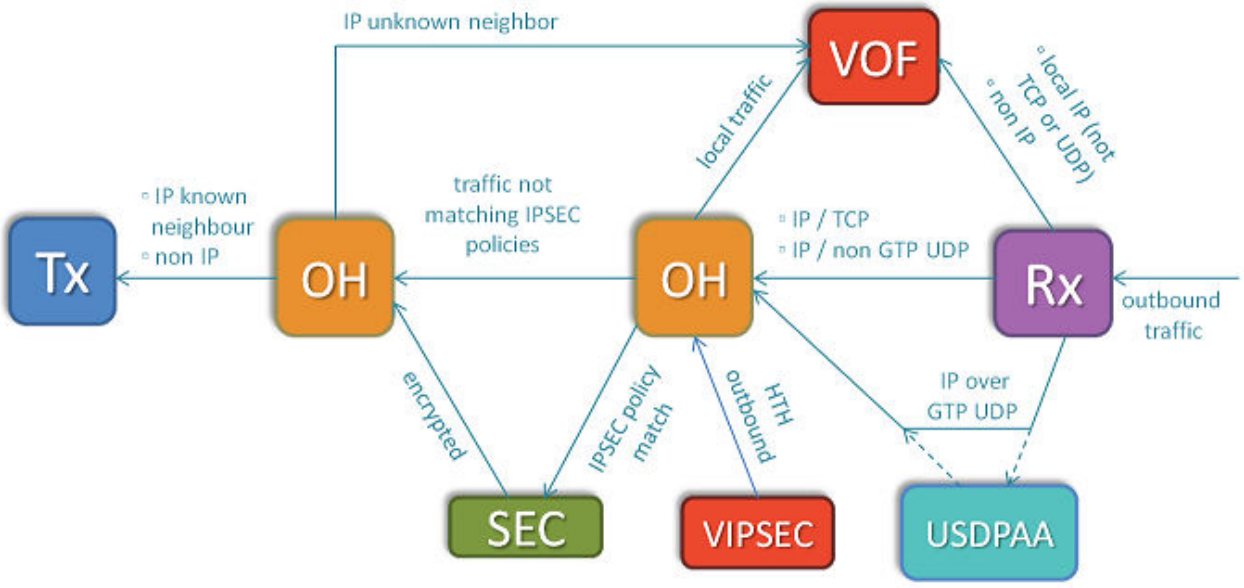
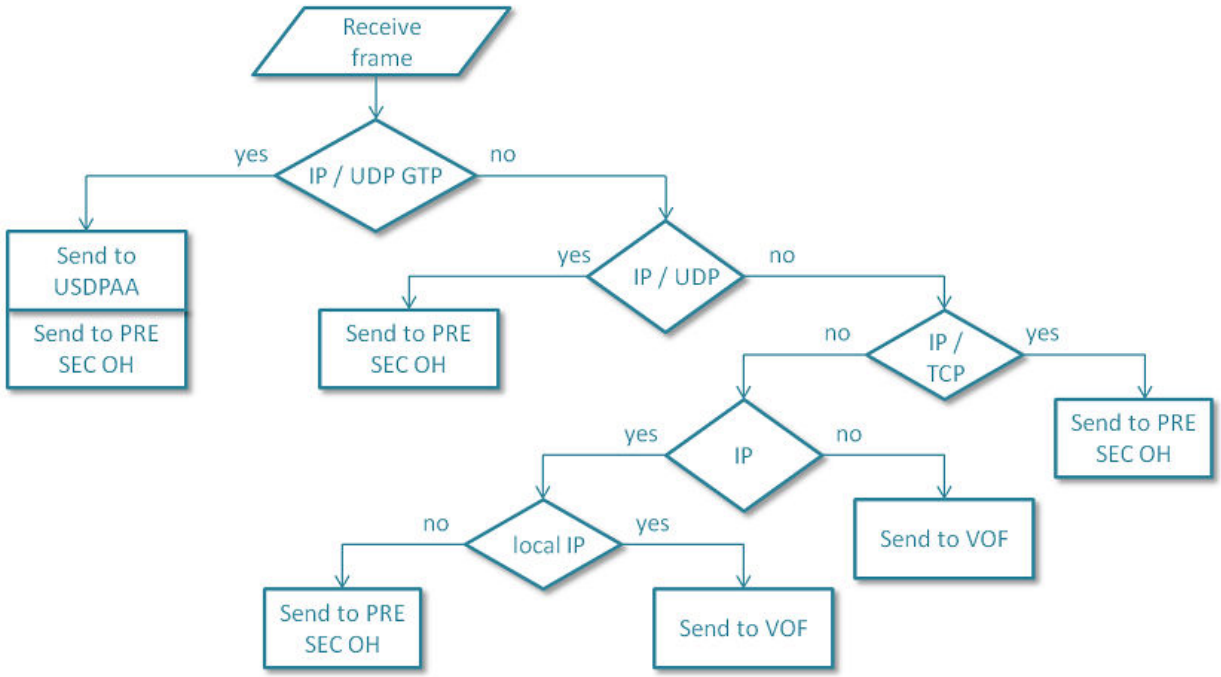
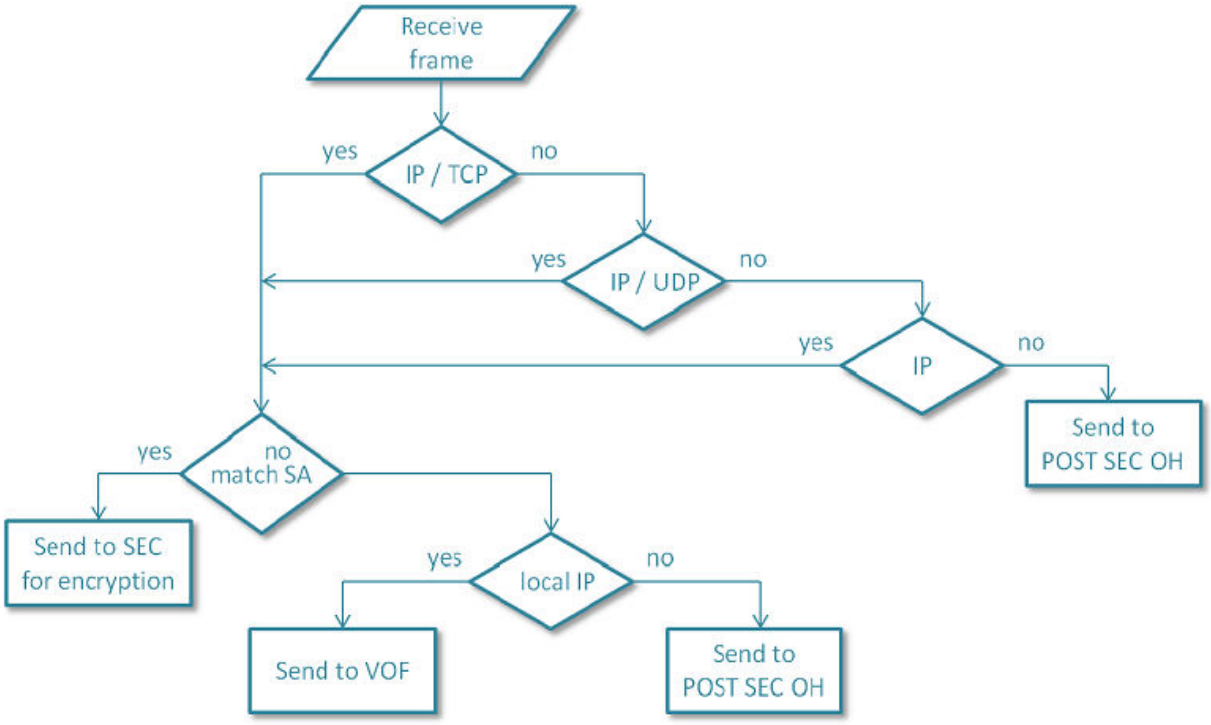
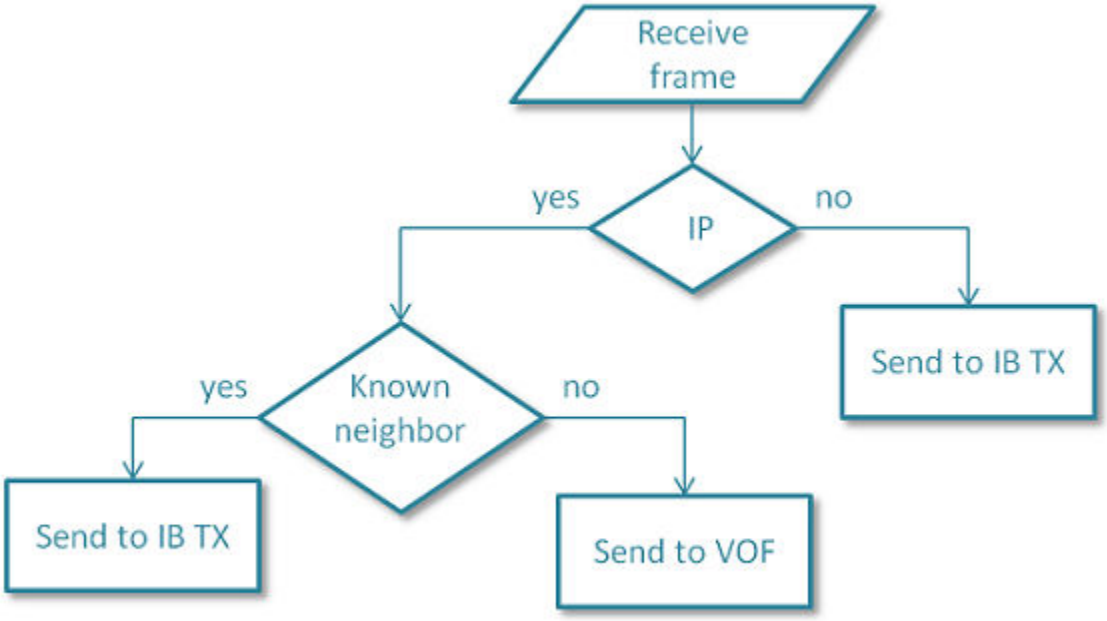
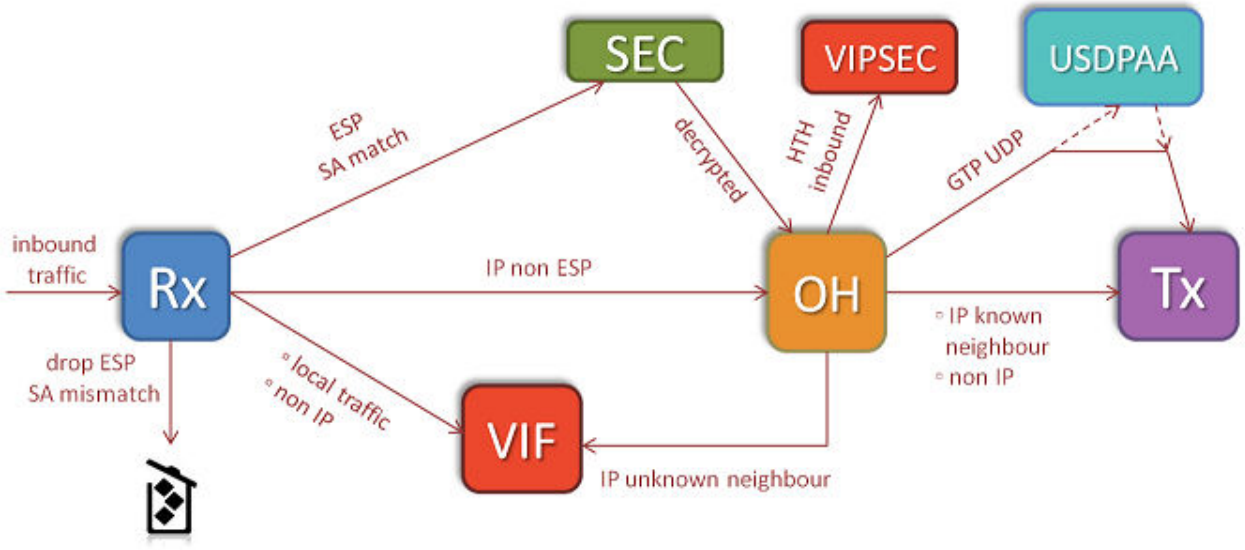
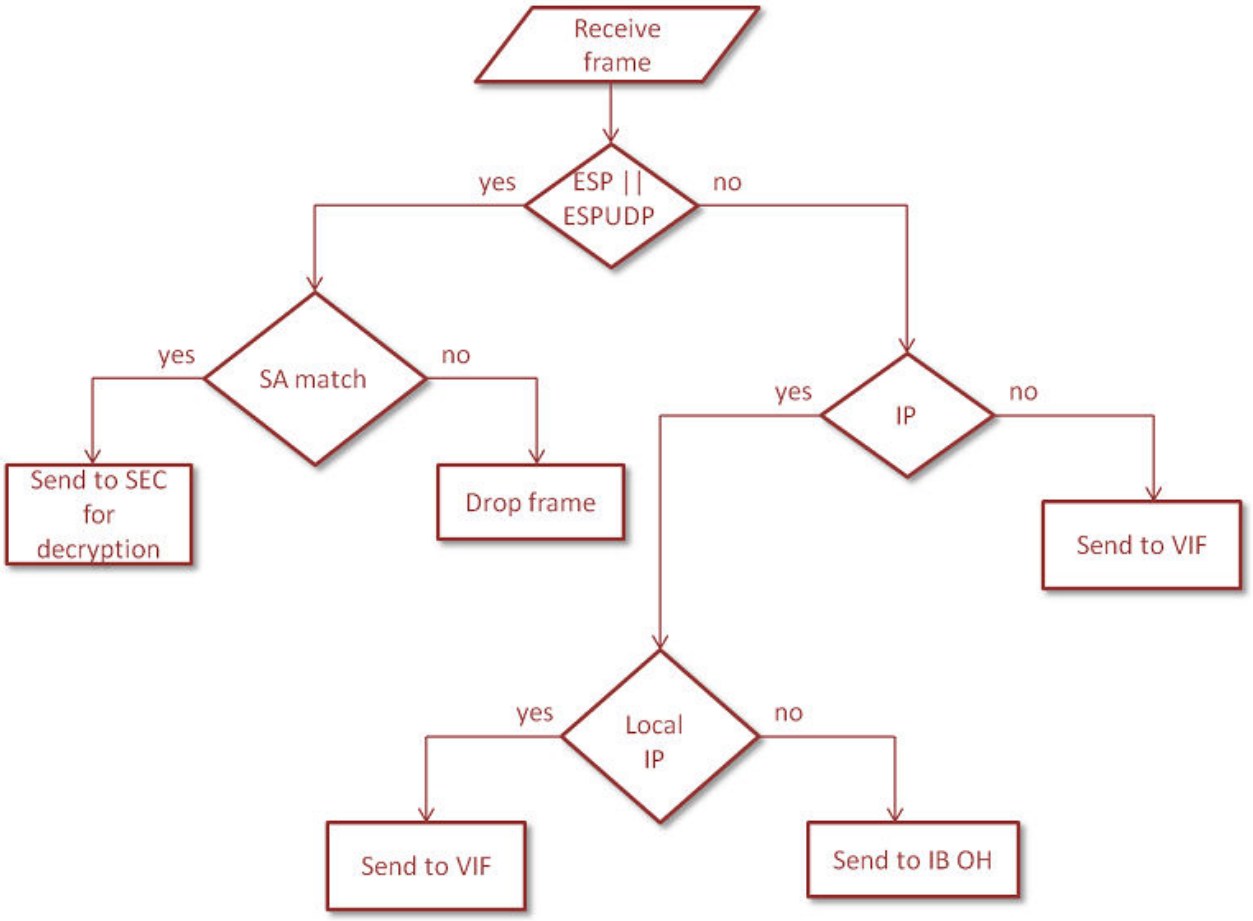
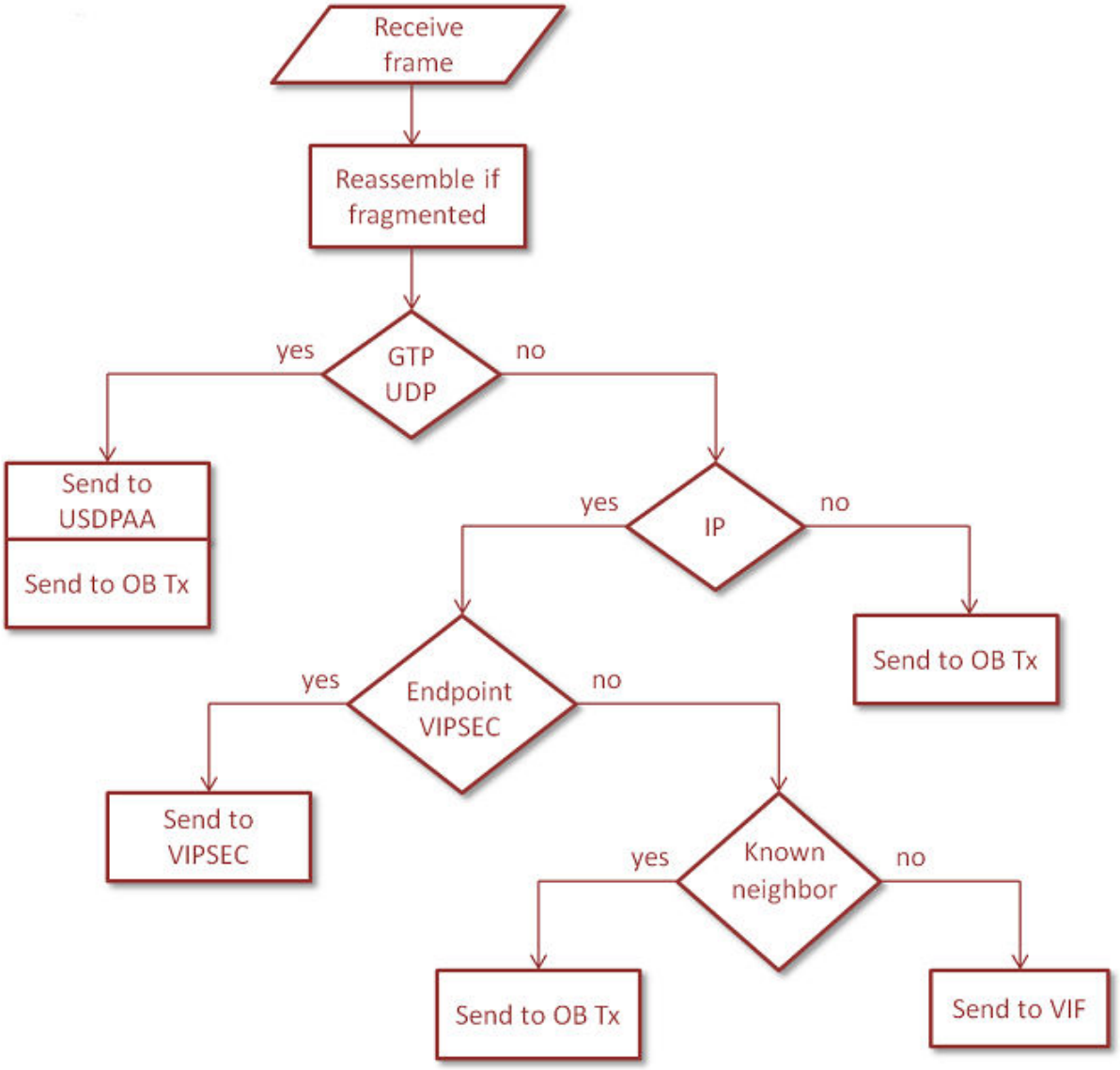
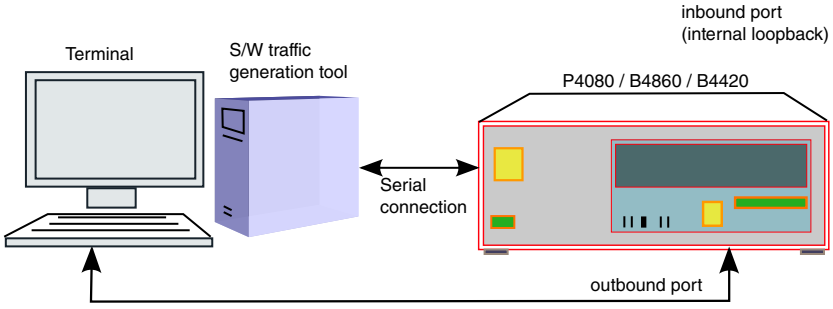
void qman_release_cgrid_range(u32 id, unsigned int count);
static inline void qman_release_cgrid(u32 id)
{
       qman_release_cgrid_range(id, 1);
}
```

9.8.2.5 USDPAA Thread and Global API

The API changes described apply to user-space (USDPAA).

fsl_d.h -- Thread Initialization simplified

The SDK 1.1 Thread initialization was verbose:

```
int qman_thread_init(int cpu, int recovery_mode);   /* remove for SDK 1.2 */
int bman_thread_init(int cpu, int recovery_mode);   /* remove for SDK 1.2 */
int qman_thread_init(void);
int bman_thread_init(void);
```

The SDK 1.2 Thread initialization is compact:

```
int qman_thread_init(void);
int bman_thread_init(void);
```

Recovery support (which was non-functional) has been removed so 'recovery_mode' is no longer a parameter. As for the 'cpu' parameter, portals are now dynamically bound to CPUs, so these functions will allocate any unused portal and it will be automatically bound to the CPU on which the caller is executing.

fsl_d.h -- Global Initialization

The SDK 1.1 Global initialization was verbose:

```
int qman_global_init(int recovery_mode);           /* remove for SDK 1.2 */
int bman_global_init(int recovery_mode);           /* remove for SDK 1.2 */
int qman_global_init(void);
int bman_global_init(void);
```

The SDK 1.2 Global initialization is compact:

```
int qman_global_init(void);
int bman_global_init(void);
```

As before, the 'recovery_mode' parameter is removed because the non-functional recovery interfaces have been removed.

9.8.2.6 USDPAA DMA API

The API changes described apply to DMA user-space (USDPAA).

dma_mem.h

The key change to this interface is that there can now be more than one DMA region available to each USDPAA process, so there is support for creating multiple such mappings, and as such most of the functions require that the DMA region be supplied as a parameter where there was no such need before.

Setup, or creation of DMA maps

The dma_mem driver used to initialise in a parameterless manner, creating the unique DMA map via dma_mem_setup(void), which has been removed. Instead, maps are created by the application using the following interfaces:

```
struct dma_mem;
#define DMA_MAP_FLAG_SHARED   0x01
#define DMA_MAP_FLAG_ALLOC    0x08
#define DMA_MAP_FLAG_NEW      0x02
#define DMA_MAP_FLAG_LAZY     0x04
#define DMA_MAP_FLAG_READONLY 0x10
struct dma_mem *dma_mem_create(uint32_t flags, const char *map_name,
                               size_t len);
```

The significance of the flags is described in more detail within the dma_mem.h header. To summarise, the SHARED flag creates a mapping to a new or existing DMA region that can be mapped by multiple USDPAA processes ('map_name' is the identifier for the region), otherwise a new region and mapping created that remains private to the process. If NEW is not specified the DMA region must already exist, whereas if NEW is specified the region must not already exist unless LAZY is also specified. LAZY refers to “lazy initialisation”, meaning that multiple processes can independently issue the same API call with the same name and specifying both the NEW and LAZY flags, with the result being that the named region will be allocated only once (by whichever process “wins the race”) and mapped into all the requesting processes.

If ALLOC is specified, then all processes that map the same region can use the dma_mem_memalign() and dma_mem_free() interfaces to allocate blocks from the region in a coordinated way. Without ALLOC, the region is created “raw”, meaning the user manipulates the entire region directly without any allocator functionality provided by the dma_mem driver.

Raw Memory Regions

If a DMA region and mapping is created with the RAW flag, it can then be accessed via;

```
void *dma_mem_raw(struct dma_mem *map, size_t *len);
```

Memory allocation

These functions are similar to those in SDK 1.1, with the exception that they require a parameter to indicate which DMA map to use. For SDK 1.1:

```
void *dma_mem_memalign(size_t boundary, size_t size);    /* SDK 1.1 version */
void dma_mem_free(void *ptr, size_t size);               /* SDK 1.1 version */
```

Additional parameter for SDK 1.2:

```
void *dma_mem_memalign(struct dma_mem *map, size_t boundary, size_t size);
void dma_mem_free(struct dma_mem *map, void *ptr);
```

Distinguishing DMA regions

```
struct dma_mem *dma_mem_findv(void *v);
struct dma_mem *dma_mem_findp(dma_addr_t p);
```

Physical/virtual address conversion

As with memory allocation, these functions require a parameter to indicate which DMA map to use, but are otherwise similar to those in SDK 1.1. In the case where the DMA map is not known, use the functions mentioned in “Distinguishing DMA regions” first. Note, these functions are actually implemented as inlines with some nasty details involving casts that should be ignored, this is simply because these routines are performance critical in packet-processing processing;

```
static inline void *dma_mem_ptov(struct dma_mem *map, dma_addr_t p) { … }
static inline dma_addr_t dma_mem_vtop(struct dma_mem *map, void *v) { … }
```

Legacy interfaces, “dma_mem_generic”

In order to facilitate porting of legacy applications to the new dma_mem API, the following mechanism is provided. A global variable within the dma_mem driver, dma_mem_generic, is NULL by default but can be set by the application once it has created a DMA mapping. From that point on, it could use the following __dma_mem_*() functions, which do not require a DMA map parameter.

```
extern struct dma_mem *dma_mem_generic;
static inline void *__dma_mem_ptov(dma_addr_t p)
{
       return dma_mem_ptov(dma_mem_generic, p);
}
static inline dma_addr_t __dma_mem_vtop(void *v)
{
       return dma_mem_vtop(dma_mem_generic, v);
}
static inline void *__dma_mem_memalign(size_t boundary, size_t size)
{
       return dma_mem_memalign(dma_mem_generic, boundary, size);
}
static inline void __dma_mem_free(void *ptr)
{
       return dma_mem_free(dma_mem_generic, ptr);
}
```

That is, in order to port a legacy application (which worked on the assumption of there being a unique, canonical DMA mapping for all DMA operations), it should suffice to;

1. During application initialisation, create a default DMA map using dma_mem_create(), and assign that to dma_mem_generic,
2. Change all the legacy dma_mem_*() calls in the application to __dma_mem_*().

9.8.2.7 USDPAA netcfg.h

The API changes described apply to user-space (USDPAA).

For SDK 1.1 and previous versions:

```
struct usdpaa_netcfg_info {
       uint8_t num_cgrids;             /* SDK 1.1 and previous */
       uint32_t *cgrids;               /* SDK 1.1 and previous */
       uint8_t num_pool_channels;      /* SDK 1.1 and previous */
       enum qm_channel *pool_channels; /* SDK 1.1 and previous */
       uint8_t num_ethports;           /* Number of ports */
       [...]
```

For SDK 1.2 and subsequent versions:

```
struct usdpaa_netcfg_info {
       uint8_t num_ethports;           /* Number of ports */
       [...]
```

'struct usdpaa_netcfg_info' no longer specifies CGR and pool-channel resources to applications. There are now allocators in the QMan API for both these resource types, and they no longer need to come from the network configuration. Note that these resources were previously coming from hard-coded work-arounds if present at all.

See the "USDPAA Resource Management Modifications for SDK 1.2" topic for details on CGRs and QMan Pool Channels.

9.8.2.8 Kernel configuration

Kconfig settings for the fsl_qbman driver have changed.

SDK 1.2 Kconfig changes:

- CONFIG_FSL_DPA_HAVE_IRQ is removed, IRQ support is always enabled.
- CONFIG_FSL_BMAN_PORTAL is removed, support for BMan portals is always enabled.
- CONFIG_FSL_QMAN_PORTAL is removed, support for QMan portals is always enabled.
- CONFIG_FSL_QMAN_PORTAL_DISABLEAUTO_DCA is removed, QMan portals are always enabled for DCA consumption of DQRR (dequeue response ring) entries.
- CONFIG_FSL_QMAN_NULL_FQ_DEMUX is removed, see the BMan and QMan API Modifications topic, Removal of “NULL FQ” API heading.
- CONFIG_FSL_QMAN_DQRR_PREFETCHING is removed, the driver is now optimised to always assume stashing is always enabled, so support for pre-fetching is removed. This removes a run-time check from the critical path.

9.8.2.9 Device Tree (Excluding QMan/BMan Resource Ranges)

Device tree changes, excluding the QMan/BMan resource ranges.

QMan/BMan portals

Portals no longer have “fsl,usdpaa-portal” or “cpu-handle” properties.

```
       qportal1: qman-portal@4000 {
               cell-index = <0x1>;
               compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
               reg = <0x4000 0x4000 0x101000 0x1000>;
               interrupts = <106 0x2 0 0>;
               fsl,qman-channel-id = <0x1>;
               cpu-handle = <&cpu1>;
```

```
        };
```

BPID 0 FQ-allocator

No buffer pool device tree node for BPID 0, because we no longer support that legacy mechanism for FQID allocation.

QMan Pool channels

Pool channel nodes have been removed and replaced by “pool channel range” nodes.

BMan buffer pools

Buffer pool nodes have not been removed, because they are still linked to by network-related nodes. However they are now ignored by the fsl_qbman driver and so no longer contain the “fsl,bpool-cfg” property type. Eventually, network configuration will obtain buffer pools dynamically, at which point there should be no more need for individual buffer pol nodes in the device-tree. (There is no current plan for when this deprecation will occur.)

Ethernet Interfaces

Ethernet nodes no longer have “fsl,qman-channel” properties linking them to pool channels.

```
        ethernet@2 {
                compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";
                fsl,fman-mac = <&enet2>;
```

9.8.2.10 Device Tree (QMan/BMan Resource Ranges)

These device tree resource properties share a common format.

QMan and BMan Portals

Portals no longer have “fsl,usdpaa-portal” or “cpu-handle” properties. For SDK 1.1 and previous:

```
qportal1: qman-portal@4000 {
             cell-index = <0x1>;
             compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
             reg = <0x4000 0x4000 0x101000 0x1000>;
             interrupts = <106 0x2 0 0>;
             fsl,qman-channel-id = <0x1>;
             fsl,usdpaa-portal;                                    /* SDK 1.1 only */
             cpu-handle = <&cpu1>;
             fsl,qman-pool-channels = <&qpool4 &qpool5 &qpool6
                                       &qpool7 &qpool8 &qpool9
                                       &qpool10 &qpool11 &qpool12
                                       &qpool13 &qpool14 &qpool15>; /*SDK 1.1 only */
        };
```

For SDK 1.2 and later versions:

```
qportal1: qman-portal@4000 {
             cell-index = <0x1>;
             compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
             reg = <0x4000 0x4000 0x101000 0x1000>;
             interrupts = <106 0x2 0 0>;
             fsl,qman-channel-id = <0x1>;
             cpu-handle = <&cpu1>;
        };
```

BPID 0 FQ-allocator

No buffer pool device tree node for BPID 0, because we no longer support that legacy mechanism for FQID allocation. See section 3.2.

QMan Pool Channels

Pool channel nodes have been removed (and replaced by “pool channel range” nodes, see the corresponding item below). See section 3.4.

BMan Buffer Pools

Buffer pool nodes have not been removed, because they are still linked to by network-related nodes. However they are now ignored by the fsl_qbman driver and so no longer contain the “fsl,bpool-cfg” property type. Eventually, network configuration will obtain buffer pools dynamically, at which point there should be no more need for individual buffer pol nodes in the device-tree. There is no precise plan for when this deprecation will occur though.

QMan and BMan Resource Ranges (Allocation)

The following resource properties share a common format, with a 2-tuple specifying a base+count pair. Eg. if the pool-channel range property specifies “<0x21 0xf>”, that corresponds to a range of pool channel IDs ranging from 33 (0x21) to 47 (0x21+0xf-1), inclusive.

FQID

“FQID range” nodes have been added to specify FQs that are available for dynamic allocation. These do not yet include all the FQs that are available in the system, because there are still some legacy requirements for pre-configured FQIDs that have not been updated to use dynamic allocation. These nodes were previously supported but were not enabled by the default device trees, whereas they are now used in all cases, via the including of arch/powerpc/boot/dts/fsl/qoriq-dpaa-res*.dtsi files (which did not exist in SDK 1.1) as follows:

```
        qman-fqids@0 {
               compatible = "fsl,fqid-range";
               fsl,fqid-range = <256 256>;
             };
```

CGRID Dynamic Allocation

“CGRID range” nodes have been added to specify CGRs that are available for dynamic allocation. Example:

```
        qman-cgrids@0 {
               compatible = "fsl,cgrid-range";
               fsl,cgrid-range = <0 256>;
             };
```

Pool Channel

“Pool channel range” nodes have been added to specify pool channels that are available for dynamic allocation. Example:

```
       qman-pools@0 {
               compatible = "fsl,pool-channel-range";
               fsl,pool-channel-range = <0x21 0xf>;
       };
```

BPID

“BPID range” nodes have been added to specify buffer pools that are available for dynamic allocation. Example:

```
       bman-bpids@0 {
               compatible = "fsl,bpid-range";
               fsl,bpid-range = <32 32>;
       };
```

9.8.2.11 USDPAA Boot Arguments

Without this boot-argument, no memory is reserved by the “fsl_usdpaa” driver and so no memory will be available for creating DMA mappings in USDPAA applications.

usdpaa_mem

The usdpaa_mem argument is not set by default by u-boot, device-trees or any other resource. The user must add it explicitly to boot commands in order to be able to run USDPAA applications. The size can be expressed using the standard suffixes for memory size notation (“256M”, “1G”, etc).

A second (and optional) argument allows multiple TLB1 indices to be reserved for mapping regions within the memory reservation. Without this extra argument, the default behaviour is to reserve only 1 TLB1 entry. Ie. “usdpaa_mem=64M,1” is equivalent to “usdpaa_mem=64M”. Note that the handling of page faults for software access to these resources will be satisfied by using the reserved TLB1 entries in a round-robin fashion, so if there are more mappings between user-space processes and DMA regions than there are TLB1 entries, and all of those mappings are being actively used at run-time, then performance will degenerate.

Since single TLB1 entries can only map power of 4 memory sizes a set of 1 or more TLB1 entries is required in to map a DMA region into each process. The USDPAA driver will use the minimum number of TLB1 entries possible to map each DMA region.

If a single process maps to two DMA regions, that would require two sets of TLB1 entries. But the same is true if two distinct processes map to one private DMA region each. So the number of TLB1 entries required is really the number of pairings between user-space processes and distinct DMA regions, or “the number of mappings”, to put it another way. The total number of TLB1 entries available depends on the SoC version, so one should check this when determining how many TLB1 entries to dedicate to USDPAA.

Note that a typical kernel would internally only use 3 or so TLB1 entries, and all the remaining entries are normally reserved by the “hugetlb” driver for a similar kind of large-page fault-handling algorithm as that implemented for USDPAA – indeed, the reason USDPAA does not just use “hugetlb” directly is that it has no mechanisms to support user-space obtaining physical addresses and performing DMA.

One must also determine whether HugeTLB support is required. This decision determines the number of TLB1 entries for USDPAA use and, in turn, how many processes and active DMA mappings used. The USDPAA could use most of the TLB1 entries if HugeTLB support is not used.

"qportals" and "bportal;s"

By default, the kernel will attempt to allocate portals for each online core. If however the “qportals” (for QMan portals) and/or “bportals” (for BMan portals) boot-arguments are specified, this behaviour will be overridden. These boot-arguments specify cores that should be assigned portals to them, with the implication being that cores that are not specified will need to “slave” off the cores that are assigned.

Table 229. boot-argument: "qportals=1,s3-4"

Core	Portal	Association
0	B	slave
1	A	affine unshared
2	C	slave
3	B	affine shared
4	C	affine shared
5	B	slave
7	B	slave

9.8.2.12 USDPAA Virtualisation and Partitioning

In a partitioned system, it is necessary to assign each partition non-conflicting subsets of the hardware resources.

All the resources mentioned in section 3 can and should be divided up, with each instance of Linux receiving distinct portals and other dynamically-allocated resources via their respective device-trees.

Some use-cases may intentionally share resources between partitions, in which coordination of the corresponding resource IDs is up to the application. The resources provided to the partitions by the device-tree are inherently managed by the drivers themselves, and these must be mutually-exclusive because the drivers in the distinct partitions have no innate coordination.

9.8.2.13 Multi-process PPAC Applications

A description of how the networking applications are affected by, and have been adapted for, multi-process support.

The USDPAA toolkit provides a template called “PPAC” (Packet Processing Application Core) for simple networking applications, together with some applications called “PPAM”s (Packet Processing Application Module) that are written on top of this template. These networking applications notably include “reflector” and “ipfwd”.

Running multiple distinct PPAM application processes (or “instances” as they are sometimes described) requires that each process have its own dedicated set of FMAN interfaces, buffer pools, and cores.

FMan Interfaces

By default, a PPAM application will automatically configure and use all available FMan interfaces, unless a specific set of interfaces is specified via the “-i” option. For example:

```
<app1> -i fm1-10g,fm2-10g
<app2> -i fm2-gb2,fm2-gb3
```

The names of FMAN interfaces that can be used with “-i” option are as follows.

Table 230. FMan "-i" Options

FMAN1	Fman2
fm1-gb0	fm2-gb0
fm1-gb1	fm2-gb1
fm1-gb2	fm2-gb2
fm1-gb32	fm2-gb3
fm1-gb4	fm2-gb4
fm1-10g	fm2-10g

As per SERDES protocol, a set of FMAN interfaces can be chosen to run with an application.

Buffer pool restrictions

The device tree specifies the ethernet nodes such that each has a set of buffer pools for FMan to use when receiving frames. One restriction of the PPAC multi-process support is that the buffer pools used by FMan interfaces in one process must not also be used by any FMan interfaces in any other process. As such the device tree may need to be adjusted to ensure that any reuse of a buffer pool by more than one FMan interface must only occur when those FMan interfaces will always belong

to the same process. For example, if one application wants to use fm1-10g and fm2-10g and another application is going to use fm2-gb2 and fm2-gb3, then corresponding ethernet nodes might look like:

```
ethernet@4 {
                        compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
                        fsl,bman-buffer-pools = <&bp9>;
                        fsl,qman-frame-queues-rx = <0x5a 1 0x5b 1>;
                        fsl,qman-frame-queues-tx = <0x7a 1 0x7b 1>;
                };

ethernet@9 {
                        compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
                        fsl,bman-buffer-pools = <&bp9>;
                        fsl,qman-frame-queues-rx = <0x66 1 0x67 1>;
                        fsl,qman-frame-queues-tx = <0x86 1 0x87 1>;
                };
ethernet@7 {
                        compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
                        fsl,bman-buffer-pools = <&bp8>;
                        fsl,qman-frame-queues-rx = <0x60 1 0x61 1>;
                        fsl,qman-frame-queues-tx = <0x80 1 0x81 1>;
                };
ethernet@8 {
                        compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
                        fsl,bman-buffer-pools = <&bp8>;
                        fsl,qman-frame-queues-rx = <0x62 1 0x63 1>;
                        fsl,qman-frame-queues-tx = <0x82 1 0x83 1>;
                };
```

Seeding buffer pools

Each PPAC-based application process will initialise the (usually 3) buffer pools used by each FMan interface that belongs to it. (If a buffer pool is used by more than one interface, it will only be initialised once.) By default the number of buffers to allocate for the triplet of pools used by an FMan interface is 0 for the first two pools and 1728 for the third. The default allocation triplet can be overridden via the “–b” option. For example, to continue the earlier example of a two-process scenario, and to have each process allocate 1600 buffers for the first pool used by any network interface and 0 for the second and third pools, they would use the following arguments :

```
<app1> -b 1600:0:0 -i fm1-10g,fm2-10g
<app2> -b 1600:0:0 -i fm2-gb2,fm2-gb3
```

Cores

By default, each PPAC-based application process will start a single thread running on core 1. Additional threads can be started and stopped on arbitrary cores using the interactive CLI, but the first/primary thread can not be removed without ending the process. So for multi-process scenarios, it is better for each application instance to specify a core or set of cores as part of the command line. For example, to split 8 cores in half between our two hypothetical application processes;

```
<app1> 0..3 -b 1600:0:0 -i fm1-10g,fm2-10g
<app2> 4..7 -b 1600:0:0 -i fm2-gb2,fm2-gb3
```

9.8.2.14 Limitations

The use of dynamic allocation of resources does not address the appropriate state of an allocation.

A resource that is deallocated by one user can subsequently be allocated by another and it will be in the same state it was left, for better or worse. As such, stability and integrity of (and between) datapath applications is entirely a matter of cooperation. Future releases will implement measures to quiesce and recover such resources to make the system more robust in the face of individual application failures.

For the reasons explained above, most types of resources can be leaked by USDPAA applications that exit (or crash) before deallocating them. Although an application can explicitly deallocate a resource in a bad state, the risk of a resource being in a bad state when an application exits without having deallocated it is considered too great – so it is leaked rather than allowing it lead to undefined behaviour in future uses.

An exception to the last comment is for QMan and BMan portals, which due to them being UIO-based means they are implicitly “deallocated” when a process exits. In a future release, portals will likely no longer be UIO-based, and in any case, they will likely be “cleaned up” on process-exit.

9.9 USDPAA Applications

9.9.1 USDPAA ceetm Demo User Guide

9.9.1.1 Introduction

This document describes the usage of "ceetm_demo" application which is built on USDPAA PPAC architecture. User can experience the functionality of QMan CEETM with this application and can also learn how to use user space CEETM driver API from source code of this application.

The concept of QMan CEETM is beyond scope of this document. The user should refer to QMan user manual for this part.

9.9.1.2 Overview of ceetm demo

ceetm_demo is an USDPAA application built on PPAC architecture. It works in a way like reflector – the traffic sent to one ethernet interface will be reflected back from the same interface. Please refer to reflector user guide for details. Besides, ceetm_demo enables traffic management on Tx side and differentiates traffic flows according to TOS value in IPv4 header.

In order to provide flexibility at most, we use a xml-format configuration file to depict configuration of CEETM which can be parsed by ceetm_demo at initialization phase. User can experience different cases of traffic management just by changing configuration file before launching ceetm_demo.

If users want to learn how to utilize CEETM driver APIs for their own case, please see function 'net_if_ceetm_init' in ceetm_demo.c as reference.

The ceetm_demo application is currently supported on all B-series and T-series NXP platforms.

9.9.1.3 Features of ceetm demo

ceetm_demo application has following features that provide flexibility to user to experience functionality of QMan CEETM:

- Shaping on LNI and Channel is configurable
- Number of CQs and weighted groups(A and B) is configurable
- SP CQ and Group can be configured for CR/ER eligible or both
- The weight of CQ is configurable
- Distinguish flows by TOS field in IPv4 protocol which is used to identify unique CQ on egress

9.9.1.4 CEETM use case

CEETM configuration may vary from case to case. Here we only show the way to implement one case which is described in QMan 3.x document. Other scenarios could be easily implemented by changing configuration file only.



Figure 238. CEETM use case

This scenario depicts the following:

- Channels #6,#7,#8 and #9 have been configured to be scheduled by the channel scheduler for LNI#3 (i.e. all packets from these channels are directed via LNI#3 to the physical network interface coupled by configuration to LNI#3).
- Channels #6 and #7 have been configure to be “unshaped”. Packets from these channels will not be subjected to shaping at the channel level and will feed the top priority level within the LNI which is also not subjected to shaping. Their class schedulers will not distinguish between CR and ER opportunities.

- Channels #8 and #9 have been configured to be “shaped”. Their class schedulers will distinguish between CR and ER opportunities. The CR/ER packets to be sent form each channel shall be subjected to a pair of CR/ER token bucket shapers specific to that channel. The aggregate of CR/ER packet from these channels shall be subject to a pair of CR/ER token bucket shapers specific to LNI#3.
- Channels #6 has only one class in use. That class queue will behave as if it were a channel queue and a peer to channel #7. Unused classes do not have to be configured as such - simply not used.
- Channel #7 has all 16 classes in use
  - The group classes have been configured as 2 groups (A and B) of 4 classes.
  - The priority of the groups A and B have both been set to be immediately below independent class 5. In a case of similar configuration group A has higher priority than group B.
- Channels #8 has 3 independent classes and 2 groups of 4 grouped classes in use.
  - The priorities of the class groups A and B have been set to be immediately below independent class 0 and class 2 respectively.
  - Independent class 0 and class group A have been configured to request and fulfill only CR packet opportunities.
  - Independent class 1 has been configured to request and fulfill both CR and ER packet opportunities.
  - Independent class 2 and class group B have been configured to request and fulfill only ER packet opportunities.
- Channels #9 has 4 independent classes and 1 groups of 8 grouped classes in use.
  - The group classes have been configured as 1 group (A) of 8 classes.
  - All independent classes and the class group (A) have been configured to request and fulfill both CR and ER packet opportunities.

9.9.1.5 CEETM configuration file

CEETM configuration file is used to depict a model of traffic management in xml format which can then be parsed by ceetm_demo application. User could experience the effect of dual-shaper and dual-scheduler of CEETM by changing this configuration file.

```
<ceetm>
    <lni control="shaped" cr="2g" er="2g">
        <channel control="unshaped" group="0">
            <cq idx='0'/>
        </channel>
        <channel control="unshaped" group="2">
            <groupA idx="6" />
            <groupB idx="6" />
            <cq idx='0' />
            <cq idx='1' />
            <cq idx='2' />
            <cq idx='3' />
            <cq idx='4' />
            <cq idx='5' />
            <cq idx='6' />
            <cq idx='7' />
            <cq idx="8" weight=”1” />
            <cq idx="9" weight=”1” />
            <cq idx="10" weight=”1” />
            <cq idx="11" weight=”1” />
            <cq idx="12" weight=”1” />
            <cq idx="13" weight=”2” />
            <cq idx="14" weight=”4” />
            <cq idx="15" weight=”8”/>
```

```
        </channel>
        <channel control="shaped" group="2" cr="500m" er="50m">
            <groupA idx="1" op="cr" />
            <groupB idx="3" op="er" />
            <cq idx='0' op="cr" />
            <cq idx='1' op="both" />
            <cq idx='2' op="er" />
            <cq idx="8" weight=”1” />
            <cq idx="9" weight=”1”  />
            <cq idx="10" weight=”1” />
            <cq idx="11" weight=”1” />
            <cq idx="12" weight=”1” />
            <cq idx="13" weight=”1” />
            <cq idx="14" weight=”1” />
            <cq idx="15" weight=”1” />
        </channel>
        <channel control="shaped" group="1" cr="250m" er="100m">
            <groupA idx="2" op="both" />
            <cq idx='0' op="both" />
            <cq idx='1' op="both" />
            <cq idx='2' op="both" />
            <cq idx='3' op="both" />
            <cq idx="8" weight=”1” />
            <cq idx="9" weight=”2” />
            <cq idx="10" weight=”4” />
            <cq idx="11" weight=”8” />
            <cq idx="12" weight=”16” />
            <cq idx="13" weight=”32” />
            <cq idx="14" weight=”64” />
            <cq idx="15" weight=”128” />
        </channel>
    </lni>
</ceetm>
```

- <ceetm>: Root element
- <lni>
  - control: ‘shaped’ if shaping is placed on this interface; ‘unshaped’ otherwise
  - cr: bandwidth for Commit Rate of interface
  - er: bandwidth for Excess Rate of interface
- <channel>
  - control: ‘shaped’ if shaping is placed on this interface; ‘unshaped’ otherwise
  - grou: '0' – no group; '1' – only groupA enabled; '2' – both groupA/B are enabled
  - cr: bandwidth for Commit Rate of channel
  - er: bandwidth for Excess Rate of channel
- <cq>
  - idx: CQ id, ranging from 0 to 15
  - op: 'cr' – cr eligible; 'er' – er eligible; or 'both'. Only applicable for SP CQ
  - weight: the weight of CQ. Only applicable for grouped CQ

9.9.1.6 Running ceetm_demo

User should boot board with USDPAA mode. After logging into the board as 'root', run following command

```
          login: root
          Password:
          root@t2080qds:~# cd /usr/etc
          root@t2080qds:/usr/etc# fmc -c usdpaa_config_t2_serdes_66_16.xml -p
usdpaa_policy_hash_ipv4.xml -a
          root@t2080qds:/usr/etc# ceetm_demo -c usdpaa_config_t2_serdes_66_16.xml -p
usdpaa_policy_hash_ipv4.xml -f ceetm_cfg.xml
          Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
          Found /fsl,dpaa/ethernet@0, Tx Channel = 802, FMAN = 0, Port ID = 2
          Found /fsl,dpaa/ethernet@1, Tx Channel = 803, FMAN = 0, Port ID = 3
          Found /fsl,dpaa/ethernet@2, Tx Channel = 804, FMAN = 0, Port ID = 2
          Found /fsl,dpaa/ethernet@3, Tx Channel = 805, FMAN = 0, Port ID = 3
          Found /fsl,dpaa/ethernet@8, Tx Channel = 800, FMAN = 0, Port ID = 0#
          Found /fsl,dpaa/ethernet@9, Tx Channel = 801, FMAN = 0, Port ID = 1
          Configuring for 4 network interfaces
          Allocated DMA region size 0x1000000
          Released 0 bufs to BPID 7
          Released 0 bufs to BPID 8
          Released 8192 bufs to BPID 9
          Thread uid:0 alive (on cpu 1)
          ceetm> add 2..7
          Thread uid:1 alive (on cpu 2)
          Thread uid:2 alive (on cpu 3)
          Thread uid:3 alive (on cpu 4)
          Thread uid:4 alive (on cpu 5)
          Thread uid:5 alive (on cpu 6)
          Thread uid:6 alive (on cpu 7)
          ceetm>
```

When application starts, only 1 thread runs. User can add more threads by 'add' command to improve throughput, or specify cpu range on which thread runs. For example

```
          root@t2080qds:/usr/etc# ceetm_demo 2..7 -c usdpaa_config_t2.xml -p
usdpaa_policy_hash_ipv4.xml -f ceetm_cfg.xml
```

In this case, 6 threads run after application starts.

The PCD configuration file may vary from board to board, here is summary of file name for boards that support CEETM.

```
          B4860QDS: usdpaa_config_b4_serdes_0x2a_0x98.xml
          T4240QDS: usdpaa_config_t4_serdes_1_1_6_6.xml
          T2080QDS: usdpaa_config_t2_serdes_66_16.xml
```

9.9.1.7 Generate trafflic flows

ceetm_demo works in a way like 'reflector'. It enhances traffic differentiation on egress side. The egress flow will be forwarded to class queue according to ‘tos’ value in IPv4 header.

TOS is a 8-bit value, 4 msb in this case means CHANNEL id within a LNI while 4 lsb means CQ id (0 – 0xf) within a CHANNEL. So if you want to egress traffic flow through CQ 10 in channel 2, the tos vaule should be 0x2a.

User can see the effect of packet scheduling by sending traffic flows with different TOS values.

Currently QMan CEETM supports dual-rate shapers and dual-scheduler on two levels. Please refer to QMan user guide for elaboration about them.

9.9.2 DPAA Offloading Applications Users Guide

9.9.2.1 Introduction

This document supplies several methods for modifying the QorIQ USDPAA offloading applications using the DPA offloading driver API.

The document explores the following target use cases providing detailed and interactive examples. These include:

- USDPAA classifier demo application
- USDPAA IP fragmentation demo application
- USDPAA IP reassembly demo application
- USDPAA IPSec offloading application
- USDPAA IPSec & IP Forwarding offloading application using Network Function Layer

See Appendix A of the DPAA Offloading Applications User Guide for list of supported processors.

The following table provides a short summary of each example.

Table 231. Summary of USDPAA offloading applications

Application	Description
USDPAA classifier demo application	Provides an example on how to use basic frame manager features such as parse, classify, distribution and header manipulation.
USDPAA IP fragmentation demo application	Illustrates how to configure basic IP fragmentation provided by the Frame Manager. For frame manager version 3 devices the reassembly application also demonstrates the Virtual Storage Profile configuration and selection for reassembled flows.
USDPAA IP reassembly demo application	Explains how to configure basic IP reassembly provided by the Frame Manager. For Frame Manager version 3 devices the reassembly application also demonstrates the Virtual Storage Profile configuration and selection for reassembled flows.
USDPAA IPSec offloading application	Reveals how to use standard Linux tools to configure IPSec offloading using the DPA offloading driver.
USDPAA IPSec & IP forwarding offloading application using Network Function Layer	Demonstrates how IPSec and IP forwarding services can be offloaded using the DPAA offloading *Network Function Layer*

Before using any of the applications in this document, you need to build and install the DPAA offloading drivers as described in section "*Appendix B - Enabling DPA Offloading Drivers in the Linux Kernel*".

9.9.2.2 Altering the classifier_demo application

This application demonstrates how packet classification can be offloaded to the frame manager by using the DPA offloading driver. It shows several functionalities of the classifier, traffic manager(on CEETM capable platforms) and statistics components.

9.9.2.2.1 Overview

The application performs the following operations on the created resources (a set of three DPA Classifier Exact Match tables managed by key and a group of DPA Stats counters):

- Create an Exact Match Table
- Insert entry in an Exact Match Table
- Delete entry by Key in an Exact Match Table
- Flush an Exact Match Table
- Free an Exact Match Table
- Dynamically create header manipulations
- Set up header manipulation on a table entry
- Free the header manipulation associated with a table entry
- Create a DPA Stats class counter for the entire Exact Match Table for each table created
- Retrieve synchronously and asynchronously the DPA Stats counters
- Remove all created DPA Stats counters



Figure 239. Classifier Demo Application Scheme

The application has defined a number CLI commands, each is meant to validate different functionalities. The user can add or remove classification entries by typing the command “add4” for IPv4, respectively “add6” for IPv6. For removing the inserted entries, the commands “rem4”, or “rem6” must be used in the USDPAA PPAM command line.

When a frame is received, it is first distributed based on protocol, then if the table lookup process results in a HIT condition, the frame is sent to the next classifier table where, based on DSCP value, the frame is forwarded to a logical frame queue corresponding to a traffic manager class queue in case of CEETM capable platforms. At the traffic manager level, based on priorities, the frame is forwarded or not to the TX port. On P platforms the frame is enqueued on the first fqid configured for the TX port.

A header manipulation chain is attached to each of the entries which are inserted in the IPv4/IPv6 Exact Match table. Each header manipulation chain consists of a TTL decrement, a PAT (Port Address Translation) and a forwarding header manipulation (L2 header update). The values in each header manipulation operation are updated at runtime based on the information passed by the user along with the add4/6 command.

During the life cycle of the application, the user has the possibility to perform insert or delete classification keys or to read values of the DPA Stats class counters by entering the following commands in the USDPAA PPAM command line:

Table 232. Application

Command Line	Description
add4 <IPSA><IPDA><PROTO><new-MACDA><new-SPORT><new-DPORT>	dynamically insert a new entry in 3_tuple_ipv4_classif Exact Match table.
add6 <IPSA><IPDA><PROTO><new-MACDA><new-SPORT><new-DPORT>	dynamically insert a new entry in 3_tuple_ipv6_classif Exact Match table.
rem4 <IPSA><IPDA><PROTO>	dynamically remove an entry from 3_tuple_ipv4_classif Exact Match table.
rem6 <IPSA><IPDA><PROTO>	dynamically remove an entry from 3_tuple_ipv6_classif Exact Match table.
get_classif_stats	retrieve the statistics for the created classifier table counters in asynchronous mode and print the returned values.
get_classif_stats_sync	retrieve the statistics for the created classifier table counters in synchronous mode and print the returned values.
get_traffic_stats	retrieve the statistics for the created traffic manager/ethernet counters in asynchronous mode and print the returned values.
get_traffic_stats_sync	retrieve the statistics for the created traffic manager/ethernet counters in synchronous mode and print the returned values.
reset_stats	reset the statistics for all created DPA Stats counters.

In order to have a successful operation, the user must verify that the returned values of the counters statistics matches the number of sent frames.

Classifier demo is a USDPAA application that requires the following resources:

- `classif_demo_config.xml` - config file
- `classif_demo_policy.xml` - policy file

This user space application initializes the classification schemes using the FMC library, and afterwards it manages the DPA Classifier table.

The application receives traffic on the configured Ethernet port. The traffic is split in IPv4 and IPv6 flows, and then lookups are performed in the DPA Classifier table. If the result is HIT, the frame is sent to the next configured classifier table, the dscp_classif.

On B4 platforms, at this point, based on DSCP, the frame is enqueued to the appropriate logical frame queue. The traffic manager based on its internal algorithms decides to send or not the packet back to be transmitted on the interface from which it was received.

On the other hand, in case of P4 platforms, the traffic is classified based on the DSCP table entries, shown in the tables below, and then enqueued on the fqid configured for the TX port used by the application.

If the result is MISS, the frame is silently dropped in both cases.

The application creates a two Exact Match tables managed by key, none of them contains prefilled entries, but the user can dynamically insert or remove keys with add4/6 or rem4/6 PPAM CLI commands. Each table supports a maximum of 64 keys. Each key is composed from the following parts: IPSA, IPDA and protocol. The add commands have an additional field MACDA that is used for the forwarding header manipulation.

Another Exact Match table is populated by the application with 12 entries that represent the main QOS traffic classes AFxy. Those entries are static, defined in the application and cannot be modified. For each traffic class a CEETM channel is defined, each channel has three CEETM class queues (corresponding to the level of drop precedence) and a class congestion group (that aggregates all dropped traffic) defined. Due to lack of CEETM support on P4 platforms all traffic is sent from the DSCP classification table to the first fqid configured on the TX port and the table structure is the same as represented bellow.

Table 233. Application QOS Table (Classifier Exact Match Table)

Channel	Class Queue	QOS	DSCP	TOS	Entry
CH0	CQ0	AF11	0x0A	0x28	0
CH0	CQ1	AF12	0x0C	0x30	1
CH0	CQ2	AF13	0x0E	0x38	2
CH1	CQ0	AF21	0x12	0x48	3
CH1	CQ1	AF22	0x14	0x50	4
CH1	CQ2	AF23	0x16	0x58	5
CH2	CQ0	AF31	0x1A	0x68	6
CH2	CQ1	AF32	0x1C	0x70	7
CH2	CQ2	AF33	0x1E	0x78	8
CH3	CQ0	AF41	0x22	0x88	9
CH3	CQ1	AF42	0x24	0x90	10
CH3	CQ2	AF43	0x26	0x98	11

NOTE

Classifier demo application does not use the cores or the IP stack for performing classification (look-up actions) or header manipulation operation (protocol operations). It uses the frame manager features for traffic classification and header manipulation.

9.9.2.2.2 Running classifier_demo

Classifier_demo application supports the following platforms:

- P2041
- P4080
- B4860
- B4420
- T4240
- T2080
- LS1043A

To run the application, you may need to use a specific device tree file that comes with the Linux DPA offloading driver. The following shows you how to produce this device tree file for your platform.

9.9.2.2.2.1 Application environment specifications

This application uses the setup pictured bellow with the following connections:

Table 234. Port Connections

SoC	Traffic Port
P4080	`fm1-mac0`
P2041	`fm0-mac1`
B4860	`fm0-mac5`
B4420	`fm0-mac3`
T4240	`fm0-mac4`
T2080	`fm0-mac3`
LS1043A	`fm0-mac0`



Figure 240. Use Case Set-up

If you want *classifier_demo* application to use a different traffic port than the ones listed above, you need to update the XML port configuration file for your platform and to use the proper command line arguments for your new traffic port. Please refer to the paragraph **Application start-up and configuration** for further details.

9.9.2.2.2.2 Application start-up and configuration

Use the steps described later in **Compiling the Device Tree for USDPAA Applications** to generate a device tree binary file. Boot the board with the compiled kernel, `arch/powerpc/boot/uImage`, and the `DTB` file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

The following commands assume that the application runs on a B4860QDS board. When running on a different supported platform, please use the proper configuration file and the parameters listed in the section above. Run these commands to set up the USDPAA network configuration and PCD resources used by the application:

```
export DEF_CFG_PATH=/usr/etc/classifier_demo_config-b4860.xml
export DEF_PCD_PATH=/usr/etc/classifier_demo_policy.xml
export DEF_PDL_PATH=/etc/fmc/config/hxs_pdl_v3.xml
```

First ensure that the following device files have been created:

- `/dev/dpa_classifier`
- `/dev/dpa_stats`

The `classifier_demo` can be run with the following command:

```
/usr/bin/classifier_demo -f 0 -t 6
```

The following arguments specify the frame manager ports used by the application:

- `-f [FMan index]`
- `-t [Traffic Port index]`

The output displayed by the application would be something similar with the following:

```
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/dpa-fman0-oh@3, Tx Channel = 80b, FMAN = 0, Port ID = 2
Found /fsl,dpaa/dpa-fman0-oh@4, Tx Channel = 80c, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@4, Tx Channel = 806, FMAN = 0, Port ID = 4
```

```
Found /fsl,dpaa/ethernet@5, Tx Channel = 807, FMAN = 0, Port ID = 5
Found /fsl,dpaa/ethernet@16, MAC-LESS node
Found /fsl,dpaa/ethernet@17, MAC-LESS node
Configuring for 1 network interface
Allocated DMA region size 0x1000000
dpa_classifier_demo: using the following config file: /usr/etc/classifier_demo_config-b4860.xml
dpa_classifier_demo: using the following PCD file: /usr/etc/classifier_demo_policy.xml
dpa_classifier_demo: using the following PDL file: /etc/fmc/config/hxs_pdl_v3.xml
cpu hotplug daemon not running: No such file or directory
dpa_classifier_demo is assuming FMan:0 and port:6
Successfully initialized CEETM resources for FMan 0 Port 6

Successfully CREATED DPA Classifier DSCP table (td=0)
Successfully CREATED DPA Classifier Exact Match table (td=1).
Successfully Modified Miss Action for DPA Classifier Exact Match table (td=1).
Successfully CREATED DPA Classifier Exact Match table (td=2).
Successfully Modified Miss Action for DPA Classifier Exact Match table (td=2).
DPA Stats library successfully initialized

Successfully Initialized DPA Stats instance: 0
Successfully created DPA Stats counter: 0
Successfully created DPA Stats counter: 1
Successfully created DPA Stats counter: 2
Successfully created DPA Stats counter: 3
Successfully created DPA Stats counter: 4
Successfully created DPA Stats counter: 5
Released 0 bufs to BPID 7
Released 0 bufs to BPID 8
Released 8192 bufs to BPID 9
Thread uid:0 alive (on cpu 1)
```

9.9.2.2.2.3 Running the application

At this point the user can start sending traffic with the traffic generator.

By running a set of add4 and add6 commands after the application is started, the user can populate the IPv4 and IPv6. The commands for populating the IPv4 table are:

```
➢ add4 192.168.1.1 128.224.10.10 tcp 44:00:00:00:00:01 881 921
```

```
➢ add4 192.168.25.1 128.224.20.10 udp 44:00:00:00:00:02 882 922
```

```
➢ add4 192.168.50.1 128.224.30.10 TCP 44:00:00:00:00:03 883 923
```

```
➢ add4 192.168.75.1 128.224.40.10 UDP 44:00:00:00:00:04 884 924
```

```
➢ add4 192.168.100.1 128.224.50.10 6 44:00:00:00:00:05 885 925
```

```
➢ add4 192.168.125.1 128.224.60.10 17 44:00:00:00:00:06 886 926
```

The commands for populating the IPv6 table are:

```
➢ add6 3ffe:1944:0100:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:010b TCP
66:00:00:00:00:01 6011 501
```

```
➢ add6 3ffe:1944:0200:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:020b UDP
66:00:00:00:00:02 6012 502
```

```
➢ add6 3ffe:1944:0300:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:030b 6
66:00:00:00:00:03 6013 503
```

```
➢ add6 3ffe:1944:0400:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:040b 17
66:00:00:00:00:04 6014 504
```

```
➢ add6 3ffe:1944:0500:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:050b tcp
66:00:00:00:00:05 6015 505

➢ add6 3ffe:1944:0600:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:060b udp
66:00:00:00:00:06 6016 506
```

After entering those commands the content of the DPA Classifier Exact Match tables will be:

Table 235. IPv4 Classification Table (Classifier Exact Match Table)

Flow	IP SRC	IP DST	Protocol
1	192.168.1.1	128.224.10.10	TCP
2	192.168.25.1	128.224.20.10	UDP
3	192.168.50.1	128.224.30.10	TCP
4	192.168.75.1	128.224.40.10	UDP
5	192.168.100.1	128.224.50.10	TCP
6	192.168.125.1	128.224.60.10	UDP

Table 236. IPv6 Classification Table (Classifier Exact Match Table)

Flow	IP SRC	IP DST	Protocol
1	3ffe:1944:0100:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:010b	TCP
2	3ffe:1944:0200:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:020b	UDP
3	3ffe:1944:0300:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:030b	TCP
4	3ffe:1944:0400:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:040b	UDP
5	3ffe:1944:0500:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:050b	TCP
6	3ffe:1944:0600:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:060b	UDP

By sending with a traffic generator flows identically to the ones described in the tables above, the user will notice that the traffic is received back with the TTL decremented and the MACDA, L3 source port and destination ports updated as specified in the *add4* / *add6* command that inserted the entry. On P4 platforms, all traffic is received back. If additional flows are defined in the traffic generator configuration, the miss statistics will also be increased. Based on the traffic sent the user can read the statistics by running the *get_classif_stats*/*get_classif_stats_sync* or *get_traffic_stats*/*get_traffic_stats_sync* commands. In order to have a successful operation, the user must verify that the returned values of the counter statistics matches the number of sent frames.

In order to shut down the application you can type *quit* in the application console.

9.9.2.3 Adapting the fragmentation_demo application

The application demonstrates how IP fragmentation can be offloaded to the FMan by using the direct XML configuration and the DPA offloading driver.

9.9.2.3.1 Overview

The IP fragmentation function works strictly on offline ports and performs fragmentation of IPv4 and IPv6 packets into IP fragments according to RFC 791 and RFC 2460. The DPA Stats component usage is highlighted in the same application by creating single and class counters for the IP Fragmentation packet header manipulation.

The application initializes the IP fragmentation and the Parse-Classify-Police-Distribute (PCD) description through the use of the FMC Library.

Traffic is received on a backhaul Rx port (BP) by USDPAA fragmentation_demo application. IP packets that pass the Parse-Classify and Distribute descriptor configured on the Rx port are received in the application. The application swaps the Ethernet MAC addresses and enqueue the traffic in the offline port. This port performs a classification based on VLAN and, depending on the value, it performs IP fragmentation if the frame size is larger than a pre-configured value. Afterwards it forwards the obtained fragments back to the Tx queue of the BP port.

The user has the possibility of performing a number of actions on the DPA Stats counters by typing the following commands in the USDPAA PPAM command line:

Table 237. USDPAA PPAM commands

Command Line	Description
get_stats	retrieve the statistics for the created counters in an asynchronous mode and print the returned values.
get_stats_sync	retrieve the statistics for the created counters in a synchronous mode and print the returned values.
reset_stats	reset the statistics for the created counters.

To have a successful operation, you should verify that the returned values of the counters statistics matches the number of sent frames.

NOTE

The fragmentation demo application is different from the fragmentation performed in the Linux IP stack because the demo does not use the cores or the IP stack for performing fragmentation operation. It uses the frame manager for fragmentation processing.

9.9.2.3.2 Running fragmentation_demo

fragmentation_demo application supports the following platforms:

- P2041
- P4080
- B4860
- B4420
- T4240
- T2080
- LS1043A

In order to run it you may need to use a specific device tree file that comes with the DPA offloading driver. Please read below about how to produce this device tree file for your platform.

9.9.2.3.2.1 Application environment specifications

The environment is the same as described for the classifier_demo application. Please see **Classifier_demo: Application environment specifications.**

If you need the *fragmentation_demo* application to run on a different traffic port than the defaults, you will need to adapt the XML port configuration file for your platform and use the proper command line arguments for your new traffic port. Please refer to the paragraph **Application start-up and configuration** for further details.

9.9.2.3.2.2 Application start-up and configuration

Use the steps described later in **Appendix A: Compiling the Device Tree for IP Offloading** to generate a device tree binary file. Boot the board with the compiled kernel, `arch/powerpc/boot/uImage`, and the `DTB` file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

The following commands assume that the application runs on a B4860QDS board. When running on a different supported platform, please use the proper parameters listed in **Running Classifier_demo: Application environment specifications**. For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH=/usr/etc/fragmentation_demo_config-b4860.xml
export DEF_PCD_PATH=/usr/etc/fragmentation_demo_policy.xml
export DEF_PDL_PATH=/etc/fmc/config/hxs_pdl_v3.xml
```

First ensure that the following device files have been created:

- `/dev/dpa_stats`

The `fragmentation_demo` can be run with the following command:

```
/usr/bin/fragmentation_demo -f 0 -t 6 -o 2
```

The following arguments specify the FMan ports used by the application:

- `-f [FMan index]`
- `-o [Offline Port index]`
- `-t [BP Rx port index]`

The output displayed by the application would be something similar with the following:

```
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@0, Tx Channel = 802, FMAN = 0, Port ID = 0
Found /fsl,dpaa/ethernet@1, Tx Channel = 803, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@2, Tx Channel = 804, FMAN = 0, Port ID = 2
Found /fsl,dpaa/ethernet@5, Tx Channel = 807, FMAN = 0, Port ID = 5
Configuring for 2 network interfaces
Allocated DMA region size 0x1000000
Released 0 bufs to BPID 4
fragmentation_demo is assuming FMan:0 and eth:6 and offline port:2
Released 0 bufs to BPID 7
Released 0 bufs to BPID 8
Warn: drained 8192 bufs from BPID 9
Released 8192 bufs to BPID 9
```

```
Thread uid:0 alive (on cpu 1)
fragmentation_demo >
```

9.9.2.3.2.3 Running the application

The user should send IPv4 or IPv6 frames generated by an external traffic generator. The following example of IPv4 frames was used to validate the IPv4 Fragmentation:

Table 238. fragmentation_demo IPv4 Input Traffic

MAC SA	VLAN TCI	IP Source	Protocol	Size	IP Fragment Offset
00:00:00:00:00:29	0x2614	192.168.1.1	UDP	1280	0
00:00:00:00:00:2a	0x2615	192.168.1.2	UDP	1280	0
00:00:00:00:00:2b	0x2616	192.168.1.3	UDP	1280	0
00:00:00:00:00:2c	0x2617	192.168.1.4	UDP	1280	0

IPv4 frames that have the VLAN TCI value equal to 0x2614 or 0x2616 are fragmented into fragments of maximum 256 bytes in size, while frames that have the VLAN TCI value equal to 0x2615 or 0x2617 are fragmented into fragments of maximum 512 bytes in size.

IPv6 frames that have the VLAN TCI value equal to 0x6800 or 0x6802 are fragmented into fragments of maximum 256 bytes in size, while frames that have the VLAN TCI value equal to 0x6801 or 0x6803 are fragmented into fragments of maximum 512 bytes in size.

The following example of IPv6 frames was used to validate the IPv6 fragmentation:

Table 239. fragmentation_demo IPv6 input traffic

MAC SA	VLAN TCI	IP Source	Protocol	Size
00:00:00:00:00:2d	0x6800	3FFE:1944:0400:000A:0000:00BC:2500:0D01	UDP	1024
00:00:00:00:00:2e	0x6801	3FFE:1944:0400:000A:0000:00BC:2500:0D02	UDP	1024
00:00:00:00:00:2f	0x6802	3FFE:1944:0400:000A:0000:00BC:2500:0D03	UDP	1024
00:00:00:00:00:30	0x6803	3FFE:1944:0400:000A:0000:00BC:2500:0D04	UDP	1024

After sending the traffic, the user can see the statistics values for the IP fragmentation process. The counter “OH_FRAG1” stands for a single counter of type Fragmentation with MTU of 256 bytes, the counter “OH_FRAG2” stands for single counter of type Fragmentation with MTU of 512 bytes and both objects are also grouped in a class counter of type Fragmentation.

```
STATISTICS:       FRAG_TOTAL_FRAMES FRAG_FRAMES FRAG_GEN_FRAGS
OH_FRAG1        :                 0           0             0
OH_FRAG2        :                 0           0             0
CLS_MBR_OH_FRAG1:                 0           0             0
CLS_MBR_OH_FRAG1:                 0           0             0
```

In order to shut down the application you can type *quit* in the application console.

9.9.2.4 Manipulating the reassembly_demo application

This application demonstrates how IP reassembly can be offloaded to the FMan by using the direct XML configuration and the DPA offloading driver.

9.9.2.4.1 Overview

This application demonstrates how IP reassembly can be offloaded to the FMan by using the direct XML configuration and the DPA offloading driver. The IP Reassembly feature can be configured on an RX port or an Offline port and performs reassembly on detected IPv4 and IPv6 protocol fragments. For B4 platforms, the application also provides Virtual Storage Profile support. The DPA Stats component usage is highlighted in the same application by creating single counters for Ethernet and Reassembly, and both single and class counters for Classification Nodes.

The application initializes the Reassembly functionality and the Parse-Classify-Police-Distribute (PCD) description through the use of the FMC Library.

Traffic is received on a backhaul Rx port (BP) by USDPAA reassembly_demo application. IP fragments are detected and reassembled before arriving into the application. The application swaps Ethernet MAC addresses and enqueues the traffic back to the Tx queue of the same port.

The user has the possibility of performing a number of actions on the DPA Stats counters by typing the following commands in the USDPAA PPAM command line:

Table 240. DPA Stats class counter actions

Command Line	Description
get_stats	retrieve the statistics for the created counters in an asynchronous mode and print the returned values.
get_stats_sync	retrieve the statistics for the created counters in a synchronous mode and print the returned values.
reset_stats	reset the statistics for the created counters.

To have a successful operation, the user must verify that the returned values of the counters statistics matches the number of sent frames for each stage of the test.

9.9.2.4.1.1 Virtual storage profiles

The virtual storage profile is only available on frame manager v3. The configuration is transparent to the user. Three storage profiles are defined in the policy file `reassembly_demo_policy-v3.xml`:

```
<vsp name="Default_VSP" base="0"/>
<vsp name="IPv4_Reass_VSP" base="1"/>
<vsp name="IPv6_Reass_VSP" base="2"/>
```

The first storage profile, Default_VSP is also the default storage profile of the port. The second and the third storage profiles are used for IPv4 and IPv6 flows. Each of the storage profiles described above has a set buffer pool associated with it (up to four buffer pools can be defined per vsp)

The buffer pools are dynamically allocated and initialized by reassembly_demo.

When an IPv4 flow is reassembled / classified on inbound port, the IPv4_Reass_VSP is used and the buffers are selected from one of the buffer pools available for this VSP. When an IPv6 flow is reassembled/classified on inbound port, the

IPv6_Reass_VSP with buffers from one of the buffer pools available for this VSP is used. For any other flows the Default_VSP is selected.

The VSP selection mechanism described in this paragraph is totally transparent to the user and doesn't affect the use case behavior in any way.

In order to test the non-consistent storage profile functionality the user needs to take into consideration that if a fragment that enters the reassembly distribution is not the first fragment, de default VSP of the port is selected. Otherwise according to the classification, if the fist fragment has the L4 destination port equal to a specific value, the IPv4 VSP is selected. If the reassembled frame has fragments in buffers from different buffer pools, then there is a NCSP event triggered and the frame is enqueued to the NCSP queue. For testing this particular case the user has to take care sending the first fragment of the IPv4 frame (the one with the Fragment Offset flag with value 0) after any other fragment (Ex: fragment3, fragment2, fragment1, fragment4). For the moment the NCSP could be tested only with IPv4 frames.

9.9.2.4.1.2 Differences from Linux IP stack

Reassembly_demo application does not use the cores or the IP stack for performing reassembly operation. It uses the frame manager for processing the reassembly.

9.9.2.4.2 Running reasembly_demo

reasembly_demo application supports the following platforms:

- P2041
- P4080
- B4860
- B4420
- T4240
- T2080
- LS1043A

In order to run it you may need to use a specific device tree file that comes with the DPA offloading driver. Please read below about how to produce this device tree file for your platform.

9.9.2.4.2.1 Application environment specifications

The environment is the same as described for the classifier_demo application. Please see **Classifier_demo: Application environment specifications.**

If you need the *reassembly_demo* application to run on a different traffic port than the defaults, you will need to

1. adapt the XML port configuration file for your platform
2. adapt the DTS file that Linux kernel boots up with and
3. use the proper command line arguments for your new traffic port.

Please refer to the paragraph **Application start-up and configuration** for further details.

9.9.2.4.2.2 Application start-up and configuration

Use the steps described later in **Appendix A: Compiling the Device Tree for IP Reassembly** to generate a device tree binary file. Boot the board with the compiled kernel, `arch/powerpc/boot/uImage`, and the `DTB` file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

The following commands assume that the application runs on a B4860QDS board. When running on a different supported platform, please use the proper parameters listed in **Running Classifier_demo: Application environment**

specifications. For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH=/usr/etc/reassembly_demo_config-b4860.xml
export DEF_PCD_PATH=/usr/etc/reassembly_demo_policy.xml
export DEF_PDL_PATH=/etc/fmc/config/hxs_pdl_v3.xml
```

First ensure that the following device files have been created:

- `/dev/dpa_stats`

The `reassembly_demo` can be run with the following command:

`/usr/bin/reassembly_demo –f 0 –t 6`

The following arguments specify the FMan ports used by the application:

- `-f [FMan index]`
- `-t [Traffic Port index]`

The output displayed by the application would be something similar with the following:

```
Loading configuration
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/dpa-fman0-oh@3, Tx Channel = 80b, FMAN = 0, Port ID = 2
Found /fsl,dpaa/dpa-fman0-oh@4, Tx Channel = 80c, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@0, Tx Channel = 802, FMAN = 0, Port ID = 0
Found /fsl,dpaa/ethernet@1, Tx Channel = 803, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@2, Tx Channel = 804, FMAN = 0, Port ID = 2
Found /fsl,dpaa/ethernet@5, Tx Channel = 807, FMAN = 0, Port ID = 5
Configuring for 1 network interface
Allocated DMA region size 0x1000000
Released 4096 bufs to BPID 1
Released 4096 bufs to BPID 2
Released 8192 bufs to BPID 3
reassembly_demo is assuming FMan:0 and MAC:6
Released 0 bufs to BPID 7
Released 0 bufs to BPID 8
Released 8192 bufs to BPID 9
Thread uid:0 alive (on cpu 1)
reassembly_demo >
```

9.9.2.4.2.3 Running the application

The user should send IPv4 or IPv6 frames generated by an external traffic generator. The following example of IPv4 and IPv6 frames can be used to validate the IP reassembly:

Table 241. reassembly_demo IPv4 & IPv6 Input Traffic

VLAN TCI	IP Source	Protocol	Size (w/ CRC)	ID field	Flags	Fragment Offset
0x0001	192.168.0.1	UDP	298	5	MF	0
0x0001	192.168.0.1	IPv4	298	5	MF	256 (32 x 8)

Table continues on the next page...

Table 241. reassembly_demo IPv4 & IPv6 Input Traffic (continued)

VLAN TCI	IP Source	Protocol	Size (w/ CRC)	ID field	Flags	Fragment Offset
0x0001	192.168.0.1	IPv4	298	5	MF	512 (64 x 8)
0x0001	192.168.0.1	IPv4	298	5	0	768 (96 x 8)
0x0002	192.168.0.2	UDP	298	5	MF	0
0x0002	192.168.0.2	IPv4	298	5	MF	256 (32 x 8)
0x0002	192.168.0.2	IPv4	298	5	MF	512 (64 x 8)
0x0002	192.168.0.2	IPv4	298	5	0	768 (96 x 8)
0x0003	192.168.0.3	UDP	298	5	MF	0
0x0003	192.168.0.3	IPv4	298	5	MF	256 (32 x 8)
0x0003	192.168.0.3	IPv4	298	5	MF	512 (64 x 8)
0x0003	192.168.0.3	IPv4	298	5	0	768 (96 x 8)
0x0004	192.168.0.4	UDP	298	5	MF	0
0x0004	192.168.0.4	IPv4	298	5	MF	256 (32 x 8)
0x0004	192.168.0.4	IPv4	298	5	MF	512 (64 x 8)
0x0004	192.168.0.4	IPv4	298	5	0	768 (96 x 8)
0x0001	3FFE:1944:0100:000A:0000:00BC:2500:0D0B	UDP	414	5	MF	0
0x0001	3FFE:1944:0100:000A:0000:00BC:2500:0D0B	IPv6	414	5	MF	344
0x0001	3FFE:1944:0100:000A:0000:00BC:2500:0D0B	IPv6	414	5	MF	688
0x0001	3FFE:1944:0100:000A:0000:00BC:2500:0D0B	IPv6	414	5	0	1032
0x0002	3FFE:1944:0200:000A:0000:00BC:2500:0D0B	UDP	414	5	MF	0
0x0002	3FFE:1944:0200:000A:0000:00BC:2500:0D0B	IPv6	414	5	MF	344
0x0002	3FFE:1944:0200:000A:0000:00BC:2500:0D0B	IPv6	414	5	MF	688

Table continues on the next page...

Table 241. reassembly_demo IPv4 & IPv6 Input Traffic (continued)

VLAN TCI	IP Source	Protocol	Size (w/ CRC)	ID field	Flags	Fragment Offset
0x0002	3FFE:1944:0200:000A:0000:00BC:2500:0D0B	IPv6	414	5	0	1032
0x0003	3FFE:1944:0300:000A:0000:00BC:2500:0D0B	UDP	414	5	MF	0
0x0003	3FFE:1944:0300:000A:0000:00BC:2500:0D0B	IPv6	414	5	MF	344
0x0003	3FFE:1944:0300:000A:0000:00BC:2500:0D0B	IPv6	414	5	MF	688
0x0003	3FFE:1944:0300:000A:0000:00BC:2500:0D0B	IPv6	414	5	0	1032
0x0004	3FFE:1944:0400:000A:0000:00BC:2500:0D0B	UDP	414	5	MF	0
0x0004	3FFE:1944:0400:000A:0000:00BC:2500:0D0B	IPv6	414	5	MF	344
0x0004	3FFE:1944:0400:000A:0000:00BC:2500:0D0B	IPv6	414	5	MF	688
0x0004	3FFE:1944:0400:000A:0000:00BC:2500:0D0B	IPv6	414	5	0	1032

The IP Reassembly feature is considered to be working properly if for the provided example of IP fragments are reassembled in the following frames:

Table 242. Expected Reassembled Frames

IP Source	Protocol	Size	Flags	Fragment Offset
192.168.0.1	IPv4/UDP	1066	0	0
192.168.0.2	IPv4/UDP	1066	0	0
192.168.0.3	IPv4/UDP	1066	0	0
192.168.0.4	IPv4/UDP	1066	0	0
3FFE:1944:0100:000A:0000:00BC:2500:0D0B	IPv6/UDP	1438	0	0

Table continues on the next page...

Table 242. Expected Reassembled Frames (continued)

IP Source	Protocol	Size	Flags	Fragment Offset
3FFE:1944:0100:000A:0000:00BC:2500:0D0B	IPv6/UDP	1438	0	0
3FFE:1944:0100:000A:0000:00BC:2500:0D0Bf	IPv6/UDP	1438	0	0
3FFE:1944:0100:000A:0000:00BC:2500:0D0B	IPv6/UDP	1438	0	0

After sending the traffic, the user can validate the statistics values for the IP reassembly process on the Ethernet interface used to send and receive the traffic and the Classification Node:

```
reassembly_demo > get_stats_sync

REASS      : TIMEOUT RFD_POOL_BUSY INT_BUFF_BUSY EXT_BUFF_BUSY SG_FRAGS DMA_SEM NCSP
                0        0            0            0           0       0     0
REASS_IPV4 : FRAMES FRAGS_VALID FRAGS_TOTAL FRAGS_MALFORMED FRAGS_DISCARDED AUTOLEARN_BUSY
EXCEED_16FRAGS
                0        0           0          0               0               0               0
REASS_IPV6 : FRAMES FRAGS_VALID FRAGS_TOTAL FRAGS_MALFORMED FRAGS_DISCARDED AUTOLEARN_BUSY
EXCEED_16FRAGS
                0        0           0          0               0               0               0
ETH        : DROP_PKTS  BYTES  PKTS BC_PKTS MC_PKTS CRC_ALIGN_ERR UNDERSIZE_PKTS OVERSIZE_PKTS
                0          0    0     0      0          0              0            0
ETH        : FRAGMENTS JABBERS 64BYTE_PKTS 65_127BYTE_PKTS 128_255BYTE_PKTS 256_511BYTE_PKTS
512_1023BYTE_PKTS 1024_1518BYTE_PKTS
                0        0          0            0               0               0
0                0
ETH        : OUT_PKTS OUT_DROP_PKTS OUT_BYTES IN_ERRORS OUT_ERRORS IN_UNICAST_PKTS OUT_UNICAST_PKTS
                0          0           0          0           0           0            0
CNT_CLASSIF: IPv6_KEY0 IPv6_KEY1 IPv6_KEY2 IPv6_KEY3 MISS
                0          0         0         0       0
CLS_CLASSIF: IPv6_KEY0 IPv6_KEY1 IPv6_KEY2 IPv6_KEY3 MISS
                0          0         0         0       0
CNT_CLASSIF: IPv4_KEY0 IPv4_KEY1 IPv4_KEY2 IPv4_KEY3 MISS
                0          0         0         0       0
CLS_CLASSIF: IPv4_KEY0 IPv4_KEY1 IPv4_KEY2 IPv4_KEY3 MISS
                0          0         0         0       0
```

In order to shut down the application you can type *quit* in the application console.

9.9.2.5 Customizing the ipsec_offload application

USDPAA ipsec_offload is a multi-threaded application which applies a subset of IPsec transformations to the IPv4/IPv6 traffic.

9.9.2.5.1 IPSec_offload application overview

The IPv4/IPv6 traffic is processed between two Ethernet ports. The inbound port expects encrypted/tunneled traffic while the outbound port expects clear-text traffic. The classification, encryption/decryption, authentication, encapsulation/decapsulation is accomplished using the DPA offload driver which connects and configures DPAA traffic processing accelerators (Fman, SEC) to completely offload IPSec processing based on common Linux configuration tools (setkey, ip xfrm, arp, ip neigh). The following features are currently supported:

- ESP tunnel mode.
- TOS/ECN/DSCP propagation.



Figure 241. IPSec_offload flows

9.9.2.5.1.1 IPSec_offload outbound flows

The clear text traffic is received on the **outbound Rx** and classified based on UDP destination port. Frames matching UDP destination port 2152 are received by USDPAA frame processing threads.The rest of the frames are sent directly to **pre-encryption OH** port. On this port the traffic is classified according to configured IPSec traffic selectors. Frames for which IPsec policy is applicable are sent to SEC for security processing. Optionally, this port performs IP fragmentation for frames matching IPsec policy if frame size is larger than a configured value (`mtu_pre_enc`). Traffic not matching IPsec policy bypass security processing and are enqueued directly to post-encryption OH port. The SEC output frames are enqueued to **post-**

encryption OH port. On this port the Ethernet source and destination addresses are updated with Tx port MAC address and next hop MAC address respectively.



Figure 242. Outbound processing

Below are some flow diagrams that explain (as per the application's PCD) how frames are being processed at each node.



Figure 243. Outbound Rx



Figure 244. Outbound Pre SEC OH



Figure 245. Outbound Post SEC OH

9.9.2.5.1.2 IPSec_offload inbound flows

IPsec traffic (IPv4/IPv6 ESP and UDP-encap ESP) received on the inbound port is classified according to offloaded security associations. Frames for which a security association is found are sent to the appropriate SEC engine input queues for

decryption/decapsulation; traffic for which a security association is not found is dropped. Non-IPsec traffic is directly sent to **inbound** OH port. Decrypted and non-IPsec traffic is reassembled if fragmented, optionally passes inbound policy verification and further a post IPsec classification which sends frames matching UDP destination port 2152 to USDPAA frame processing threads. Frames not matching UDP destination port 2152 have their Ethernet source and destination MAC addresses updated with Tx port MAC address and next hop MAC address respectively.



Figure 246. Inbound processing

Below are some flow diagrams that explain (as per the application's PCD) how frames are being processed at each node.



Figure 247. Inbound Rx



Figure 248. Inbound OH

9.9.2.5.1.3 Differences from existing ipsecfwd application

This USDPAA application allows complete offload of IPSec data paths, thus relieving the cores from the tasks of submitting/ receiving frames to/from the SEC engine prior or after tunnel payload processing nd from frame forwarding after IPsec processing. Core intervention is performed only for tasks related to application specific processing. Also, the application supports standard Linux tools for configuration.

9.9.2.5.2 Running the Application

The procedures for configuring the environment, starting up and configuring the Ipsec_offload application are discussed.

9.9.2.5.2.1 Application environment specifications

The application can run in a real network setup configuration and inter-operate with other IPSec gateways. There is also an option to put the inbound port in loopback mode and return IPSec traffic to the inbound port. The details on how to run the application refers to this mode.



Figure 249. Use case set-up in loopback mode

Table 243. Port connections

SoC	Inbound Port	Outbound Port
P4080	`fm1-mac1`	`fm1-mac0`
P2041	`fm0-mac2`	`fm0-mac1`
B4860	`fm0-mac4`	`fm0-mac5`
B4420	`fm0-mac2`	`fm0-mac3`
T4240	`fm0-mac1`	`fm0-mac0`
T2080	`fm0-mac2`	`fm0-mac3`

If you need the *ipsec_offload* application to run on different traffic ports than the ones listed above, you will need to

1. adapt the XML port configuration file for your platform and
2. adapt the DTS file that Linux kernel boots up with.

Please refer to the paragraph **Application start-up and configuration** for further details.

9.9.2.5.2.2 Application Start-up and Configuration

In this section you will find out how to run the ipsec_offload application. All the examples below are given for a B4860QDS board and assume the standard SDK environment is being used (with the standard RCW, u-boot, *-usdpaa-shared-interfaces.dts).

Note: When running on a different platform, please modify the parameters accordingly as shown in the tables below. **Custom scenarios that alter the default configurations involve changes that are out of the scope of this document.**

Setup the environment

Use the steps described in Appendix A - **Compiling the device tree for IPSec offload** to generate a device tree binary file. Boot the board with the compiled kernel (`arch/powerpc/boot/uImage`) and the DTB file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following boot arguments in uBoot:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

Setup USDPAA configurations and PCD resources

The configuration xml and the PCD xml vary according to platform. Choose the appropriate files for your platform as per the table below. For running the application in single mode or with multiple instances consult the corresponding sections of the user guide.

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_b4860.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_b4.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
```

Table 244. XML config files

SOC	DEF_CFG_PATH	DEF_PCD_PATH
P4080	/usr/etc/ ipsec_offload_config_p4080.xml	/usr/etc/ipsec_offload_pcd_p4.xml
P2041	/usr/etc/ ipsec_offload_config_p2041.xml	/usr/etc/ipsec_offload_pcd_p2.xml
B4860	/usr/etc/ ipsec_offload_config_b4860.xml	/usr/etc/ipsec_offload_pcd_b4.xml
B4420	/usr/etc/ ipsec_offload_config_b4420.xml	/usr/etc/ipsec_offload_pcd_b4.xml
T4240	/usr/etc/ ipsec_offload_config_t4240.xml	/usr/etc/ipsec_offload_pcd_t4.xml
T2080	/usr/etc/ ipsec_offload_config_t2080.xml	/usr/etc/ipsec_offload_pcd_t2.xml

Start the application

Following is a brief description of ipsec_offload's parameters and an explanantion on how to set them:

-c: the configuration file for the platform (platform specific - see table above)

-p: policy file

--vif: virtual inbound interface index

--vof: virtual outbound interface index

--vipsec: virtual interface for host to host tunnels

These interfaces are used by Linux to receive and send local traffic - IP traffic matching Linux IP addresses and non-IP traffic like ARP.

The VIF and VOF parameters are always macless nodes. To choose what interfaces to use, look for interfaces with one of the following ethernet addresses:

```
00:11:22:33:44:55 or 00:11:22:33:44:66 or 00:11:22:33:44:77
```

Any combination of these nodes can be used.

The VIPSEC can be either a macless interface or an oNIC interface (for B4 and T4 platforms only). The ipsec_offload oNIC interface always has 00:11:22:33:44:88 as an ethernet address.

Note: the virtual interfaces parameters have to be provided in the following order: vif, vof, vipsec.

--ib-loop: configures the inbound port in loop mode - all frames received on the Tx port will be fed back to the Rx port

To run the application on B4860QDS, execute the following command:

```
/usr/bin/ipsec_offload -c /usr/etc/ipsec_offload_config_b4860.xml -p /usr/etc/
ipsec_offload_policy.xml --vif macless0 --vof eth1 --vipsec eth2 --ib-loop
```

To shut down the application you can type the command *quit* in the application console. At shutdown the application flushes its offloading tables, but it does not flush the items that were configured in the Linux kernel. Therefore, in case you have performed additional route, neighbor or IPSec configurations using the Linux tools it is more than a good idea to revert/flush them as well using the appropriate commands once the application was shut down.

IMPORTANT NOTE: Multiple SA's can be configured but if the VIF interface is UP and has an IP on it, all the SA's for outbound direction should have that IP as tunnel source, and all the SA's for inbound should have that VIF IP as destination. It is possible to create multiple SA's in which the IP has to be identical but vary the SPI, encryption or authentication keys or algorithm and so on. This constraint comes from the fact that VIF IP has to be the tunnel IP and this information is used to distribute SA's on diffrent IPSec instances.

EXAMPLE:

```
VIF eth3
ifconfig eth3 192.168.60.1 netmask 255.255.255.0 up

add 192.168.60.1 192.168.50.1 esp 0x901 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabplX" -A hmac-
sha1 "abcdefghipqrstuvwxyO";
add 192.168.60.1 192.168.50.2 esp 0x902 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabplY" -A hmac-
sha1 "abcdefghipqrstuvwxyM";
add 192.168.60.1 192.168.50.3 esp 0x903 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabplZ" -A hmac-
sha1 "abcdefghipqrstuvwxyP";

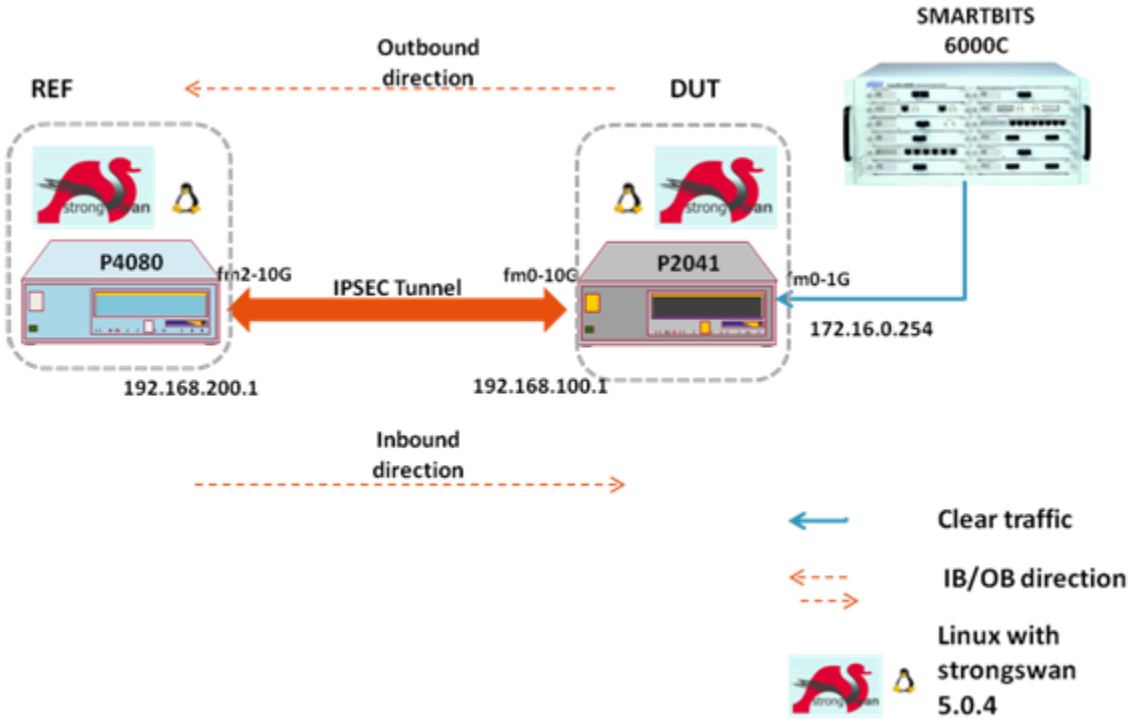
Outbound
spdadd 172.13.0.1/32[any] 172.14.0.1/32[any] udp -P out ipsec esp/tunnel/192.168.60.1-192.168.50.1/
require;
spdadd 172.13.0.2/32[any] 172.14.0.2/32[any] udp -P out ipsec esp/tunnel/192.168.60.1-192.168.50.2/
require;
spdadd 172.13.0.3/32[any] 172.14.0.3/32[any] udp -P out ipsec esp/tunnel/192.168.60.1-192.168.50.3/
require;

Inbound
spdadd 172.14.0.1/32[any] 172.13.0.1/32[any] udp -P in ipsec esp/tunnel/192.168.50.1-192.168.60.1/
require;
spdadd 172.14.0.2/32[any] 172.13.0.2/32[any] udp -P in ipsec esp/tunnel/192.168.50.2-192.168.60.1/
require;
spdadd 172.14.0.3/32[any] 172.13.0.3/32[any] udp -P in ipsec esp/tunnel/192.168.50.3-192.168.60.1/
require;
```

Following script can be used as an example:

```
#!/bin/sh
echo "flush;" | setkey -c
echo "spdflush;" | setkey -c

for i in `seq $1`
do
        spi=`expr 201 + $i`
```

```
        echo "spdadd 172.16.0.$i/32[any] 172.17.0.$i/32[any] udp
                -P out ipsec esp/tunnel/192.168.100.1-192.168.200.$i/require; " | setkey -c

        echo "spdadd 172.17.0.$i/32[any] 172.16.0.$i/32[any] udp
                -P in ipsec esp/tunnel/192.168.200.$i-192.168.100.1/require;" | setkey -c

        echo "add 192.168.100.1 192.168.200.$i esp 0x$spi
                -E 3des-cbc \"abcdefghipqrstuvwxyabcde\"
                -A hmac-sha1 \"abcdefghipqrstuvwxya\";" | setkey -c

        echo "add 192.168.200.$i 192.168.100.1 esp 0x$spi
                -E 3des-cbc \"abcdefghipqrstuvwxyabcde\"
                -A hmac-sha1 \"abcdefghipqrstuvwxya\";" | setkey -c
done
```

Application configuration for IPsec

IPSec can be configured for offloading using setkey tool. To add an SA and two corresponding SPs add the following lines in a setkey.conf file:

```
flush;
spdflush;
add 192.168.100.1 192.168.200.1 esp 0x201
        -E 3des-cbc "abcdefghipqrstuvwxyabcde"
        -A hmac-sha1 "abcdefghipqrstuvwxya";
spdadd 172.16.0.1/32[any] 172.17.0.1/32[any] udp
        -P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[any] 172.16.0.1/32[any] udp
        -P in ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
```

Now run the command:

```
setkey –f setkey.conf
```

These commands create an ESP tunnel with the following endpoints: IP src 192.168.100.1 – IP dst 192.168.200.1, SPI 0x201, 3DES-CBC encryption and HMAC-SHA1 authentication. The UDP traffic coming from 172.16.0.1 going to 172.17.0.1 will be tunneled using the defined tunnel.

To configure Linux interfaces, neighboring and routing for the loopback setup run the following commands:

```
#enable IP forwarding
echo "1" > /proc/sys/net/ipv4/ip_forward
#inbound interface
ifconfig macless0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/macless0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/macless0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/macless0/accept_local
#outbound interface
ifconfig eth1 172.16.0.254 netmask 255.255.0.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth1/accept_local
#route to tunnel destination
route add -net 192.168.200.0/24 gw 192.168.100.254 dev macless0
#route to tunneled traffic destination - outbound interface in loop mode
route add -net 172.17.0.0/16 gw 172.16.0.1 dev eth1
#static neigh entry for loopback mode
```

```
ip neigh add 192.168.100.254 lladdr <inbound_port_mac_addres> dev macless0
```

9.9.2.5.2.3 Running Traffic

Assuming that the traffic is generated with a directly connected Linux box, you need to configure the Linux box to output frames for `172.16.0.1` on the connected interface (ethX):

```
ifconfig <ethX> 172.16.0.2 netmask 255.255.0.0 up
```

You can use the hping tool to generate UDP packets and tcpdump to capture traffic on interface ethX. Each sent frame should be returned by the application.

```
hping -2 -s 2152 -p 2152 -c 1 172.17.0.1
```

At any point you can use the following commands to show packet and byte statistics.

- ```
  sa_stats <sa-id>
  ```

  This command shows the statistics for a particular SA.
- ```
  ipsec_stats <ipsec-instance-id>
  ```

  This command shows the global IPSec statistics for the specified IPSec instance. This includes the miss counters for the inbound and outbound direction.

9.9.2.5.3 ipsec_offload application flow

The application follows PPAC/PPAM design. PPAM is responsible for initialization and for forwarding received frames (UDP destination port 2152) between outbound port and outbound/inbound offline ports. The following tasks are part of the initialization:

- Get command line configuration parameters
- Allocate memory and create buffer pools for additional required buffer pools (i.e – IP fragmentation, IP reassembly)
- Compile and apply PCD model for the application
- Init IPsec offloading component
- Start XFRM and neighbouring notification processing
- Mappings required by USDPAA to forward frames between ports

Offloading work is performed by XFRM and neighbour notifications processing threads. Each thread listens on a specific Netlink socket and receives messages containing the event and associated information. This information is used to configure IPsec offloading datapath.

9.9.2.5.4 IPSEC tunnel setup using ipsec offload application and Strongswan with IKEv2

The application can work in conjunction with stongswan client using IKEv2 protocol. The figure below ilustrates a possible setup that creates a tunnel between DUT and REF devices.



Figure 250. IPSEC tunnel setup with Strongswan and IKEv2

REF can be any client machine and DUT can be any board with **DPAA capabilities**. The **smartbit** can also be replaced by another client machine.

The tunnel is negotiated by both machines and the security associations created on DUT, are offloaded. On **outbound direction,** traffic is sent from **SMARTBITS**, encrypted on DUT and then decrypted on the REF.

On **inbound direction** traffic can be sent from REF using for instance *wget* application:

wget http://172.16.0.1

9.9.2.5.4.1 REF (Linux with Strongswan 5.0.4) configuration

- ipsec.conf (configuration file used by strongswan):

```
 conn %default
        ikelifetime=60m
        keylife=20m
        rekeymargin=3m
        keyingtries=1
        authby=secret
        keyexchange=ikev2
        mobike=no

conn ref
        left=192.168.200.1
        leftsubnet=172.17.0.0/16
        right=192.168.100.1
        rightsubnet=172.16.0.0/16
        auto=add
        reauth=no
        esp=3des-sha1
```

- ipsec.secrets(pre shared secret used by strongswan):

```
 # /etc/ipsec.secrets - strongSwan IPsec secrets file

192.168.100.1 192.168.200.1 : PSK 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
```

To start the IKEv2 daemon on REF, the following command will be executed:

ipsec start

Notes:

No tunnels are created yet.

The interfaces and route configuration for REF machine is the following (it can be different according to the needs):

```
fm2-10g   Link encap:Ethernet  HWaddr 00:04:9f:00:03:09
          inet addr:192.168.200.1  Bcast:192.168.200.255  Mask:255.255.255.0
          inet6 addr: fe80::204:9fff:fe00:309/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1496 errors:0 dropped:304 overruns:0 frame:0
          TX packets:785 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:344650 (336.5 KiB)  TX bytes:112250 (109.6 KiB)
          Memory:fe5f0000-fe5f0fff

fm2-10g:0 Link encap:Ethernet  HWaddr 00:04:9f:00:03:09
          inet addr:192.168.100.254  Bcast:192.168.100.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Memory:fe5f0000-fe5f0fff

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.0.0      192.168.100.1   255.255.0.0     UG    0      0        0 fm2-10g
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
192.168.100.0   0.0.0.0         255.255.255.0   U     0      0        0 fm2-10g
192.168.200.0   0.0.0.0         255.255.255.0   U     0      0        0 fm2-10g
```

9.9.2.5.4.2 DUT (Linux with Strongswan 5.0.4) configuration

- ipsec.conf (configuration file used by trongswan):

```
conn %default
        ikelifetime=120m
        keylife=60m
        rekeymargin=3m
        keyingtries=1
        authby=secret
        keyexchange=ikev2
        mobike=no

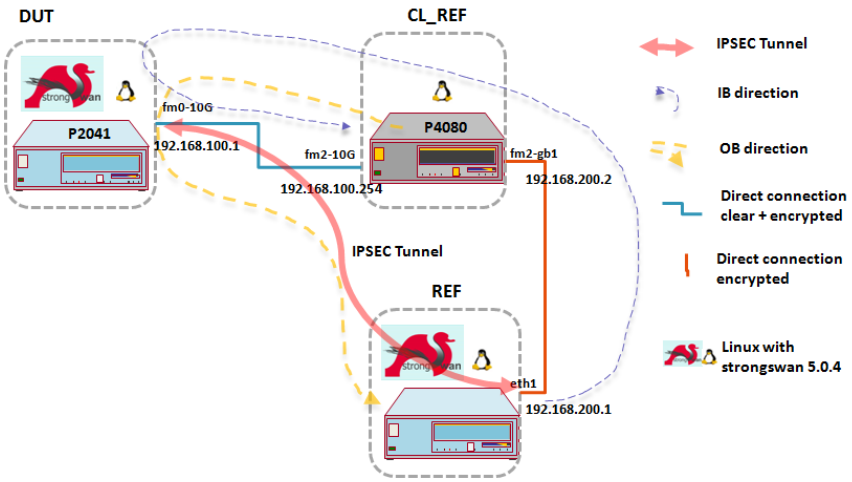

conn dut
 left=192.168.100.1
        leftsubnet=172.16.0.0/16
        reauth=no
        right=192.168.200.1
        rightsubnet=172.17.0.0/16
        auto=start
```

```
        #lifetime=360s
        #rekeyfuzz=0%
        #margintime=330s
        #rekey=yes
        esp=3des-sha1
```

- ipsec.secrets(pre shared secret used by strongswan):

```
 # /etc/ipsec.secrets - strongSwan IPsec secrets file

192.168.100.1 192.168.200.1 : PSK 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
```

To start the IKEv2 daemon on DUT, the following command will be executed:

ipsec start

Notes:

Tunnels will be created -due to *auto=start* option.

The daemon must be started after the ipsec offload application

The interfaces and route configuration for DUT machine is the following (it can be different according to the needs):

```
eth0      Link encap:Ethernet  HWaddr 00:04:9f:02:09:fe
          inet addr:192.168.100.1  Bcast:192.168.100.255  Mask:255.255.255.0
          inet6 addr: fe80::204:9fff:fe02:9fe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:6 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:328 (328.0 B)

eth1      Link encap:Ethernet  HWaddr 00:04:9f:02:09:fa
          inet addr:172.16.0.254  Bcast:172.16.255.255  Mask:255.255.0.0
          inet6 addr: fe80::204:9fff:fe02:9fa/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3 errors:6 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:238 (238.0 B)

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 eth2
172.16.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth1
172.17.0.0      192.168.100.254 255.255.0.0     UG    0      0        0 eth0
192.168.0.0     0.0.0.0         255.255.252.0   U     0      0        0 eth2
192.168.100.0   0.0.0.0         255.255.255.0   U     0      0        0 eth0
192.168.200.0   192.168.100.254 255.255.255.0   UG    0      0        0 eth0
```

9.9.2.5.4.3 Running ipsec offload application on DUT using 10G port setup

In the proposed setup, DUT and REF have a 10G connection. In this situation the ipsec offload application must be started using a **1G port for outbound** and a **10G port for inbound**.

Before starting the application, the xml configuration file that defines the ethernet ports, must be modified - in order to contain the 10G port.

The code snippet from below shows the modification:

```
diff --git a/apps/ipsec_offload/ipsec_offload_config_p2041.xml b/apps/ipsec_offload/
ipsec_offload_config_p2041.xml
index 61bfdac..1d34bb1 100644
--- a/apps/ipsec_offload/ipsec_offload_config_p2041.xml
+++ b/apps/ipsec_offload/ipsec_offload_config_p2041.xml
@@ -38,9 +38,9 @@

<cfgdata>
        <config>
-       <engine name="fm1">
-               <port type="1G" number="2" policy="ib_rx_policy"/>
+       <engine name="fm0">
+               <port type="10G" number="0" policy="ib_rx_policy"/>
                <port type="OFFLINE" number="1" policy="ib_oh_post_policy"/>
                <port type="OFFLINE" number="2" policy="ob_oh_pre_policy"/>
                <port type="OFFLINE" number="3" policy="ob_oh_post_policy"/>
```

Note:

The port numbers may change due to different hardware configuration.

The ipsec offload application will be started using the following commands:

```
 export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p2041.xml"
 export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p2.xml"
 export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
 export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"

/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p2041.xml \
-p /usr/etc/ipsec_offload_policy.xml \
-s /usr/etc/ipsec_offload_swp.xml \
--fm 0 \
--ob_eth 1,1 --ib_eth 0,2  \
--ib-oh 1 --ob-oh-pre 2 --ob-oh-post 3 \
--max-sa 16 --vif eth0 --vof eth1 --vipsec eth2 --mtu-pre-enc 500
```

Notes:

max-sa parameter must reflect the number of SA pairs. Looking in the pcd file, there are defined 16 entries for inbound esp_cc classification table. In this case there will be 16 security associations. Interfaces vif, vof and vipsec may change due to different hardware configuration.

From another console, after ipsec offload startup, the following commands will be executed:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth0/accept_local
ifconfig eth1 172.16.0.254 netmask 255.255.0.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth1/accept_local
```

```
route add -net 192.168.200.0/24 gw 192.168.100.254 dev eth0
route add -net 172.17.0.0/16 gw 192.168.100.254 dev eth0
```

Start strongswan:

ipsec start

Note:

A tunnel will be negotiated between DUT and REF.

9.9.2.5.4.4 Rekeying process with Strongswan and ipsec offload

This chapter shortly presents how to enable the rekey process using strongswan.(For more details please consult *strongswan wiki page*)

When a security association expires (after a period of time or after a number of processed packets/bytes) the operating system will notify strongswan. The expired security associations will be updated(new key material will be generated).

Ipsec offload application is able to catch the notifications and will update the offloaded SAs. The DPAA offload function used for rekeying, guarantees that no packets will be lost during this process.

To enable rekeying on DUT (see **DUT configuration** chapter), uncomment (remove "#") the following lines:

```
        #lifetime=360s
        #rekeyfuzz=0%
        #margintime=330s
        #rekey=yes
```

before starting strongswan and ipsec offload application.

The above lines will cause the created tunnels to expire at every 30 seconds (For more details please consult *strongswan wiki page* for rekeying process).

Notes:.

On inbound direction, traffic must be injected before the 30 seconds interval expires. Otherwise the security association that has expired, will stay in rekey process.

9.9.2.5.4.5 Ipsec offload application in single port configuration

Configuration setup

The figure below represents the ipsec tunnel setup using one port - for inbound and outbound directions.



REF can be any machine and DUT can be any board with **offload capabilities**. CL_REF can be any machine with a 10G port available. The difference between this setup and the previous depicted setup is that the DUT machine has only one port available on the offload path, on which will be applied both policies - for inbound and outbound directions.

On outbound direction traffic is sent from CL_REF with a traffic generator application (e.g wget or hping), encrypted on DUT sent back on CL_REF, forwarded to REF where is decrypted.

On inbound direction traffic is sent from REF with a traffic generator application (e.g wget or hping), encrypted on REF, forwarded by CL_REF , decrypted on DUT and sent back to CL_REF

Note that the tunnel is created between DUT and REF. CL_REF will be used for forwarding.

Strongswan daemon is configured in the same way as in the previous setup - both for DUT and REF.

DUT configuration

The following commands will be run on DUT after application start up but before starting strongswan:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth0/accept_local
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth1/accept_local
route add -net 192.168.200.0/24 gw 192.168.100.254 dev eth0
route add -net 172.16.0.0/16 gw 192.168.100.254 dev eth0
ip neigh add 192.168.100.254 lladdr [CL_REF mac addr] dev eth0
```

CL_REF mac addr represents the ethernet address of the CL_REF interface that is connected with DUT port

Route configuration on DUT:

```
root@p2041rdb:/root# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         192.168.1.1     0.0.0.0         UG    0      0        0 eth2
172.16.0.0      192.168.100.254 255.255.0.0     UG    0      0        0 eth0
192.168.0.0     *               255.255.252.0   U     0      0        0 eth2
192.168.100.0   *               255.255.255.0   U     0      0        0 eth0
192.168.200.0   192.168.100.254 255.255.255.0   UG    0      0        0 eth0
```

REF configuration

```
eth1      Link encap:Ethernet  HWaddr A0:36:9F:19:FE:14
          inet addr:192.168.200.1  Bcast:192.168.200.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:335 errors:0 dropped:0 overruns:0 frame:0
          TX packets:140421 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:39930 (38.9 KiB)  TX bytes:36437402 (34.7 MiB)

eth1:0    Link encap:Ethernet  HWaddr A0:36:9F:19:FE:14
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1


192.168.100.0   192.168.100.254 255.255.255.0   UG    0      0        0 eth1
192.168.100.0   *               255.255.255.0   U     0      0        0 eth1
10.171.81.0     *               255.255.255.0   U     0      0        0 eth0
192.168.200.0   *               255.255.255.0   U     0      0        0 eth1
172.16.0.0      192.168.200.1   255.255.0.0     UG    0      0        0 eth1
default         10.171.81.254   0.0.0.0         UG    0      0        0 eth0
```

CL_REF configuration

```
fm2-10g      Link encap:Ethernet  HWaddr 00:10:18:BA:E4:05
          inet addr:192.168.100.254  Bcast:192.168.100.255  Mask:255.255.255.0
          inet6 addr: fe80::210:18ff:feba:e404/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9000  Metric:1
          RX packets:1100 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7691 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:148952 (145.4 KiB)  TX bytes:3449552 (3.2 MiB)
          Interrupt:59 Memory:d6000000-d6012800

fm2-10g:0    Link encap:Ethernet  HWaddr 00:10:18:BA:E4:05
          inet addr:172.16.0.1  Bcast:172.16.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:9000  Metric:1
          Interrupt:59 Memory:d6000000-d6012800

fm2-gb1      Link encap:Ethernet  HWaddr 00:10:18:BA:E4:04
          inet addr:192.168.200.2  Bcast:192.168.200.255  Mask:255.255.255.0
          inet6 addr: fe80::210:18ff:feba:e404/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:942 errors:0 dropped:0 overruns:0 frame:0
          TX packets:940 errors:0 dropped:0 overruns:0 carrier:0
```

```
          collisions:0 txqueuelen:1000
          RX bytes:1981970 (1.8 MiB)  TX bytes:215370 (210.3 KiB)
          Interrupt:67 Memory:d8000000-d8012800



          Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.100.0   *               255.255.255.0   U     0      0        0 fm2-10g
10.171.81.0     *               255.255.255.0   U     0      0        0 eth0
192.168.200.0   *               255.255.255.0   U     0      0        0 fm2-gb1
172.16.0.0      *               255.255.0.0     U     0      0        0 fm2-10g
172.17.0.0      192.168.100.1   255.255.0.0     UG    0      0        0 fm2-10g
default         10.171.81.254   0.0.0.0         UG    0      0        0 eth0
```

Notes:

Forwarding must be enabled between fm2-10g and fm2-gb1.

Running the ipsec offload applicaion in single port configuration

For this setup, the single port xml files will be used.

The config xml for single port is like the snippet below:

```
<cfgdata>
        <config>
        <engine name="fm0">
-                 <port type="1G" number="2" policy="ib_ob_rx_policy"/>
+                <port type="10G" number="0" policy="ib_ob_rx_policy"/>
                <port type="OFFLINE" number="1" policy="ib_oh_post_policy"/>
                <port type="OFFLINE" number="2" policy="ob_oh_pre_policy"/>
                <port type="OFFLINE" number="3" policy="ob_oh_post_policy"/>
        </engine>
        </config>
</cfgdata>
```

Note:

The port numbers may change due to different hardware configuration.

The ipsec offload application will be started using the following commands:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p2041_1p.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p2_1p.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p2041_1p.xml \
-p /usr/etc/ipsec_offload_policy_1p.xml \
-s /usr/etc/ipsec_offload_swp.xml \
--vif eth0 --vof eth0 --vipsec eth3
```

Notes:

Interfaces vif, vof and vipsec may change due to different hardware configuration.

The number of maximum SA pairs can be determined by looking in the pcd file, there are defined 16 entries for inbound esp_cc classification table. In this case there will be 16 security associations.

From another console, after ipsec offload startup, the following commands will be executed:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth0/accept_local
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth1/accept_local
route add -net 192.168.200.0/24 gw 192.168.100.254 dev eth0
route add -net 172.16.0.0/16 gw 192.168.100.254 dev eth0
ip neigh add 192.168.100.254 lladdr 00:10:18:BA:E4:05 dev eth0
```

Start strongswan:

ipsec start

Note:

A tunnel will be negotiated between DUT and REF.

Sending traffic on outbound direction

wget was used for sending traffic. Any other application that can generate tcp/udp packets, can be used.

```
wget --no-proxy http://172.17.0.1/
```

The tcpdump output on CL_REF will be similar to this:

```
8:12:14.823907 00:10:18:ba:e4:05 (oui Unknown) > 00:04:9f:02:09:fb (oui Unknown), ethertype IPv4
(0x0800), length 74: 172.16.0.1.52699 > 172.17.0.1.http
18:12:14.823988 00:04:9f:02:09:fb (oui Unknown) > 00:10:18:ba:e4:05 (oui Unknown), ethertype IPv4
(0x0800), length 126: 192.168.100.1 > 192.168.200.1: ESP(spi=0xc718bd51,seq=0x13), length 92
```

On REF the tcpdump output will be similar to:

```
18:23:18.829555 00:10:18:ba:e4:04 (oui Unknown) > a0:36:9f:19:fe:14 (oui Unknown), ethertype IPv4
(0x0800), length 126: 192.168.100.1 > 192.168.200.1: ESP(spi=0xc115218e,seq=0xe), length 92
18:23:18.829555 00:10:18:ba:e4:04 (oui Unknown) > a0:36:9f:19:fe:14 (oui Unknown), ethertype IPv4
(0x0800), length 74: 172.16.0.1.51413 > 172.17.0.1.http
```

Sending traffic on inbound direction

wget was used for sending traffic. Any other application that can generate tcp/udp packets, can be used.

```
wget --no-proxy http://172.16.0.1
```

The tcpdump output on REF will be similar to this:

```
Connecting to 172.16.0.1:80... 18:22:34.220792 a0:36:9f:19:fe:14 (oui Unknown) > 00:10:18:ba:e4:04
(oui Unknown), ethertype IPv4 (0x0800), length 126: 192.168.200.1 > 192.168.100.1:
ESP(spi=0xc4163da8,seq=0x2), length 92
```

On CL_REF the tcpdump output will be similar to:

```
18:09:07.192481 00:10:18:ba:e4:05 (oui Unknown) > 00:04:9f:02:09:fb (oui Unknown), ethertype IPv4
(0x0800), length 126: 192.168.200.1 > 192.168.100.1: ESP(spi=0xc0ce4d0a,seq=0x12), length 92
18:09:07.192625 00:04:9f:02:09:fb (oui Unknown) > 00:10:18:ba:e4:05 (oui Unknown), ethertype IPv4
(0x0800), length 74: 172.17.0.2.39485 > 172.16.0.1.http
```

Single port with aggregation option

In single port configuration, ipsec offload application supports an operating mode called aggregation.

Let's suppose there are three SAs on inbound direction: SAi1, SAi2 and SAi3 and one SA on outbound: SAo1. Encrypted traffic received on inbound, would match one of the previous SAs- SAi1 to SAi3. Consider that the decrypted traffic have the following selectors:

```
for SAi1:: 172.16.0.0/16 172.17.0.0/16 udp
for SAi2: 172.18.0.0/16 172.19.0.0/16 udp
for SAi3: 172.20.0.0/16 172.21.0.0/16 udp
```

Consider that on outbound direction the policies corresponding to SAo1 will have the above selectors.

If aggregation is enabled, decrypted traffic from SAi1 to SAi3 will be forwarded to the outbound offline port pre-encryption. The forwarded traffic will match the outbound policies , will be encrypted with the SAo1 and sent on the Tx port.

For **aggregation mode**, the ipsec offload application will be started using the following commands:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p2041_1p.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p2_1p.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p2041_1p.xml \
-p /usr/etc/ipsec_offload_policy_1p.xml \
-s /usr/etc/ipsec_offload_swp.xml \
--vif eth0 --vof eth0 --vipsec eth3 --aggreg
```

Note:

Aggregation is available only in single port configuration. In this situation, the decrypted traffic must match the configured policies on outbound direction.(see aforementioned explanation)

Sample setkey configuration file:

```
#!/usr/sbin/setkey -f

# Flush the SAD and SPD
flush;
spdflush;

add 192.168.200.1 192.168.100.1 esp  0x201 -m tunnel
-E 3des-cbc 0xbaba42c2c0d033052267d10a2683325c56fa5d35c0b202d8
-A hmac-sha1 0xdadae1554cc000111b208fad023a8feffcb4e421;

add 192.168.200.1 192.168.100.2 esp  0x202 -m tunnel
-E 3des-cbc 0xbaba42c2c0d033052267d10a2683325c56fa5d35c0b202d8
-A hmac-sha1 0xdadae1554cc000111b208fad023a8feffcb4e421;

add 192.168.200.1 192.168.100.3 esp  0x203 -m tunnel
-E 3des-cbc 0xbaba42c2c0d033052267d10a2683325c56fa5d35c0b202d8
```

```
-A hmac-sha1 0xdadae1554cc000111b208fad023a8feffcb4e421;

add 192.168.100.1 192.168.200.1 esp  0x204 -m tunnel
-E 3des-cbc 0x3a3842c2c0d033052267d10a2683325c56fa5d35c0b202d8
-A hmac-sha1 0xf1a9e1554cc000111b208fad023a8feffcb4e421;

spdadd 172.16.0.0/16[any] 172.17.0.0/16[any] udp
-P in ipsec esp/tunnel/192.168.200.1-192.168.100.1/require;

spdadd 172.16.0.0/16[any] 172.17.0.0/16[any] udp
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;


spdadd 172.18.0.0/16[any] 172.19.0.0/16[any] udp
-P in ipsec esp/tunnel/192.168.200.1-192.168.100.2/require;

spdadd 172.18.0.0/16[any] 172.19.0.0/16[any] udp
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;

spdadd 172.20.0.0/16[any] 172.21.0.0/16[any] udp
-P in ipsec esp/tunnel/192.168.200.1-192.168.100.3/require;

spdadd 172.20.0.0/16[any] 172.21.0.0/16[any] any
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
```

9.9.2.5.5 Ipsec offload application in multiple instances configuration

Steps to initialize first instance - for FMAN 1

The following commands assume that the application runs on a P4080 board. Altering the presented port configurations involves various changes, including device tree updates, and that is not covered by this document.

For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p4080_1p_fman1_instance.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p4_1p.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
```

Start the application with the following command:

```
/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p4080_1p_fman1_instance.xml \
-p /usr/etc/ipsec_offload_policy_1p_fman1_instance.xml \
-s /usr/etc/ipsec_offload_swp.xml \
--vif eth4 --vof eth4 --vipsec macless0
```

Configure interfaces:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth4 192.168.100.1 netmask 255.255.255.0 up
```

```
echo "1" > /proc/sys/net/ipv4/conf/eth4/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth4/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth4/accept_local
route add -net 192.168.200.0/24 gw 192.168.100.254 dev eth4
route add -net 172.17.0.0/16 gw 192.168.100.254 dev eth4
ip neigh add 192.168.100.254 lladdr  00:e0:0c:00:93:05 dev eth4
arp -s 172.16.0.1 00:11:22:22:11:00 dev eth4
```

Create tunnels and policies for the first instance:

```
add 192.168.100.1 192.168.200.1 esp 0x201 -m tunnel -E 3des-cbc "kabcdefghipqrstuvwxyabcde" -A hmac-
sha1 "abcdefghipqrstuvwxya";
add 192.168.200.1 192.168.100.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabcde" -A hmac-
sha1 "abcdefghipqrstuvwxya";
spdadd 172.16.0.1/32[any] 172.17.0.1/32[any] udp -P out ipsec esp/tunnel/
192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[any] 172.16.0.1/32[any] udp -P in ipsec esp/tunnel/
192.168.200.1-192.168.100.1/require;
spdadd 172.16.0.1/32[any] 172.17.0.1/32[any] icmp -P out ipsec esp/tunnel/
192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[any] 172.16.0.1/32[any] icmp -P in ipsec esp/tunnel/
192.168.200.1-192.168.100.1/require;
```

Validate ESP operation

```
Send traffic that matches the outbound policies and see that ESP packets are
created. Verify also that statistics on the outbound SA increment with the
number of ESP created packets.

Send ESP packets to the board and see that they are decapsulated correctly
and check also the statistics.
```

Stept to initialize the second instance - for FMAN 0

The following commands assume that the application runs on a P4080 board. Altering the presented port configurations involves various changes, including device tree updates, and that is not covered by this document.

For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p4080_1p_fman0_instance.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p4_1p.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
```

Start the application with the following command:

```
/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p4080_1p_fman0_instance.xml \
-p /usr/etc/ipsec_offload_policy_1p_fman0_instance.xml \
-s /usr/etc/ipsec_offload_swp.xml \
```

```
--vif eth3 --vof eth3 --vipsec eth5
```

Configure interfaces:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth3 192.168.60.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth3/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth3/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth3/accept_local
route add -net 192.168.50.0/24 gw 192.168.60.254 dev eth3
route add -net 172.13.0.0/16 gw 192.168.60.254 dev eth3
ip neigh add 192.168.60.254 lladdr  00:e0:0c:00:93:05 dev eth3
arp -s 172.14.0.1 00:11:22:22:11:00 dev eth3
```

Create tunnels and policies for the second instance:

```
add 192.168.60.1 192.168.50.1 esp 0x901 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabplX" -A hmac-
sha1 "abcdefghipqrstuvwxyO";
add 192.168.50.1 192.168.60.1 esp 0x901 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabplm" -A hmac-
sha1 "abcdefghipqrstuvwxya";
spdadd 172.13.0.1/32[any] 172.14.0.1/32[any] udp -P out ipsec esp/tunnel/192.168.60.1-192.168.50.1/
require;
spdadd 172.14.0.1/32[any] 172.13.0.1/32[any] udp -P in ipsec esp/tunnel/192.168.50.1-192.168.60.1/
require;
spdadd 172.13.0.1/32[any] 172.14.0.1/32[any] icmp -P out ipsec esp/tunnel/
192.168.60.1-192.168.50.1/require;
spdadd 172.14.0.1/32[any] 172.13.0.1/32[any] icmp -P in ipsec esp/tunnel/192.168.50.1-192.168.60.1/
require;
```

Validate ESP operation

Send traffic that matches the outbound policies and see that ESP packets are created. Verify also that statistics on the outbound SA increment with the number of ESP created packets. Send ESP packets to the board and see that they are decapsulated correctly and check also the statistics.

IMPORTANT NOTE: Multiple SA's can be configured but if the VIF interface is UP and has an IP on it, all the SA's for outbound direction should have that IP as tunnel source, and all the SA's for inbound should have that VIF IP as destination. It is possible to create multiple SA's in which the IP has to be identical but vary the SPI, encryption or authentication keys or algorithm and so on. This constraint comes from the fact that VIF IP has to be the tunnel IP and this information is used to distribute SA's on diffrent IPSec instances.

9.9.2.5.6 Host to host tunnels

Overview

The ipsec_offload usecase can be used to protect data flows between a pair of hosts. An IPsec tunnel can be setup to provide end to end encryption / decryption between the two hosts. The interface that the ipsec_offload uses for host to host tunnels is the one provided via the ‘vipsec’ command line argument.

Configuration and start-up

This section details the configuration sequence needed to enable host to host tunnels on a P4080DS board

- Run the usecase:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p4080.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p4.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
/usr/bin/ipsec_offload \
--vif macless0 --vof eth3 --vipsec eth4
```

- Configure the ethernet interfaces:

```
echo "1" > /proc/sys/net/ipv4/ip_forward

ifconfig macless0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/macless0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/macless0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/macless0/accept_local

route add -net 192.168.200.0/24 gw 192.168.100.254 dev macless0
ip neigh add 192.168.100.254 lladdr 00:04:9f:02:87:26 dev macless0

ifconfig eth4  172.20.0.1 up
echo "1" > /proc/sys/net/ipv4/conf/eth4/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth4/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth4/accept_local

echo "1" > /proc/sys/net/ipv4/conf/macless0/arp_accept
echo "1" > /proc/sys/net/ipv4/conf/macless0/proxy_arp
```

- Create IPsec tunnels using the following setkey script:

```
flush;
spdflush;
add 192.168.100.1 192.168.200.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabcde" -A
hmac-sha1 "abcdefghipqrstuvwxya";
add 192.168.200.1 192.168.100.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabcde" -A
hmac-sha1 "abcdefghipqrstuvwxya";
spdadd 172.20.0.1/32[any] 172.21.0.1/32[any] icmp -P out ipsec esp/tunnel/
192.168.100.1-192.168.200.1/require;
spdadd 172.21.0.1/32[any] 172.20.0.1/32[any] icmp -P in ipsec esp/tunnel/
192.168.200.1-192.168.100.1/require;
spdadd 172.20.0.1/32[any] 172.21.0.1/32[any] udp -P out ipsec esp/tunnel/
192.168.100.1-192.168.200.1/require;
spdadd 172.21.0.1/32[any] 172.20.0.1/32[any] udp -P in ipsec esp/tunnel/
192.168.200.1-192.168.100.1/require;
```

UDP and ICMP traffic going out of eth4 will be encrypted and sent via the 192.168.100.1-192.168.200.1 tunnel. UDP and ICMP traffic coming out of the 192.168.200.1-192.168.100.1 tunnel will be decrypted and the frames addressed to the vipsec interface will be forwarded accordingly.

Traffic

Connect the other end-point of the tunnel by configuring the following on a test station which is linked to the inbound port of the board.

- Configure the interfaces:

```
ifconfig eth1 172.21.0.1/16 up
ifconfig eth1:0 192.168.200.1/24 up
route add -net 172.20.0.0/16 dev eth1
route add -net 192.168.100.0/24 dev eth1
```

- Configure IPSec tunnels by running the following setkey script:

```
flush;
spdflush;
add 192.168.100.1 192.168.200.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabcde" -A
hmac-sha1 "abcdefghipqrstuvwxya";
add 192.168.200.1 192.168.100.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyabcde" -A
hmac-sha1 "abcdefghipqrstuvwxya";
spdadd 172.21.0.1/32[any] 172.20.0.1/32[any] icmp -P out ipsec esp/tunnel/
192.168.200.1-192.168.100.1/require;
spdadd 172.20.0.1/32[any] 172.21.0.1/32[any] icmp -P in ipsec esp/tunnel/
192.168.100.1-192.168.200.1/require;
spdadd 172.21.0.1/32[any] 172.20.0.1/32[any] udp -P out ipsec esp/tunnel/
192.168.200.1-192.168.100.1/require;
spdadd 172.20.0.1/32[any] 172.21.0.1/32[any] udp -P in ipsec esp/tunnel/
192.168.100.1-192.168.200.1/require;
```

Outbound traffic

On the P4080 board, configure a route to the test station – all traffic destined to that network will go through the vipsec interface:

```
route add -net  172.21.0.0/16 dev eth4
```

Send an ICMP request from the board to the test station:

```
ping 172.21.0.1 –c 1
```

Check the Tx statistics on the inbound port (eth_stats 1 1) and the outbound SA stats (sa_stats 0). They will each be incremented by one.

On the test station run a tcpdump on the eth1 interface:

```
15:33:16.196378 IP 192.168.100.1 > 192.168.200.1: ESP(spi=0x00000201,seq=0x6), length 116
15:33:16.196378 IP 172.20.0.1 > 172.21.0.1: ICMP echo request, id 1864, seq 0, length 64
15:33:16.196448 IP 192.168.200.1 > 192.168.100.1: ESP(spi=0x00000201,seq=0x14), length 116
```

The ICMP request is seen as both encrypted and then decrypted. Also the encrypted ICMP reply can be seen.

Inbound traffic

Send a ICMP request from the test station to the board:

```
ping 172.20.0.1 -c 1
```

Check the inbound SA stats (sa_stats 1). They will be incremented by one.

On the P4080 run tcpdump on the vipsec interface:

```
17:11:05.282030 IP 172.21.0.1 > 172.20.0.1: ICMP echo request, id 11607, seq 1, length 64
17:11:05.282050 IP 172.20.0.1 > 172.21.0.1: ICMP echo reply, id 11607, seq 1, length 64
```

Only clear traffic will be seen on this interface.

Host to host tunnels via oNIC

oNIC is a new Ethernet driver which is an improved solution (with CSUM offload and zero-copy) for macless. Currently it is available only on the T4 and B4 platforms.

An oNIC device has been integrated in the ipsec_offload usecase – the vipsec device can now be oNIC instead of macless. This transition is seemingly transparent - the only thing the user needs to do is to specify an oNIC node instead of a macless node as the vipsec command line parameter.

The oNIC interface can be identified from the DTS (it has the "fsl,dpa-ethernet-generic" attribute). For example for a B4 board:

```
ethernet@19 {
    compatible = "fsl,b4860-dpa-ethernet-generic", "fsl,dpa-ethernet-generic";
    fsl,qman-frame-queues-rx =  <7000 2>;
    fsl,qman-frame-queues-tx = <7008 1>;
    fsl,oh-ports = <&oh2 &oh3>;
    local-mac-address = [00 11 22 33 44 88];
    };
```

When running ipsec_offload pass the corresponding ethernet interface as the vipsec:

```
eth6   Link encap:Ethernet  HWaddr 00:11:22:33:44:88
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

9.9.2.6 Using the dpa_offload - the NF API offloading demo application

This application demonstrates how IPSec and IP forwarding services can be offloaded to the DPAA using the *Network Function Layer* DPAA offloading drivers extension.

9.9.2.6.1 Overview

This application shows how IPSec services, as defined by the *RFC 4301*, and IP forwarding services can be offloaded from the Linux user space by using the USDPAA, the *Network Function Layer* and the DPAA offloading drivers. It equally

demonstrates how the *Network Function Layer* offloading services can be integrated with a standard operating system's IP forwarding and IPSec configuration tools.

This application overrides the Linux operating system's data path configuration and it offloads into DPAA:

- IPSec tunnels that the user configures by using the *setkey* tool
- IP routes that are configured by the use of the *ip route* and *ip neigh* tools
- inbound ingress rules configured by the use of the application command line interface (CLI)

NOTE

This application is a pilot and has the following limitations:

Table 245. DPAA Offloading Application Limitations

Limitation Description
This application does not yet support IPv6 IPSec tunnels & policies offloading.
This application does not yet support IPSec *red side fragmentation* offloading.
This application currently supports offloading only IPSec tunnels with one policy per direction.
This application does not yet support the IPSec DSCP/ECN copy feature.
This application does not yet support IPSec *inbound policy verification*.
This application currently supports only IPSec policies with specific L4 ports. It does not yet support the *ANY* specifier or L4 port ranges in the IPSec policy.
The application does not yet support IPSec *host-to-host tunnels*.
The application does not yet support IPSec and IP forwarding offloading using multiple instances in parallel.

9.9.2.6.2 Running dpa_offload

The dpa_offload demo application supports the following platforms:

- B4860

In order to run it you will need to use a specific device tree file that comes with the DPA offloading driver. Please read below about how to produce this device tree file for your platform.

9.9.2.6.2.1 Application environment specifications

The application can run in a real network setup configuration and inter-operate with other IPSec gateways. There is also an option to put the inbound port in loopback mode and return IPSec traffic to the outbound port. The details on how to run the application refers to this mode.

The simplest hardware setup is the one described in the **ipsec_offload Application environment specifications** section.

Table 246. Port connections

SoC	Inbound Port	Outbound Port
B4860	`fm1-mac5`	`fm1-mac6`

9.9.2.6.2.2 Application start-up and configuration

Use the steps described later in **Appendix A: Compiling the device tree for Network Function Layer offloading** to generate a device tree binary file. Boot the board with the compiled kernel, `arch/powerpc/boot/uImage`, and the `DTB` file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

The following commands assume that the application runs on a B4860QDS board. When running on a different supported platform, please use the proper parameters listed in **Running dpa_offload: Application environment specifications**. For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH="/usr/etc/dpa_offload_config_b4860.xml"
export DEF_POL_PATH="/usr/etc/dpa_offload_policy.xml"
export DEF_PCD_PATH="/usr/etc/dpa_offload_pcd_b4.xml"
export DEF_SWP_PATH="/usr/etc/dpa_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
```

First ensure that the following device files have been created:

- `/dev/dpa_classifier`
- `/dev/dpa_stats`
- `/dev/dpa_ipsec`

Following is a brief description of dpa_offload application's parameters and an explanantion on how to set them:

- `--vif`, virtual inbound interface name;
- `--vof`, virtual outbound interface name;
- `--vipsec`, virtual interface for host-to-host IPSec tunneling; this needs to be provided, but it is not currently used due to an application limitation;
- `--ib-loop`, set the inbound interface in loopback mode;
- `--disable-ib-ecn`, disable inbound DSCP/ECN field copy; you should always use this due to the current limitation of the application;
- `--disable-ob-ecn`, disable outbound DSCP/ECN field copy; you should always use this due to the current limitation of the application.

The *VIF* and *VOF* arguments are the names of the shared interfaces that the application is using as inbound and outbound interfaces. These are the Ethernet interface names mentioned in the **Application environment specifications** section above for the platform that you are running on.

The *VIPSEC* interface is a macless interface. Please select the interface with the following Ethernet address:

```
00:11:22:33:44:55
```

The dpa_offload application can be run with the following command:

```
/usr/bin/dpa_offload --vif fm1-mac5 --vof fm1-mac6 --vipsec macless0 --disable-ib-ecn --disable-ob-ecn
--ib-loop
```

The output displayed by the application would be something similar with the following:

```
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/dpa-fman0-oh@3, Tx Channel = 80b, FMAN = 0, Port ID = 2
Found /fsl,dpaa/dpa-fman0-oh@4, Tx Channel = 80c, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@8, Tx Channel = 800, FMAN = 0, Port ID = 9
Found /fsl,dpaa/ethernet@4, Tx Channel = 806, FMAN = 0, Port ID = 5
Found /fsl,dpaa/ethernet@5, Tx Channel = 807, FMAN = 0, Port ID = 6
Found /fsl,dpaa/ethernet@16, MAC-LESS node
```

```
Found /fsl,dpaa/ethernet@17, MAC-LESS node
Found /fsl,dpaa/ethernet@18, MAC-LESS node
Found /fsl,dpaa/ethernet@19, Tx Channel = 80a, FMAN = 0, Port ID = 0
..........  USDPAA Configuration  ..........

Network interfaces: 5

+ Fman 0, MAC 1 (OFFLINE);
        tx_channel_id: 0x80a
        fqid_rx_hash:
                (PCD: start 0x2e00, count 1)
                (PCD: start 0x10e4, count 1)
        fqid_rx_def: 0x6f
        fqid_rx_err: 0x6e

+ Fman 0, MAC 2 (OFFLINE);
        tx_channel_id: 0x80b
        fqid_rx_hash:
                (PCD: start 0x10e6, count 1)
        fqid_rx_def: 0x69
        fqid_rx_err: 0x68

+ Fman 0, MAC 3 (OFFLINE);
        tx_channel_id: 0x80c
        fqid_rx_hash:
                (PCD: start 0x10e8, count 1)
        fqid_rx_def: 0x71
        fqid_rx_err: 0x70

+ Fman 0, MAC 5 (1G);
        mac_addr: 00:e0:0c:00:4c:04
        tx_channel_id: 0x806
        fqid_rx_hash:
                (PCD: start 0x10e0, count 1)
        fqid_rx_def: 0x59
        fqid_rx_err: 0x58
        fqid_tx_err: 0x78
        fqid_tx_confirm: 0x79
        buffer pool: (bpid=16, count=2048 size=1728, addr=0x0)

+ Fman 0, MAC 6 (1G);
        mac_addr: 00:e0:0c:00:4c:05
        tx_channel_id: 0x807
        fqid_rx_hash:
                (PCD: start 0x3e00, count 1)
                (PCD: start 0x10e2, count 1)
        fqid_rx_def: 0x5b
        fqid_rx_err: 0x5a
        fqid_tx_err: 0x7a
        fqid_tx_confirm: 0x7b
        buffer pool: (bpid=16, count=2048 size=1728, addr=0x0)
Interface fman 0, index 1: enable rx
Interface fman 0, index 2: enable rx
Interface fman 0, index 3: enable rx
Mapping Rx FQ 0x4000740:0 --> Tx FQID 319
Interface fman 0, index 5: enable rx
Interface name fm1-mac5: enabled RX
Mapping Rx FQ 0x4000780:0 --> Tx FQID 321
Mapping Rx FQ 0x40007c0:0 --> Tx FQID 321
Interface fman 0, index 6: enable rx
```

```
Interface name fm1-mac6: enabled RX
Released 0 bufs to BPID 34
Released 16384 bufs to BPID 16
cpu0/0: > WARNING (FM-PCD) [CPU00, drivers/net/ethernet/freescale/fman/Peripheral
s/FM/Pcd/fm_kg.c:1074 BuildSchemeRegs]:
cpu0/0: baseFqid is 0.cpu0/0:
cpu0/0: > WARNING (FM-PCD) [CPU00, drivers/net/ethernet/freescale/fman/Peripheral
s/FM/Pcd/fm_kg.c:1074 BuildSchemeRegs]:
cpu0/0: baseFqid is 0.cpu0/0:
IB policy verification not activated
Identifying IPv4 routing tables
        0) "ob_post_ipv4_route_cc" ... cc_node=0x10c04200
        1) "ib_post_ipv4_route_cc" ... cc_node=0x10c03f70
2 tables.
Identifying IPv4 rule tables
        0) "ib_ipv4_rule_cc" ... cc_node=0x108372f8
1 tables.
Identifying IPv6 routing tables
        0) "ob_post_ipv6_route_cc" ... cc_node=0x10c041e8
        1) "ib_post_ipv6_route_cc" ... cc_node=0x10c03f58
2 tables.
Identifying IPv6 rule tables
        0) "ib_ipv6_rule_cc" ... cc_node=0x10bfdb00
1 tables.
Initializing IPv4 route tables...
        - td=rt_table_no=13 ... ccnode=0x10c04200
        - td=rt_table_no=14 ... ccnode=0x10c03f70
Initializing IPv4 rule tables...
        - td=15, ccnode=0x108372f8, ifname=fm1-mac5
Initializing IPv6 route tables...
        - td=rt_table_no=16 ... ccnode=0x10c041e8
        - td=rt_table_no=17 ... ccnode=0x10c03f58
Initializing IPv6 rule tables...
        - td=18, ccnode=0x10bfdb00, ifname=fm1-mac5
Hit Ctrl+C, send SIGINT or write quit to terminate.
>
```

To shut down the application you can type the command *quit* in the application console. At shutdown the application flushes its offloading tables, but it does not flush the items that were configured in the Linux kernel. Therefore, in case you have performed additional route, neighbor or IPSec configurations using the Linux tools it is more than a good idea to revert/flush them as well using the appropriate commands once the application was shut down.

Typing *help* in the application console will display the available application CLI commands.

9.9.2.6.2.3 Running the application

First step to take is to configure the IP addresses on the inbound and outbound interfaces of the host. For example:

```
ip addr add 192.168.100.1/24 dev fm1-mac5

ip addr add 172.16.0.254/16 dev fm1-mac6
```

Before continuing, you need to bring down the VIF interface so that it doesn't interfere with the IPSec policies setup:

```
ip link set dev fm1-mac5 down
```

IPSec can be configured for offloading using the *setkey* tool. To create an IPSec tunnel (SA) and two security policies (one for inbound and one for outbound) add the following lines to a `setkey.conf` text file:

```
flush;
spdflush;

add 192.168.100.1 192.168.200.1 esp 0x201
-E 3des-cbc "abcdefghipqrstuvwxyabcde"
-A hmac-sha1 "abcdefghipqrstuvwxya";

spdadd 172.16.0.1/32[1230] 172.17.0.1/32[2600] udp
    -P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[2600] 172.16.0.1/32[1230] udp
    -P in ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
```

To execute the script and set up the tunnels, you can run the command:

```
setkey -f setkey.conf
```

These commands will create an ESP tunnel with the following endpoints: IP src 192.168.100.1 - IP dst 192.168.200.1, SPI 0x201, 3DES-CBC encryption and HMAC-SHA1 authentication. The UDP traffic coming from 172.16.0.1, with UDP source port 2130 and going to 172.17.0.1, with UDP destination port 2600 will be directed through the defined tunnel.

Once the IPSec tunnels are set up, we can bring the VIF interface back up:

```
ip link set dev fm1-mac5 up
```

You are ready now to add your IP gateways and IP routes using commands like:

```
ip neigh add 172.16.0.1 lladdr 00:10:18:BA:E4:04 dev fm1-mac6
ip neigh add 192.168.100.254 lladdr 68:05:ca:12:2f:0f dev fm1-mac5
```

When running in loopback mode you should configure the next hop on the inbound port with its own Ethernet address (marked in bold in the example above).

IP routes can be configured using the *ip route* tool as in:

```
ip route add 192.168.200.0/24 via 192.168.100.254 table 13 dev fm1-mac5
ip route add 172.17.0.0/16 via 172.16.0.1 table 14 dev fm1-mac6
```

The route table numbers can be found in the application startup information by looking after the *Initializing IP route tables* section. The first table in the section is always the outbound route table, while the second is the inbound route table. Please note that there are different initialization sections for IPv4 and IPv6 routing tables.

IP ingress rules can currently be configured for the inbound direction only. For this purpose you have the *ib_rule_add4*, *ib_rule_add6*, *ib_rule_del4* and *ib_rule_del6* commands available. These rules apply to the clear traffic received on the unprotected (inbound) interface. The usual syntax of an *add rule* command is:

```
ib_rule_add4 <src_ip>/<prefix> <dst_ip>/<prefix> <priority> <dest_route_table_no> [ <hex_tos> ]
```

where

- *src_ip*/*prefix* is the source net IP address;
- *dst_ip*/*prefix* is the destination net IP address;
- *priority* is the priority level that you want this rule to have relative to the other rules in the ingress IP rule table; please note that there cannot be two rules with the same priority;
- *dest_route_table_no* is the number of the **outbound** route table that this flow will be directed to; identification of the IP route tables was described earlier in this paragraph;

- *hex_tos* is an optional argument specifying the *Type Of Service* or *Traffic Class* in hexadecimal representation.

IP rules apply to inbound clear traffic allowing it to be propagated to the specified outbound IP routing tables. All inbound clear traffic flows that does not match any of the configured ingress IP rules is silently discarded.

The application also supports displaying IPSec statistics. You can get SA related statistics using the *sa_stats* CLI command and also global IPSec statistics using the *ipsec_stats* CLI command.

9.9.2.7 References

1. [P4080/B4860/B4420] Integrated Multicore Communication Processor Family Reference Manual (pdf)
2. USDPAA PPAC User Guide (Knowledge Center)
3. QMan/BMan API Reference (Knowledge Center)

9.9.2.8 Appendix A – Preparing DPA Offloading DTB Files

Detailed reference for compiling the device tree for the following DPAA offloading scenarios:

1. USDPAA
2. IP Offloading
3. IP Reassembly
4. Shared interfaces
5. NF Offloading

For these SoC products:

1. P2041
2. P4080
3. B4860
4. B4420
5. T4240
6. T2080
7. LS1043A (only USDPAA, IP Offloading and IP Reassembly scenarios)

9.9.2.8.1 Compiling the device tree for USDPAA applications

Compiling the device tree for P4080

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/p4080ds-usdpaa.dts
   arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p4080ds-usdpaa.dtb
   arch/powerpc/boot/dts/p4080ds-usdpaa.dts
```

The DTB file `p4080ds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for P2041

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/p2041rdb-usdpaa.dts
   arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p2041rdb-usdpaa.dtb
   arch/powerpc/boot/dts/p2041rdb-usdpaa.dts
```

The DTB file `p2041rdb-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for B4860

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa.dts
   arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa.dtb
   arch/powerpc/boot/dts/b4860qds-usdpaa.dts
```

The DTB file `b4860qds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for B4420

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/b4420qds-usdpaa.dts
   arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4420qds-usdpaa.dtb
   arch/powerpc/boot/dts/b4420qds-usdpaa.dts
```

The DTB file `b4420qds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for T4240

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/t4240qds-usdpaa.dts
   arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t4240qds-usdpaa.dtb
   arch/powerpc/boot/dts/t4240qds-usdpaa.dts
```

The DTB file `t4240qds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for T2080

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/t2080qds-usdpaa.dts
   arch/powerpc/boot/dts
```

```
scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t2080qds-usdpaa.dtb
   arch/powerpc/boot/dts/t2080qds-usdpaa.dts
```

The DTB file `t2080qds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for LS1043A

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a-rdb-usdpaa.dts arch/arm64/boot/dts/freescale

make freescale/fsl-ls1043a-rdb-usdpaa.dtb
```

The DTB file `fsl-ls1043a-rdb-usdpaa.dtb` will be generated in the `arch/arm64/boot/dts/freescale` subdirectory. LS1043A boot up mechanism may require also that you build an *ITB* file which aggregates your Linux kernel image, *DTB* file and your root file system (in case you are using RAMBOOT). This file can be generated using the *mkimage* tool.

9.9.2.8.2 Compiling the device tree for IP offloading

Compiling the device tree for P4080

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/p4080ds-usdpaa.dts
arch/powerpc/boot/dts/

cp drivers/staging/fsl_dpa_offload/dts/p4080si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/p4080si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/p4080si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p4080ds-usdpaa-offload.dtb
arch/powerpc/boot/dts/p4080ds-usdpaa.dts
```

The DTB file `p4080ds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for P2041

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/p2041rdb-usdpaa.dts
arch/powerpc/boot/dts/

cp drivers/staging/fsl_dpa_offload/dts/p2041si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/p2041si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/p2041si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p2041rdb-usdpaa-offload.dtb
arch/powerpc/boot/dts/p2041rdb-usdpaa.dts
```

The DTB file `p2041rdb-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for B4860

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa.dts
 arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o
 b4860qds-usdpaa-offload.dtb
arch/powerpc/boot/dts/b4860qds-usdpaa.dts
```

The DTB file `b4860qds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for B4420

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/b4420qds-usdpaa.dts
 arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/b4420si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4420si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/b4420si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o
 b4420qds-usdpaa-offload.dtb
arch/powerpc/boot/dts/b4420qds-usdpaa.dts
```

The DTB file `b4420qds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for T4240

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/t4240qds-usdpaa.dts
 arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/t4240si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t4240si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/t4240si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o
 t4240qds-usdpaa-offload.dtb arch/powerpc/boot/dts/t4240qds-usdpaa.dts
```

The DTB file `t4240qds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for T2080

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/t2080qds-usdpaa.dts
 arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/t208xsi-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t2080si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/t2080si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o
 t2080qds-usdpaa-offload.dtb arch/powerpc/boot/dts/t2080qds-usdpaa.dts
```

The DTB file `t2080qds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for LS1043A

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a-rdb-usdpaa.dts arch/arm64/boot/dts/freescale

cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a.dtsi arch/arm64/boot/dts/freescale

cp drivers/staging/fsl_dpa_offload/dts/ls1043a-chosen-offload.dtsi arch/arm64/boot/dts/freescale/
ls1043a-chosen.dtsi

make freescale/fsl-ls1043a-rdb-usdpaa.dtb
```

The DTB file `fsl-ls1043a-rdb-usdpaa.dtb` will be genereted in the `arch/arm64/boot/dts/freescale` subdirectory. LS1043A boot up mechanism may require also that you build an *ITB* file which aggregates your Linux kernel image, *DTB* file and your root file system (in case you are using RAMBOOT). This file can be generated using the *mkimage* tool.

9.9.2.8.3 Compiling the device tree for IP reassembly

Compiling the device tree for P4080

On P4080 there is no specific device tree building process for IP reassembly. You can use the one described in **Compiling the Device Tree for USDPAA Applications**.

Compiling the device tree for P2041

On P2041 there is no specific device tree building process for IP reassembly. You can use the one described in **Compiling the Device Tree for USDPAA Applications**.

Compiling the device tree for B4860

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa.dts
arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
```

```
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-reass.dtsi
 arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa-reass.dtb
arch/powerpc/boot/dts/b4860qds-usdpaa.dts
```

The DTB file `b4860qds-usdpaa-reass.dtb` will be built in the current directory.

Compiling the device tree for B4420

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4420qds-usdpaa.dts
arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/b4420si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4420si-chosen-reass.dtsi
 arch/powerpc/boot/dts/fsl/b4420si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4420qds-usdpaa-reass.dtb
arch/powerpc/boot/dts/b4420qds-usdpaa.dts
```

The DTB file `b4420qds-usdpaa-reass.dtb` will be built in the current directory.

Compiling the device tree for T4240

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/t4240qds-usdpaa.dts
arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/t4240si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t4240si-chosen-reass.dtsi
 arch/powerpc/boot/dts/fsl/t4240si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t4240qds-usdpaa-reass.dtb
arch/powerpc/boot/dts/t4240qds-usdpaa.dts
```

The DTB file `t4240qds-usdpaa-reass.dtb` will be built in the current directory.

Compiling the device tree for T2080

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/t2080qds-usdpaa.dts
arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/t208xsi-pre.dtsi
 arch/powerpc/boot/dts/fsl
```

```
cp drivers/staging/fsl_dpa_offload/dts/t2080si-chosen-reass.dtsi
 arch/powerpc/boot/dts/fsl/t2080si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t2080qds-usdpaa-reass.dtb
arch/powerpc/boot/dts/t2080qds-usdpaa.dts
```

The DTB file `t2080qds-usdpaa-reass.dtb` will be built in the current directory.

Compiling the device tree for LS1043A

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a-rdb-usdpaa.dts arch/arm64/boot/dts/freescale

cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a.dtsi arch/arm64/boot/dts/freescale

cp drivers/staging/fsl_dpa_offload/dts/ls1043a-chosen-reass.dtsi arch/arm64/boot/dts/freescale/
ls1043a-chosen.dtsi

make freescale/fsl-ls1043a-rdb-usdpaa.dtb
```

The DTB file `fsl-ls1043a-rdb-usdpaa.dtb` will be genereted in the `arch/arm64/boot/dts/freescale` subdirectory. LS1043A boot up mechanism may require also that you build an *ITB* file which aggregates your Linux kernel image, *DTB* file and your root file system (in case you are using RAMBOOT). This file can be generated using the *mkimage* tool.

9.9.2.8.4 Compiling the device tree for ipsec offload

Compiling the device tree for P4080

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/p4080si-pre.dtsi
 arch/powerpc/boot/dts/fsl
```

```
cp drivers/staging/fsl_dpa_offload/dts/p4080si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/p4080si-chosen.dtsi
```

```
cp drivers/staging/fsl_dpa_offload/dts/p4080ds-usdpaa-shared-interfaces.dts
 arch/powerpc/boot/dts
```

```
scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p4080ds-usdpaa-shared-mac.dtb
 arch/powerpc/boot/dts/p4080ds-usdpaa-shared-interfaces.dts
```

The DTB file `p4080ds-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for P2041

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/p2041si-pre.dtsi
 arch/powerpc/boot/dts/fsl
```

```
cp drivers/staging/fsl_dpa_offload/dts/p2041si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/p2041si-chosen.dtsi
```

```
cp drivers/staging/fsl_dpa_offload/dts/p2041rdb-usdpaa-shared-interfaces.dts
 arch/powerpc/boot/dts
```

```
scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p2041rdb-usdpaa-shared-mac.dtb
 arch/powerpc/boot/dts/p2041rdb-usdpaa-shared-interfaces.dts
```

The DTB file `p2041rdb-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for B4860

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa-shared-interfaces.dts
 arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa-shared-mac.dtb
 arch/powerpc/boot/dts/b4860qds-usdpaa-shared-interfaces.dts
```

The DTB file `b4860qds-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for B4420

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4420si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4420si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/b4420si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/b4420qds-usdpaa-shared-interfaces.dts
 arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4420qds-usdpaa-shared-mac.dtb
 arch/powerpc/boot/dts/b4420qds-usdpaa-shared-interfaces.dts
```

The DTB file `b4420qds-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for T4240

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/t4240si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t4240si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/t4240si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/t4240qds-usdpaa-shared-interfaces.dts
 arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t4240qds-usdpaa-shared-mac.dtb
 arch/powerpc/boot/dts/t4240qds-usdpaa-shared-interfaces.dts
```

The DTB file `t4240qds-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for T2080

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/t208xsi-pre.dtsi
 arch/powerpc/boot/dts/fsl

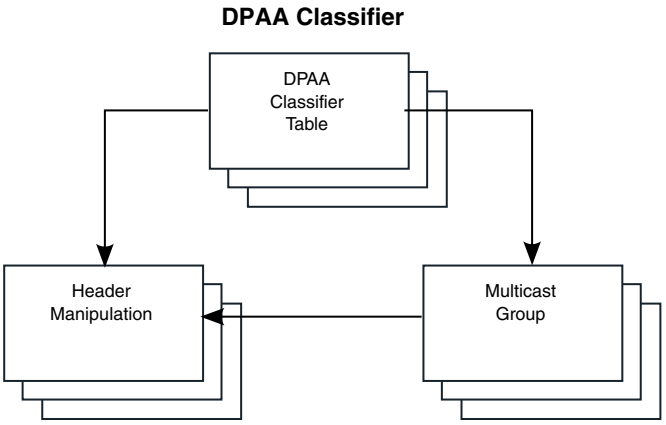
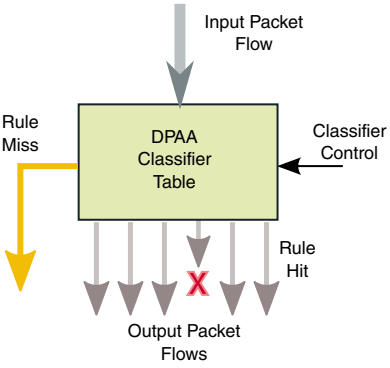
cp drivers/staging/fsl_dpa_offload/dts/t2080si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/t2080si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/t2080qds-usdpaa-shared-interfaces.dts
 arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t2080qds-usdpaa-shared-mac.dtb
 arch/powerpc/boot/dts/t2080qds-usdpaa-shared-interfaces.dts
```

The DTB file `t2080qds-usdpaa-shared-mac.dtb` will be built in the current directory.

9.9.2.8.5 Compiling the device tree for Network Function Layer offloading

Compiling the device tree for B4860

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
 arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-offld.dtsi
 arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa-nf-offload.dts
 arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa-nf-offload.dtb
 arch/powerpc/boot/dts/b4860qds-usdpaa-nf-offload.dts
```

The DTB file `b4860qds-usdpaa-nf-offload.dtb` will be generated in the current directory.

9.9.2.9 Appendix B - Enabling DPA Offloading Drivers in the Linux Kernel

DPA offloading works only with the DPA offloading Linux kernel drivers. In order to build the DPA offloading Linux drivers, you need to follow the steps presented in this paragraph.

Go to your Linux kernel source code directory and type the command

```
make menuconfig
```

In the build menu that is presented, enable the following option

```
Device Drivers
    Staging Drivers
        <*> Freescale Datapath Offloading Driver
```

In this example the driver is built into the Linux kernel. If you would rather use a loadable module, then you can select the "M" option.

```
Device Drivers
    Staging Drivers
        <M> Freescale Datapath Offloading Driver
```

You also need to remember to insmod the driver at runtime before running the DPA offloading applications.

Some of the DPA offloading applications require that the FMan driver resource allocation algorithm be disabled. In order to do that, in the build menu make sure the option

```
Device Drivers
    Network device support
        Ethernet driver support
            Freescale devices
                Frame Manager support
                    Freescale Frame Manager (datapath) support
                        [ ] Enable FMan dynamic resource allocation algorithm
```

is disabled.

The last step is to actually launch the kernel build using the make command.

9.9.2.10 Revision history

This table summarizes revisions to this document.

Table 247. Revision history

Revision	Date	Description
0	08/2013	Initial public release.

9.9.3 DPAA Offloading Drivers Reference Manual

9.9.3.1 Introduction

This document covers some components implementation details, but the main focus is the programming interfaces (APIs) exposed by these components.

9.9.3.2 DPA Classifier

The DPA Classifier is a module which allows software to accelerate lookups and decisions based on frame content using DPA.

The software must provide the initial FMan coarse classification nodes to be managed by the Classifier tables and must set up a set of **classification rules** . Rules can be added, edited or removed at any time. The DPA Classifier table will manage the provided FMan resource (coarse classification node) and may extend it if necessary during runtime.



Figure 251. DPA Classifier control objects

9.9.3.2.1 Table

The DPA Classifier's main control object is the DPA Classifier Table. A general functional diagram of the DPA Classifier table is shown below.



Figure 252. DPA Classifier control objects

Before using the DPA Classifier, the software must create a PCD skeleton. The easiest way to create the PCD skeleton is by using the Fman Configuration tool (FMC).

The user software is responsible for keeping the synchronization between the static FMan resources that it creates at init time and the way it uses DPA Classifier. For instance, it is responsible for properly configuring and linking the FMan KeyGen schemes and the FMan Coarse classification nodes so that the proper keys are extracted from the frames. The DPA Classifier will only assume that the key generation process is correct and fit for that type of table.

A classification rule (also called a classifier entry) is an action that will be executed when the entry is hit. The entries in the table are indexed by lookup key, which can have different formats depending on the table.

The result of a table lookup for an input packet is either:

- a hit condition, which means that the key generated from the input packet data (usually header fields) has matched with the key description associated with a rule in the table or

- a miss condition, which means that the key generated from the input packet data did not match the key description of any entry (rule) in the table.

Depending on the type of the lookup, the miss condition might exist or not. For certain types of lookup (e.g. indexed lookup), the miss condition cannot be defined. The DPA classifier supports the following types of lookups:

Table 248. Port Connections

Lookup Type	Description
Exact match lookup	The hardware will search the table for a key which is an exact sequence of bytes. The key can be used as is, or can be masked. If the key matches exactly over the key of an entry, a hit occurs. If there is no match with any of the keys in the table, a miss situation occurs.
Indexed lookup	The hardware will use a sequence of bits from the key to create an index. The index will be used as a direct index in the table and the action from the entry with that index will be executed. For this type of table there cannot be any miss as the index table is required to be filled with all entries at initialization time. Also, keys from an indexed table cannot be (ever) removed. They can only be modified.
HASH lookup	The hardware will calculate a HASH value out of the key generated from the input packet data, and it will use this masked HASH value as a first index for a table lookup. This will result in finding a HASH set. In the second phase, an exact match lookup is performed among the rules in that particular HASH set to find the exact table entry which matches the lookup key. If a match is found, a lookup hit occurs and the associated action is executed. Otherwise, a lookup miss condition occurs.

Exact match lookup is offering more performance on tables with few entries, while HASH tables offer more performance on densely populated tables. However, the memory consumption for exact match tables is lower and more efficient than for HASH tables, hence it is recommended to use this type of table whenever possible.

The action associated with a table entry falls into 3 possible categories:

1. **Drop frame** action;
2. **Enqueue frame** action - send the frame to a specified frame queue; this action has several options such as enable policing, enable statistics, header manipulation operations;
3. Send frame to a**new lookup** - the frame is directed towards a new DPA Classifier table;
4. **Multicast** - replicate the frame and send it to multiple destinations by using a multicast group.

The DPA Classifier table has two possible ways to manage entries. If entry management **by key** is selected, the DPA Classifier keeps internally a **shadow table** for the entries. This allows it to offer features which are additional to those offered by the low level driver alone, such as lookups and transparent entry index management.

If entry management **by reference** is selected, the DPA Classifier will identify the entries only by their Id (reference), and not by their key. The shadow table is not created in this case. The application will keep the mapping between the keys and their table entry Ids and this will help DPA Classifier to provide more performance for runtime operations.

The DPA Classifier can work with certain *prefilled tables*, which are

- prefilled HASH tables,

- prefilled exact match tables.

Prefilled tables are tables which are populated by the user application before DPA Classifier takes control of the FMan coarse classification node. This is not applicable for indexed tables, where the user always provides all the entries at initialization time. The user application must communicate to the DPA Classifier how many static entries are present in the table at initialization time. If the number of static entries is n, the DPA Classifier module will consider that the static entries are the first n entries in the table and with the highest priority (when priorities are enabled). Working with prefilled tables comes with certain limitations listed below.

- DPA Classifier can only work in key management mode with prefilled tables. Prefilled tables managed by reference are not supported.
- Prefilled classifier HASH tables do not support adding entries with header manipulations.

9.9.3.2.1.1 Table API

There are four selections for the DPA Classifier table API:

- Create and Remove Tables
- Insert, Remove or Update Tables
- Update Default Table Policy
- Table Look-up

9.9.3.2.1.1.1 Create and Remove Tables

Table 249. API to Create and Remove Tables

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_table_create`	Creates a DPA Classifier table using a coarse classification node. This call does not allocate MURAM.	• handle of the initial Cc node; • the type of table to create; • the table size; • other table params.	Descriptor of the newly created DPA Classifier table.
`dpa_classif_table_free`	Releases all resources associated with a DPA Classifier table and destroys it.	• descriptor of the DPA Classifier table to destroy.	None.

9.9.3.2.1.1.1.1 Function: dpa_classif_table_create

Syntax

```
 int dpa_classif_table_create(const struct dpa_cls_tbl_params *params,
int *td);

struct dpa_cls_tbl_params
{
 void      *cc_node;
 void      *distribution;
 void      *classification;
 enum dpa_cls_tbl_type   type;
 enum dpa_cls_tbl_entry_mgmt  entry_mgmt;
 union
```

```
 {
  struct dpa_cls_tbl_hash_params  hash_params;
  struct dpa_cls_tbl_indexed_params  indexed_params;
  struct dpa_cls_tbl_exact_match_params exact_match_params;
 };
 unsigned int    prefilled_entries;
};

enum dpa_cls_tbl_type
{
 DPA_CLS_TBL_HASH = 0,  /**< HASH table */
 DPA_CLS_TBL_INDEXED,  /**< Indexed table */
 DPA_CLS_TBL_EXACT_MATCH /**< Exact match table */
};

struct dpa_cls_tbl_hash_params
{
 unsigned int num_sets;
 unsigned int max_ways;
 unsigned int hash_offs;
 uint8_t  key_size;
};

struct dpa_cls_tbl_indexed_params
{
 unsigned int entries_cnt;
};

struct dpa_cls_tbl_exact_match_params
{
 unsigned int entries_cnt;
 uint8_t  key_size;
 bool   use_priorities;
};

enum dpa_cls_tbl_entry_mgmt
{
 DPA_CLS_TBL_MANAGE_BY_KEY = 0,
 DPA_CLS_TBL_MANAGE_BY_REF
};
```

Parameters

- **params** - data structure containing the parameters of the table to create;
  - **cc_node** - handle of the Cc node to start with; please refer to the function description for more information;
  - **distribution** - handle of a FMan distribution to send frames to instead of enqueuing them. If this handle is provided (not NULL) the enqueue action will only select the frame queue, but it will not actually enqueue the frame to the selected frame queue. Instead it will send the frame to the indicated distribution for further processing.
  - **classification** - handle of a FMan classification to send frames after distribution. For FMan v3 previously determined frame queue will be retained as frame destination. This parameter may be used to perform header manipulations operations such as fragmentation after the enqueue decision has been made.
  - **type** - the type of table to create; configured using dpa_cls_tbl_type enum;
  - **entry_mgmt** - table entry management mechanism for runtime; configured using dpa_cls_tbls_entry_mgmt enum;

NOTE

Not all types of tables support all types of entry management. Indexed tables can only be managed by key, because table entry insert doesn't make any sense on this type of table. Only entry modification is available for this type of tables. In this case there is no way for the classifier to provide the user an entry reference for the entry management by reference mechanism to work.

- **hash_params**
  - num_sets – the number of sets in the HASH table; this controls the table size;
  - max_ways – the maximum number of ways (depth) of the HASH table; this controls the capability of the HASH to resolve conflicts;
  - hash_offs – lets the user control how the HASH value is used to determine the HASH set; this is the offset in bits from the MSB of the calculated HASH value where DPA Classifier should start applying the mask to get the index for HASH set lookup;
  - key_size - the size of the lookup keys in bytes.
- **indexed_params**
  - entries_cnt - the number of entries to be stored in the indexed table;
- **exact_match_params**
  - entries_cnt - maximum number of entries expected to be stored in the table;
  - key_size – the size of the keys in bytes;
  - use_priorities - Use priorities for each entry in table if set to true.
- **prefilled_entries** - Number of entries in the table which are pre-filled (already used). The assumption is always that these entries are the first entries in the table and with the highest priority. There are several limitations for prefilled tables relative to the normal (blank) tables: a) prefilled HASH tables cannot be managed by ref; b) header manipulations created using the DPA Classifier header manipulation API cannot be used on prefilled tables.

- **td** - a location where the function will return the descriptor of the newly created table, in case of success; this descriptor will be further used by the application to refer to this DPA Classfier table in other API function calls.

Description

Configures and initializes a DPA Classifier table using a coarse classification node generated with the FMan Configuration tool. This function never allocates new MURAM space.

Once the DPA Classifier takes control of a FMan Cc node, all management must be performed through its API. If applications use different APIs to modify the Cc node's properties in the same time while the DPA Classifier has ownership of the node, unpredictable behavior and data inconsistency can occur. Each type of DPA Classifier table requires a coarse classification node of a special type as its initial Cc node. The types of the initial Cc nodes relative to the types of DPA classifier tables are specified below.

Table 250.

DPA Classifier table type	Required initial FMan Cc node type
Exact Match Table	Match Table Cc Node
Indexed Table	Indexed Cc Node
HASH Table	HASH Table Cc Node

Depending on the value of the **entry_mgmt** parameter, the behavior of the runtime functions changes. In a table created with `DPA_CLS_TBL_MANAGE_BY_KEY`, runtime modify / delete / stats operations can be accomplished using any of the runtime

function versions. For tables which use DPA_CLS_TBL_MANAGE_BY_REF, only the runtime function versions with the suffix "by_ref" are supported for modification / deletion / stats.

The DPA_CLS_TBL_MANAGE_BY_REF is an entry management mode which is optimized in terms of speed and memory consumption of the DPA Classifier. Management by reference (Id), however, needs the application to take care of the mapping between lookup keys and entry Ids, because the Classifier only uses the entry Ids.

If DPA_CLS_TBL_MANAGE_BY_KEY is selected, the DPA Classifier will keep an internal shadow table for the mapping between the lookup keys and the actual table entries. This way, the user can specify entries also by their key and the DPA Classifier will do the (software) mapping to determine the entry Id.

For the HASH and exact match DPA classifier tables, the miss action of the table is stored separately and is not included in the number of entries requested by the user. Indexed tables never have miss action.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if one of the function arguments or parameters of the table was incorrect;
- -ENOSYS, if a feature selected by the table parameters is not supported;
- -ENOMEM, if there is no more memory to create a new DPA classifier table; this may mean that either the internal management data structures of the table could not be allocated, or the shadow table (if one was requested) could not be allocated;
- -EBUSY, if a FMan low level driver function call has failed.

9.9.3.2.1.1.1.2 Function: dpa_classif_table_free

Syntax

```
int dpa_classif_table_free(int td);
```

Parameters

- **td** - descriptor of the DPA Classifier table to destroy.

Description

Releases all resources associated with a DPA Classifier table and destroys it.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if the provided table descriptor is not valid;
- -EBUSY, if a FMan low level driver function call has failed.

9.9.3.2.1.1.2 Insert, Remove or Update Table Entries

Table 251. API to Insert, Remove or Update Table Entries

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_table_insert_entry`	Adds an entry (classification rule) in the specified DPA Classifier table.	• descriptor of the DPA Classifier table where the entry needs to be added; • the key description (its format depends on the type of table); • the action description in case of hit. • entry priority (if necessary).	The new entry's reference (or entry Id) in case of success.
`dpa_classif_table_delete_entry_by_key`	Removes an entry in the specified DPA Classifier table. The entry is identified by the lookup key.	• descriptor of the DPA Classifier table containing the entry to be removed; • the key description of the entry to be removed.	None.
`dpa_classif_table_delete_entry_by_ref`	Removes an entry in the specified DPA Classifier table. The entry is identified by its reference (Id).	• descriptor of the DPA Classifier table containing the entry to be removed; • the reference of the entry to be removed.	None.
`dpa_classif_table_modify_entry_by_key`	Modifies an entry in the specified DPA Classifier table. The entry is identified by the lookup key.	• descriptor of the DPA Classifier table containing the entry to be modified; • the key description of the entry to be modified; • the new key description to replace the existing key description; • the new action description in case of hit.	None.

Table continues on the next page...

Table 251. API to Insert, Remove or Update Table Entries (continued)

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_table_modify_entry_by_ref`	Modifies an entry in the specified DPA Classifier table. The entry is identified by reference (Id).	• descriptor of the DPA Classifier table containing the entry to be modified; • the reference of the entry to be modified; • the new key description to replace the existing key description; • the new action description in case of hit.	None.
`dpa_classif_table_get_entry_stats_by_key` **NOTE** As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.	Returns statistics for a specified entry in a specified table. Optionally reset the statistics once they are read. The entry is identified by the lookup key.	• descriptor of the DPA Classifier table containing the entry to return statistics for; • the key description of the classification rule to return statistics for; • indication whether the statistics should be reset once they are read.	The requested entry statistics.
`dpa_classif_table_get_entry_stats_by_ref` **NOTE** As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.	Returns statistics for a specified entry in a specified table. Optionally reset statistics once they are read. The entry is identified by its reference (Id).	• descriptor of the DPA Classifier table containing the entry to return statistics for; • reference of the classification rule to return statistics for; • indication whether the statistics should be reset once they are read.	The requested entry statistics.

Table continues on the next page...

Table 251. API to Insert, Remove or Update Table Entries (continued)

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_table_get_miss_stats` **NOTE** As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.	Returns miss statistics for a specified table. The table is identified by its descriptor (td).	descriptor of the DPA Classifier table to return statistics for.	The requested table miss statistics.

9.9.3.2.1.1.2.1 Function: dpa_classif_table_insert_entry

Syntax

```
int dpa_classif_table_insert_entry(
 int       td,
 const struct dpa_offload_lookup_key  *key,
 const struct dpa_cls_tbl_action  *action,
 int       priority,
 int       *entry_id);

struct dpa_cls_tbl_action
{
 enum dpa_cls_tbl_action_type    type;
 bool        enable_statistics;
 union
 {
  struct dpa_cls_tbl_enq_action_desc  enq_params;
  struct dpa_cls_tbl_next_table_desc  next_table_params;
  struct dpa_cls_tbl_mcast_group_desc  mcast_params;
 };
};

enum dpa_cls_tbl_action_type
{
 DPA_CLS_TBL_ACTION_NONE = 0,
 DPA_CLS_TBL_ACTION_DROP,
 DPA_CLS_TBL_ACTION_ENQ,
 DPA_CLS_TBL_ACTION_NEXT_TABLE,
 DPA_CLS_TBL_ACTION_MCAST
};

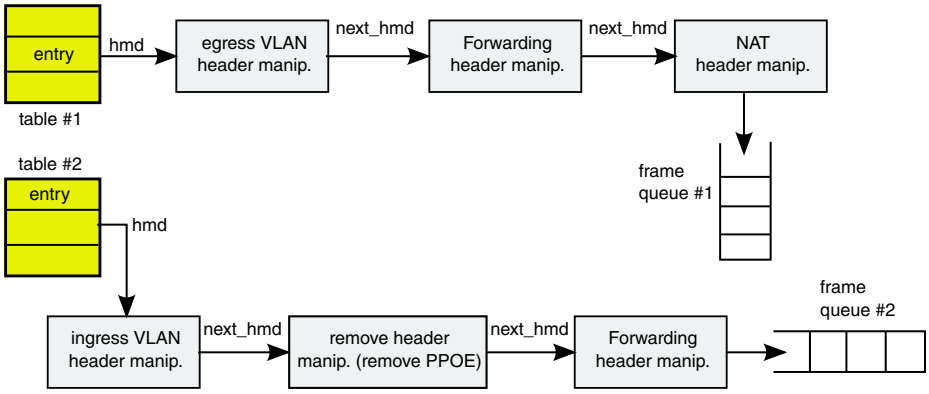
struct dpa_cls_tbl_enq_action_desc
{
 bool      override_fqid;
 uint32_t     new_fqid;
```

```
 struct dpa_cls_tbl_policer_params *policer_params;
 int      hmd;
 uint8_t     new_rel_vsp_id;
};

struct dpa_cls_tbl_next_table_desc
{
 int  next_td;
 int  hmd;
};

struct dpa_cls_tbl_mcast_group_desc
{
 int  grpd;
 int  hmd;
};

struct dpa_cls_tbl_policer_params
{
 bool   modify_policer_params;
 bool   shared_profile;
 unsigned int new_rel_profile_id;
};
```

Parameters

- **td** - descriptor of the DPA Classifier table to insert in;
- **key** - pointer to the key description (its format depends on the type of table);
- **action** - the action description in case of hit;
  - **type** – the type of action to take: drop, enqueue, send to next table, send to multicast group; the drop action descriptor type doesn't need any further parameters; selected from the dpa_cls_tbl_action_type enum; action type NONE is not accepted;
  - **enable_statistics** – true to enable statistics for this action; this attribute is ignored if the action is of type send to next table;
  - **enq_params**
    - **override_fqid** – true if the attribute new_fqid will override the KeyGen selected frame queue;
    - **new_fqid** – if override_fqid is true, this holds the explicit frame queue Id where to place the frame;
    - **policer_params** – policing parameters; if NULL, no policing is performed during the enqueue operation
      - **modify_policer_params** – true if the default policer parameters will be overridden;
      - **shared_profil** - true if this policer profile is shared between ports; relevant only if modify_policer_params is also true;
      - **new_rel_profile_id** - this parameter should indicate the policer profile offset within the port's policer profiles or from the SHARED window; relevant only if modify_policer_params is true;
    - **hmd** – header manipulation object descriptor, or DPA_OFFLD_DESC_NONE if no header manipulation is required for the enqueue action; the header manipulation object defined by the provided descriptor must be a header manipulation chain head;
    - **new_rel_vsp_id** – new virtual storage profile Id. This parameter is mandatory when override_fqid is set to true and the port has virtual storage profiles defined. Otherwise it is not used..
  - **mcast_params** – this parameter is used when the action is multicast.

  - **grpd** – multicast group descriptor. This value identifies an existing multicast group.
  - **hmd** – header manipulation object descriptor, or DPA_OFFLD_DESC_NONE if no header manipulation is required. The header manipulation object defined by the provided descriptor must be a header manipulation chain head. The header manipulation operation will be performed before sending the frame to the multicast group.
- **next_table_params**
  - **next_td** – descriptor of the next DPA Classifier table where the frame will be sent for further classification.
  - **hmd** - descriptor of the header manipulation chain to use before sending the frames to the next table, or DPA_OFFLD_DESC_NONE if no header manipulation is required. The header manipulation object defined by the provided descriptor must be a header manipulation chain head.
- **priority** - the priority of the entry. This parameter is meaningful only if td is an exact match table with priority per entries. The priority value of the entry influences the position of the entry in the table relative to the other entries. Entries with lower priority values go to the top of the table. Priority values can be negative. If two entries with the same priority are inserted in the table, they will be positioned one after the other in the table, with the older one first.
- **entry_id** - reference to the newly created entry returned by this function on success.

Description

Adds an entry (classification rule) in the specified DPA Classifier table. If the MURAM for the table was pre-allocated, this operation doesn't consume MURAM.

NOTE

The hardware currently doesn't support longest prefix match on the exact match or HASH tables. If there are more entries in the table that match the lookup (e.g. because of their mask) the first one will always be returned by the hardware lookup.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EEXIST`, if the specified table entry already exists;
- `-EINVAL`, if there is an error in the provided entry parameters;
- `-EBUSY`, if a FMan low level driver function call has failed;
- `-ENOMEM`, if there is no more memory for adding or managing a new table entry;
- `-ENOSPC`, if there is no more room to add a new table entry (table is full).

9.9.3.2.1.1.2.2 Function: dpa_classif_table_delete_entry_by_key

Syntax

```
int dpa_classif_table_delete_entry_by_key(
 int      td,
 const struct dpa_offload_lookup_key *key);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to be removed;
- **key** - the key description of the entry to be removed.

Description

Removes an entry in the specified DPA Classifier table. The entry is identified by the lookup key. If the MURAM for the table was pre-allocated, this function doesn't free up any MURAM space.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;;
- - `ENOSYS`, if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;
- `-ENODEV`, if the entry with the specified key was not found in the table;
- `-EBUSY`, if a FMan low level driver function call has failed.

9.9.3.2.1.1.2.3 Function: dpa_classif_table_delete_entry_by_ref

Syntax

```
int dpa_classif_table_delete_entry_by_ref(int td, int entry_id);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to be removed;
- **entry_id** - the reference (Id) of the entry to be removed.

Description

Removes an entry in the specified DPA Classifier table. The entry is identified by its reference (Id). If the MURAM for the table was pre-allocated, this function doesn't free up any MURAM space.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-EBUSY`, if a FMan low level driver function call has failed.

9.9.3.2.1.1.2.4 Function: dpa_classif_table_modify_entry_by_key

Syntax

```
int dpa_classif_table_modify_entry_by_key(
    int                             td,
const struct dpa_offload_lookup_key        *key,
    const struct dpa_cls_tbl_entry_mod_params    *mod_params);

enum dpa_cls_tbl_modify_type
{
    DPA_CLS_TBL_MODIFY_KEY = 0,
    DPA_CLS_TBL_MODIFY_ACTION,
    DPA_CLS_TBL_MODIFY_KEY_AND_ACTION
};
```

```
struct dpa_cls_tbl_entry_mod_params
{
    enum dpa_cls_tbl_modify_type        type;
    struct dpa_offload_lookup_key        *key;
    struct dpa_cls_tbl_action       *action;
};
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to be modified;
- **key** - the key description of the entry to be modified;
- **mod_params** - the new parameters to replace the existing key descriptor and/or action descriptor;
  - **type** - specifies the desired modification; can be either key descriptor modification only, action descriptor modification only, or both; configured using the dpa_cls_tbl_modify_type enum;
  - **key** - new key descriptor to replace the descriptor that is indexing this entry; this is ignored if the type of modification is `DPA_CLS_TBL_MODIFY_ACTION`;
  - **action** - the new action description in case of hit; this is ignored if the type of modification is `DPA_CLS_TBL_MODIFY_KEY`.

Description

Modifies an entry in the specified DPA Classifier table. The entry is identified by the lookup key. This function never allocates new MURAM space.

The entry modification types `DPA_CLS_TBL_MODIFY_KEY` and `DPA_CLS_TBL_MODIFY_KEY_AND_ACTION` are only available for exact match tables.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the required type of modification is not supported or if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;
- `-EBUSY`, if a FMan low level driver function call has failed.

9.9.3.2.1.1.2.5 Function: dpa_classif_table_modify_entry_by_ref

Syntax

```
int dpa_classif_table_modify_entry_by_ref(
    int                             td,
    int                             entry_id,
    const struct dpa_cls_tbl_entry_mod_params    *mod_params);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to be modified;
- **entry_id** - the reference (Id) of the entry to be modified;
- **mod_params** - the new parameters to replace the existing key descriptor and/or action descriptor;

Description

Modifies an entry in the specified DPA Classifier table. The entry is identified by its reference (Id). This function never allocates new MURAM space. The entry modification types `DPA_CLS_TBL_MODIFY_KEY` and `DPA_CLS_TBL_MODIFY_KEY_AND_ACTION` are only available for exact match tables.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the required type of modification is not supported;
- `-EBUSY`, if a FMan low level driver function call has failed.

9.9.3.2.1.1.2.6 Function: dpa_classif_table_flush

Syntax

```
int dpa_classif_table_flush(int td);
```

Parameters

- **td** - descriptor of the DPA Classifier table to flush;

Description

Removes all the entries in a DPA Classifier Table. After this operation is completed the entries cannot be recovered.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;;
- `-EBUSY`, if a FMan low level driver call has failed.

9.9.3.2.1.1.2.7 Function: dpa_classif_table_get_entry_stats_by_key

Syntax

NOTE

As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.

```
int dpa_classif_table_get_entry_stats_by_key(
 int       td,
 const struct dpa_offload_lookup_key  *key,
 struct dpa_cls_tbl_entry_stats  *stats);

struct dpa_cls_tbl_entry_stats
{
    uint32_t    pkts;
    uint32_t    bytes;
};
```

Parameters

- **td** - of the DPA Classifier table containing the entry to return statistics for;
- **key** - key description of the classification rule to return statistics for;
- **stats** - pointer to a structure allocated especially to receive the requested statistics;
  - **pkts** - the total number of packets that have hit the entry;
  - **bytes** - the total number of bytes in all the packets that have hit the entry.

Description

Returns the statistics for a specified entry in a specified table. The entry is identified by the lookup key.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;
- `-ENODEV`, if the entry with the specified key was not found in the table.

9.9.3.2.1.1.2.8 Function: dpa_classif_table_get_entry_stats_by_ref

Syntax

NOTE

As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.

```
int dpa_classif_table_get_entry_stats_by_ref(
    int                         td,
    int                         entry_id,
    struct dpa_cls_tbl_entry_stats    *stats);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to return statistics for;
- **entry_id** - reference (Id) of the classification rule to return statistics for;
- **stats** - pointer to a structure allocated especially to receive the requested statistics.

Description

Returns the statistics for a specified entry in a specified table. The entry is identified by its reference (Id).

Return Value

The function returns zero on success or a negative error code otherwise. The returned error code is the following:

- `-EINVAL`, if there are errors in the parameters provided to the function.

9.9.3.2.1.1.2.9 Function: dpa_classif_table_get_miss_stats

Syntax

NOTE

As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.

```
int dpa_classif_table_get_miss_stats(
    int                         td,
    struct dpa_cls_tbl_entry_stats    *stats);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to return statistics for;
- **stats** - pointer to a structure allocated especially to receive the requested statistics.

Description

Returns the statistics for a specified table identified by its descriptor (td).

Return Value

The function returns zero on success or a negative error code otherwise. The returned error code is the following:

- `-EINVAL`, if there are errors in the parameters provided to the function.
- `-EPERM`, if there are errors while retrieving the statistics.

9.9.3.2.1.1.3 Update Default Table Policy

Table 252. API to Update Default Table Policy

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_table_modify_miss_action`	Modifies the action taken in case of lookup miss condition.	• descriptor of the DPA Classifier table to modify the miss action for; • pointer to the new action description in case of lookup miss.	None.

9.9.3.2.1.1.3.1 Function: dpa_classif_table_modify_miss_action

Syntax

```
int dpa_classif_table_modify_miss_action(
int                         td,
const struct dpa_cls_tbl_action    *miss_action);
```

Parameters

- **td** - descriptor of the DPA Classifier table to modify the miss action for;
- **miss_action** - pointer to the new action description in case of lookup miss;

Description

Modifies the action taken in case of lookup miss condition.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the user requested an unsupported modification of the miss action;
- `-EBUSY`, if a FMan low level driver function call has failed.

9.9.3.2.1.1.4 Table Lookup

Table 253. API for Table Lookup

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_table_lookup_by_key`	Performs a software lookup in the specified table. If successful (i.e. the entry exists in that table) the action descriptor for that entry is returned. The entry is identified by the lookup key.	• descriptor of the DPA Classifier table to search in; • pointer to the key descriptor of the entry to search for.	The action descriptor properties in case the entry is found in the table.
`dpa_classif_table_lookup_by_ref`	Performs a software lookup in the specified table. If successful (i.e. the entry exists in that table) the action descriptor for that entry is returned. The entry is identified by its reference (Id).	• descriptor of the DPA Classifier table to search in; • the reference of the entry to search for;	The action descriptor properties in case the entry is found in the table.

9.9.3.2.1.1.4.1 Function: dpa_classif_table_lookup_by_key

Syntax

```
int dpa_classif_table_lookup_by_key(
int                             td,
    const struct dpa_offload_lookup_key        *key,
struct dpa_cls_tbl_action            *action);
```

Parameters

- **td** - descriptor of the DPA Classifier table to search in;
- **key** - pointer to the key descriptor of the entry to search for;
- **action** - pointer to a specially allocated structure to receive the action descriptor properties in case the entry is found in the table; this structure is to be considered valid only when the function returns zero.

Description

Performs a lookup in the specified table for an entry specified by a key. If successful (i.e. the entry exists in that table) the action descriptor for that entry is returned. This function works only if entry management by key was selected for the table.

Please be aware that this is not a hardware accelerated lookup. This lookup is performed by the DPA Classifier in its internal shadow tables. It is recommended to use this function with consideration.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;
- `-ENODEV`, if the entry with the specified key was not found in the table.

9.9.3.2.1.1.4.2 Function: dpa_classif_table_lookup_by_ref

Syntax

```
int dpa_classif_table_lookup_by_ref(
int                     td,
    int                     entry_id,
    struct dpa_cls_tbl_action    *action);
```

Parameters

- **td** - descriptor of the DPA Classifier table to search in;
- **key** - pointer to the key descriptor of the entry to search for;
- **action** - pointer to a specially allocated structure to receive the action descriptor properties in case the entry is found in the table; this structure is to be considered valid only when the function returns zero.

Description

Returns the action descriptor for an entry specified by a reference (Id) existing in a specified table. This function is using the internal shadow tables of the DPA Classifier, hence it can be called only for tables created with the `DPA_CLS_TBL_MANAGE_BY_KEY` flag.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;

9.9.3.2.2 Header Manipulation

Header manipulation can be performed on any frame in the following situations:

- when it hits the enqueue action,
- when it is sent to a multicast group or
- when it is sent to a new classification (new lookup).

The available header manipulations are designed based on specific network processes and are listed below.

- NAT (Network Address and Port Translation)
- Forwarding
- Remove header manipulation
- Insert header manipulation
- Update L3/L4 header manipulation
- VLAN specific header manipulation
- MPLS specific header manipulation

Each of these header manipulations can consist of one or more header manipulation **operations**, but this is transparent to the DPA Classifier user. Some of the header manipulation operations may be optional and can be disabled. Header manipulations can be chained in sequence to create more complex actions however, care must be taken at the order of operations. Certain operations may be impacted by previous insert or remove operations which modify the offsets of the headers in the frame.



Figure 253. Header manipulation operations chain example

A header manipulation operations chain is always created starting from the tail and working your way towards the head of the chain. The last header manipulation in the chain is identified by passing `next_hmd=DPA_OFFLD_DESC_NONE` (no other header manipulations are beyond it). The chain head is identified by calling the appropriate header manipulation operation create function using the argument `chain_head=true`. When DPA Classifier identifies the chain head, it initializes the low level driver resources for the entire header manipulation chain. The header manipulation operations chain is now ready to be attached to a DPA Classifier table entry.

To attach an existing header manipulation chain to a DPA Classifier table entry, one needs to set the hmd attribute of the enqueue parameters (`dpa_cls_tbl_enq_action_desc`) related to the new table entry (`dpa_cls_tbl_action`) to point to the descriptor of the header manipulation chain head. Setting the hmd to point to any other header manipulation operation different from a chain head will result in an error. Once the table insert function is called, the table entry and the header manipulation chain enter effect. It is **not possible** to use header manipulations in tables created in different PCDs. It is possible to share the same header manipulation operations **chain** on multiple table entries.

When there is no more need for the header manipulation chain, the header manipulation operations that make it up can be removed in any order. The DPA Classifier will refuse to free header manipulation operations belonging to chains which are in use (i.e. still attached to table entries). One must delete all table entries or detach the header manipulation chain from all entries in order to be able to remove operations from the chain. The low level driver resources are removed only when the chain head operation is removed. It is **not possible** to reuse parts of a header manipulation chain. Header manipulation operation parameters can be modified at runtime using a set of “modify” functions. However, the user application cannot change the **type** of the header manipulation operation. The proper modify function must be called for the type of the existing header manipulation operation. If the header manipulation operation type is not matched to the **modify** function, an error is returned by the **modify** function.

9.9.3.2.2.1 Header Manipulation API

There are two selections for the DPA Classifier table API:

9.9.3.2.2.1.1 Create and Remove Operations

Table 254. API to Create and Remove Header Manipulation Operations

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_set_nat_hm`	Creates / imports a NAT type header manipulation object.	• NAT header manipulation parameters; • descriptor of the next header manipulation object in chain; • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created.	The descriptor of the current header manipulation object once it is created.
`2 dpa_classif_set_fwd_hm`	Creates / imports a forwarding type header manipulation object.	• forwarding header manipulation parameters; • descriptor of the next header manipulation object in chain. • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created.	The descriptor of the current header manipulation object once it is created.

Table continues on the next page...

Table 254. API to Create and Remove Header Manipulation Operations (continued)

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_set_remove_hm`	Creates / imports a remove type header manipulation object.	• ingress (remove) header manipulation parameters; • descriptor of the next header manipulation object in chain. • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created.	The descriptor of the current header manipulation object once it is created.
`dpa_classif_set_insert_hm`	Creates / imports an insert type header manipulation object.	• egress insert header manipulation parameters; • descriptor of the next header manipulation object in chain; • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created.	The descriptor of the current header manipulation object once it is created.
`dpa_classif_set_update_hm`	Creates / imports an update type header manipulation object.	• egress update header manipulation parameters; • descriptor of the next header manipulation object in chain. • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created.	The descriptor of the current header manipulation object once it is created.

Table continues on the next page...

Table 254. API to Create and Remove Header Manipulation Operations (continued)

Function Name	Description	Input Parameters	Output Parameters
dpa_classif_set_vlan_hm	Creates / imports a VLAN specific header manipulation (either ingress or egress) object.	• VLAN specific header manipulation parameters; • descriptor of the next header manipulation object in chain. • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created.	The descriptor of the current header manipulation object once it is created.
dpa_classif_set_mpls_hm	Creates / imports a MPLS specific header manipulation object..	• MPLS specific header manipulation parameters; • descriptor of the next header manipulation object in chain. • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created.	The descriptor of the current header manipulation object once it is created.
dpa_classif_free_hm	Releases a header manipulation object.	• descriptor of the header manipulation object to destroy.	None.

9.9.3.2.2.1.1.1 Function: dpa_classif_set_nat_hm

Syntax

```
int dpa_classif_set_nat_hm(const struct dpa_cls_hm_nat_params *nat_params,
     int next_hmd, int *hmd, bool chain_head,
const struct dpa_cls_hm_nat_resources *res);

/* Definition of a NAT related header manipulation */
struct dpa_cls_hm_nat_params {
 int        flags;
 enum dpa_cls_hm_nat_proto    proto;
 enum dpa_cls_hm_nat_type     type;
```

```
 union {
        struct dpa_cls_hm_traditional_nat_params nat;
        struct dpa_cls_hm_nat_pt_params  nat_pt;
 };
 uint16_t       sport;
 uint16_t       dport;
 /*
  * More attributes may be added to this structure when ICMP NAT
  * support will be available.
  */
 void        *fm_pcd;
 bool        reparse;
};

/* NAT header manipulation low level driver resources */
struct dpa_cls_hm_nat_resources {
 void *l3_update_node;
 void *l4_update_node;
};

/* Supported protocols for NAT header manipulations */
enum dpa_cls_nat_proto {
 DPA_CLS_NAT_PROTO_UDP,
 DPA_CLS_NAT_PROTO_TCP,
 DPA_CLS_NAT_PROTO_ICMP,
 DPA_CLS_NAT_PROTO_LAST_ENTRY
};

/* NAT operation type */
enum dpa_cls_hm_nat_type {
 DPA_CLS_HM_NAT_TYPE_TRADITIONAL,
 DPA_CLS_HM_NAT_TYPE_NAT_PT,
 DPA_CLS_HM_NAT_TYPE_LAST_ENTRY
};

/*
 * Flag values indicating the possible fields to be updated with the
 * NAT header manipulation
 */
enum dpa_cls_hm_nat_flags {
 DPA_CLS_HM_NAT_UPDATE_SIP = 0x01,
 DPA_CLS_HM_NAT_UPDATE_DIP = 0x02,
 DPA_CLS_HM_NAT_UPDATE_SPORT  = 0x04,
 DPA_CLS_HM_NAT_UPDATE_DPORT  = 0x08,
};

/* Traditional NAT parameters */
struct dpa_cls_hm_traditional_nat_params {
 struct dpa_offload_ip_address  sip;
 struct dpa_offload_ip_address  dip;
};

/* NAT-PT parameters */
struct dpa_cls_hm_nat_pt_params {
 enum dpa_cls_hm_nat_pt_type  type;
 union {
  struct ipv4_header  ipv4;
  struct ipv6_header  ipv6;
 } header;
};
```

```
/* Type of protocol translation for NAT */
enum dpa_cls_hm_nat_pt_type {
 DPA_CLS_HM_NAT_PT_IPv6_TO_IPv4,
 DPA_CLS_HM_NAT_PT_IPv4_TO_IPv6
};
```

Parameters

- **nat_params** - NAT header manipulation parameters:
  - **flags** – NAT operation flags specify which fields in the packet should be updated; this is a combination of the values in the dpa_cls_hm_nat_flags enum; combine the values using the or logical operand;
  - **proto** – protocol to perform NAT for; selected from dpa_cls_nat_proto enum ;
  - **type** – selects the flavor of NAT to configure; selected from the dpa_cls_hm_nat_type enum;
  - **nat** – traditional NAT header manipulation parameters; used only when traditional NAT is selected using the type attribute;
    - **sip** - new source IP address; used only when selected using the flags attribute;
    - **dip** - new destination IP address; used only when selected using the flags attribute;
  - **nat_pt** - NAT-PT header manipulation parameters; used only when NAT-PT is selected using the type attribute;
    - **type** - specifies the protocol replacement for NAT-PT: either IPv4-to-IPv6 or IPv6-to-IPv4;
    - **header** - data for protocol header replacement:
      - **ipv4** - IPv4 header data to replace IPv6 with;
      - **ipv6** - IPv6 header data to replace IPv4 with;
  - **sport** - new L4 protocol source port number; used when selected using the flags attribute;
  - **dport** - new L4 protocol destination port number; used only when selected using the flags attribute;

NOTE

More attributes may be added to this structure when ICMP NAT support will be available.

  - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. res argument is provided), fm_pcd is irrelevant;
  - **reparse** - *TRUE* to force re-parsing of the packet after this NAT;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **l3_update_node** – a handle to a header manipulation node which may combine a local IPv4/IPv6 update header manipulation with an IP protocol replace. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed;
- **l4_update_node** - handle to the local TCP/UDP update header manipulation node. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case no L4 header updates are necessary for this NAT flow);

Description

Creates or imports a NAT type header manipulation object. If the function is successful it returns at the hmd location the descriptor of the created header manipulation object.

If the res parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the fm_pcd handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the res parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when chain_head parameter is provided as true). When working in this mode the fm_pcd handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if a FMan low level driver function call has failed.

9.9.3.2.2.1.1.2 Function: dpa_classif_set_fwd_hm

Syntax

```
int dpa_classif_set_fwd_hm(
 const struct dpa_cls_hm_fwd_params  *fwd_params,
 int       next_hmd,
 int       *hmd,
bool       chain_head
const struct dpa_cls_hm_fwd_resources *res);

struct dpa_cls_hm_fwd_params {
 enum dpa_cls_hm_out_if_type   out_if_type;
 union {
  struct dpa_cls_hm_fwd_l2_param eth;
  struct dpa_cls_hm_fwd_pppoe_param pppoe;
  struct dpa_cls_hm_fwd_ppp_param ppp;
 };
 struct dpa_cls_hm_ip_frag_params  ip_frag_params;
 void       *fm_pcd;
 bool       reparse;
};

enum dpa_cls_hm_out_if_type {
 DPA_CLS_HM_IF_TYPE_ETHERNET,
 DPA_CLS_HM_IF_TYPE_PPPoE,
 DPA_CLS_HM_IF_TYPE_PPP,
```

```
 DPA_CLS_HM_IF_TYPE_LAST_ENTRY
};

struct dpa_cls_hm_ip_frag_params {
 uint16_t     mtu;
 uint8_t     scratch_bpid;
enum dpa_cls_hm_frag_df_action df_action;
};

enum dpa_cls_hm_frag_df_action {
 DPA_CLS_HM_DF_ACTION_FRAG_ANYWAY,
 DPA_CLS_HM_DF_ACTION_DONT_FRAG,
 DPA_CLS_HM_DF_ACTION_DROP
};

struct dpa_cls_hm_fwd_l2_param {
 uint8_t    macda[ETH_ALEN];
 uint8_t    macsa[ETH_ALEN];
};

struct dpa_cls_hm_fwd_pppoe_param {
 struct dpa_cls_hm_fwd_l2_param l2;
 struct pppoe_header   pppoe_header;
};

struct dpa_cls_hm_fwd_ppp_param {
 uint16_t     ppp_pid;
};

/* IP forwarding header manipulation low level driver resources */
struct dpa_cls_hm_fwd_resources {
 void *fwd_node;
 void *pppoe_node;
 void *ip_frag_node;
};
```

Parameters

- **fwd_params** - IP forwarding header manipulation parameters:
  - **out_if_type** – output interface type; based on this selection the DPA Classifier decides which header manipulations are needed to perform forwarding; selected from dpa_cls_hm_out_if_type enum;
  - **eth** – necessary parameters for an Ethernet output interface:
    - **macda** – new Ethernet destination MAC address to update the L2 header;
    - **macsa** – new Ethernet source MAC address to update the L2 header;
  - **pppoe** – necessary parameters for a PPPoE output interface:
    - **l2** – L2 header update parameters;
    - **pppoe_header** – PPPoE header to be inserted in the packets. The PPPoE payload length field is updated automatically (you can set it to zero).
  - **ppp** – necessary parameters for a PPP output interface:
    - **ppp_pid** – PPP PID value to use in the PPP header to be inserted.
  - **ip_frag_params** – parameters related to optional IP fragmentation:

- **mtu** – interface Maximum Transfer Unit in bytes. Use zero if no IP fragmentation should be performed (disable IP fragmentation);
- **scratch_bpid** – scratch buffer pool ID. This is necessary for the IP fragmentation on FMan v2 devices only. On FMan v3 or newer devices this parameter is ignored. It is also ignored if IP fragmentation is disabled;
- **df_action** - specifies how to deal with packets with DF flag on;

NOTE

Note: IP fragmentation is not supported in this context. Please use the “update” type header manipulation instead if IP fragmentation is needed.

- **fm_pcd** – handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. res argument is provided), fm_pcd is irelevant;
- **reparse** - *TRUE* to force re-parsing of the packet headers after this forwarding header manipulation;

- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; DPA_OFFLD_DESC_NONE can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **fwd_node** - handle to the forwarding header manipulation node. In case of an Ethernet or PPPoE output interface this is a local header replace header manipulation node (for Ethernet MAC addresses). In case of a PPP output interface this is a header manipulation node which combines a protocol specific header removal (for Ethernet and VLAN tags) with a local header insert manipulation. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

NOTE

PPPoE protocol specific header manipulation is not supported yet.

- **pppoe_node** - Handle to the PPPoE specific node. This is an internal protocol specific insert PPPoE header manipulation node. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case the output interface type is not PPPoE).
- **ip_frag_node** - handle to the IP fragmentation node. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case no IP fragmentation is enabled for this IP forwarding flow).

Description

Creates or imports a forwarding type header manipulation object. This type of header manipulation can be configured to do either of the following:

- L2 header update
- L2 header update + IP fragmentation

L2 header update operation is performed according to the output port type:

- for Ethernet ports = MAC address update
- for PPPoE ports = PPPoE header insert
- for PPP ports = PPP header insert

DPA Classifier takes into account an Ethernet/IP frame to start with and, depending on the selection of output interface type, it decides what header manipulations are necessary.

IP fragmentation cannot be enabled all alone in the forwarding header manipulation context. If only IP fragmentation is needed on a packet flow (without any other header manipulation ops), the “update” type header manipulation is recommended instead of the “forwarding” type header manipulation. For additional details, please refer to `dpa_classif_set_update_hm`.

If the res parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the res parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when `chain_head` parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if a FMan low level driver function call has failed.

9.9.3.2.2.1.1.3 Function: dpa_classif_set_remove_hm

Syntax

```
int dpa_classif_set_remove_hm(
 const struct dpa_cls_hm_remove_params *remove_params,
 int       next_hmd,
 int       *hmd,
bool       chain_head,
const struct dpa_cls_hm_remove_resources *res);

struct dpa_cls_hm_remove_params {
 enum dpa_cls_hm_remove_type   type;
 struct dpa_cls_hm_custom_rm_params  custom;
 void       *fm_pcd;
 bool       reparse;
};

enum dpa_cls_hm_remove_type {
 DPA_CLS_HM_REMOVE_ETHERNET,  /* removes ETH and all QTags */
 DPA_CLS_HM_REMOVE_PPPoE,  /* removes ETH, all QTags and PPPoE */
 DPA_CLS_HM_REMOVE_PPP,
 DPA_CLS_HM_REMOVE_CUSTOM,
 DPA_CLS_HM_REMOVE_LAST_ENTRY
};

 struct dpa_cls_hm_custom_rm_params {
 uint8_t      offset;
 uint8_t      size;
```

```
};

/* Ingress remove header manipulation low level driver resources */
struct dpa_cls_hm_remove_resources {
 void *remove_node;
};
```

Parameters

- **remove_params** - ingress (remove) header manipulation parameters:
  - **type** - selects the type of the remove header manipulation operation to perform. Protocol specific header removals don't need any further parameters; selected from dpa_cls_hm_remove_type enum.

    NOTE

    Protocol specific PPPoE header removal is not yet supported by the FMan microcode.

  - **custom** - parameters for the custom remove header manipulation. If type is anything else than "custom remove", these parameters are ignored;
    - **offset** - offset in bytes, relative to the start of the packet, to start removing data from;
    - **size** - the size in bytes of the section to remove;
  - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. res argument is provided), fm_pcd is irelevant;
  - **reparse** - *TRUE* to force re-parsing of packet headers after this header remove;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;

  NOTE

  When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

  - **remove_node** - handle to either a header removal node or a protocol specific header removal node (for Ethernet and all VLAN tags). This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

Description

Creates or imports a remove type header manipulation object.

The `REMOVE_CUSTOM` header manipulation action is a raw data remove operation using an offset (relative to the start of the packet) and a size (in bytes of the area to remove) and it doesn't involve any checksum updates to the frame. The operation is not aware of any protocols that may exist in the frame. The offset and removal size restrictions that apply here are those of the FMan microcode.

If the res parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the res parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when chain_head parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if a FMan low level driver function call has failed.

9.9.3.2.2.1.1.4 Function: dpa_classif_set_insert_hm

Syntax

```
int dpa_classif_set_insert_hm(
 const struct dpa_cls_hm_insert_params *insert_params,
 int       next_hmd,
 int       *hmd,
bool       chain_head,
const struct dpa_cls_hm_insert_resources *res);

struct dpa_cls_hm_insert_params {
 enum dpa_cls_hm_insert_type   type;
 union {
  struct dpa_cls_hm_eth_ins_params eth;
  struct dpa_cls_hm_pppoe_ins_params pppoe;
  uint16_t     ppp_pid;
  struct dpa_cls_hm_custom_ins_params custom;
 };
 void       *fm_pcd;
 bool       reparse;
};

enum dpa_cls_hm_insert_type {
 DPA_CLS_HM_INSERT_ETHERNET, /* Insert Ethernet + QTags */
 DPA_CLS_HM_INSERT_PPPoE, /* Insert PPPoE, ETH and QTags */
 DPA_CLS_HM_INSERT_PPP,
 DPA_CLS_HM_INSERT_CUSTOM, /* General insert */
 DPA_CLS_HM_INSERT_LAST_ENTRY
};

/*
 * Maximum number of VLAN tags supported by the insert header manipulation
 */
#define DPA_CLS_HM_MAX_VLANs      6

/* Ethernet header insert params */
struct dpa_cls_hm_eth_ins_params {
 struct ethhdr   eth_header;
 unsigned int   num_tags;
```

```
 struct vlan_header  qtag[DPA_CLS_HM_MAX_VLANs];
};

struct dpa_cls_hm_pppoe_ins_params {
 struct dpa_cls_hm_eth_ins_params eth;
 struct pppoe_header   pppoe_header;
};

struct dpa_cls_hm_custom_ins_params {
 uint8_t    offset;
 uint8_t    size;
 const uint8_t   *data;
};

/* Egress insert header manipulation low level driver resources */
struct dpa_cls_hm_insert_resources {
 void *insert_node;
};
```

Parameters

- **insert_params** - egress insert header manipulation parameters:
  - **type** - specifies the type of insert header manipulation; selected from the `dpa_cls_hm_egress_ins_type` enum;
  - **eth** - Ethernet header insert parameters if **type** is “insert Ethernet”:
    - **eth_header** - Ethernet header to insert;
    - **num_tags** - number of VLAN tags to insert; if zero, no VLAN tags will be inserted in the packet;
    - **qtag** – relevant only if num_tags is not zero; contains an array with the data of the VLAN tags to insert;
  - **pppoe** - PPPoE header insert parameters if **type** is “insert PPPoE”;
    - **eth** - parameters of the Ethernet header to insert together with PPPoE header;
    - **pppoe_header** – PPPoE header to insert;
  - **ppp_pid** - PPP PID value to use in the PPP header if **type** is “insert PPP”;
  - **custom** - custom insert header manipulation operation parameters. These are relevant only if a custom insert header manipulation operation is selected.
    - **offset** – the offset in bytes relative to the start of the packet where the new header will be inserted;
    - **size** – the size in bytes of the new header to be inserted;
    - **data** – pointer to the buffer containing the data of the new header to be inserted.
  - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. res argument is provided), **fm_pcd** is irelevant;
  - **reparse** - *TRUE* to force re-parsing of the packet headers after this header insert;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;

- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;

> **NOTE**
> When using external resource allocation the user shall ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. The DPA Classifier cannot verify the header manipulation parameters.

- **insert_node** - handle to either an internal header insert or an internal protocol specific header insert node. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

Description

Creates or imports an insert type header manipulation object.

The `INSERT_CUSTOM` header manipulation action is a raw data insert operation using an offset (relative to the start of the packet), a size (in bytes of the area to insert) and a data pointer and it doesn't involve any checksum updates to the frame. The operation is not aware of any protocols that may exist in the frame. The offset and insert size restrictions that apply here are those of the FMan microcode.

If the **res** parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the **res** parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when `chain_head` parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.2.1.1.5 Function: dpa_classif_set_update_hm

Syntax

```
int dpa_classif_set_update_hm(
 const struct dpa_cls_hm_update_params  *update_params,
 int        next_hmd,
 int        *hmd,
bool        chain_head,
const struct dpa_cls_hm_update_resources  *res);

struct dpa_cls_hm_update_params {
 int       op_flags;
 union {
  struct ipv4_header    new_ipv4_hdr;
  struct ipv6_header    new_ipv6_hdr;
 } replace;

 union {
```

```
  struct dpa_cls_hm_l3_update_params l3;
  struct dpa_cls_hm_l4_update_params l4;
 } update;

 struct dpa_cls_hm_ip_frag_params  ip_frag_params;
 void       *fm_pcd;
 bool       reparse;
};

/* Op flags */
enum dpa_cls_hm_update_op_flags {
 DPA_CLS_HM_UPDATE_NONE    = 0,

 DPA_CLS_HM_UPDATE_IPv4_UPDATE   = 0x01,
 DPA_CLS_HM_UPDATE_IPv6_UPDATE   = 0x02,
 DPA_CLS_HM_UPDATE_UDP_TCP_UPDATE  = 0x04,

 DPA_CLS_HM_REPLACE_IPv4_BY_IPv6  = 0x08,
 DPA_CLS_HM_REPLACE_IPv6_BY_IPv4  = 0x10
};

/* Field flags */
enum dpa_cls_hm_l3_field_flags {
 DPA_CLS_HM_IP_UPDATE_IPSA   = 0x01,
 DPA_CLS_HM_IP_UPDATE_IPDA   = 0x02,
 DPA_CLS_HM_IP_UPDATE_TOS_TC   = 0x04,
 DPA_CLS_HM_IP_UPDATE_ID    = 0x08,
 DPA_CLS_HM_IP_UPDATE_TTL_HOPL_DECREMENT = 0x10
};

enum dpa_cls_hm_l4_field_flags {
 DPA_CLS_HM_L4_UPDATE_SPORT   = 0x01,
 DPA_CLS_HM_L4_UPDATE_DPORT   = 0x02,
 DPA_CLS_HM_L4_UPDATE_CALCULATE_CKSUM = 0x04
};

/* L3 protocols field update parameters */
struct dpa_cls_hm_l3_update_params {
 struct dpa_offload_ip_address  ipsa;
 struct dpa_offload_ip_address  ipda;
 uint8_t     tos_tc;
 uint16_t     initial_id;

 /* select a combination from enum dpa_cls_hm_l3_field_flags */
 int      field_flags;
};

/* L4 protocols field update parameters */
struct dpa_cls_hm_l4_update_params {
 uint16_t sport; /* new L4 source port value */
 uint16_t dport; /* new L4 destination port value */

 /* select a combination from enum dpa_cls_hm_l4_field_flags */
 int  field_flags;
};

/* Update header manipulation low level driver resources */
struct dpa_cls_hm_update_resources {
 void *update_node;
```

```
 void *ip_frag_node;
};
```

Parameters

- **update_params** - egress update header manipulation parameters:
  - **op_flags** - flags defining the header manipulation operations to perform. They are a combination of the flags defined in the dpa_cls_hm_update_op_flags enum; only one flag from each group (e.g, update group, replace group, etc) can be selected; combine the values using the or logical operand;enum;
  - **replace** - parameters for header replace operations:
    - **new_ipv4_hdr** - IPv4 header data. This header can be used either for IPv4 field updates or for IPv6 to IPv4 header replace.
    - **new_ipv6_hdr** – IPv6 header data. This header can be used either for IPv6 field updates or for IPv4 to IPv6 header replace
  - **update** – parameters for protocol specific header update operations:
    - **l3** – L3 protocol field values. This data is used for L3 protocol header updates:
      - **ipsa** – new source IP address;
      - **ipda** – new destination IP address;
      - **tos_tc** – new TOS (for IPv4) or Traffic Class (for IPv6);
      - **initial_id** – initial IPv4 ID; this is used only if op_flags selected IPv4 update;
      - **field_flags** - a combination of flags designating the header fields to replace; the available options are defined in the `dpa_cls_hm_l3_field_flags` enum; combine the values using the or logical operand;
    - **l4** – L4 protocol field values. This data is used for L4 protocol header updates:
      - **sport** – new L4 source port value;
      - **dport** – new L4 destination port value;
      - **field_flags** - a combination of flags designating the header fields to replace; the available options are defined in the `dpa_cls_hm_l4_field_flags` enum; combine the values using the or logical operand;
  - **ip_frag_params** – IP fragmentation parameters. This is an optional operation and can be disabled if op_flags is not `DPA_CLS_HM_UPDATE_NONE`.

    NOTE

    There is currently a limitation when using IP fragmentation. IP fragmentation cannot be combined with other header manipulations, hence it can be used only as single update by setting the op_flags to DPA_CLS_HM_UPDATE_NONE.

  - **fm_pcd** – handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. res argument is provided), fm_pcd is irelevant;
  - **reparse** - *TRUE* to force re-parsing of packet headers after this header update;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;

- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, he low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **update_node** - handle to a header manipulation node with different header manipulations enabled, depending on the options selected in the parameters: local IPv4/IPv6 update header manipulation, a local TCP/UDP update header manipulation or an internal IP header replace. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case no L3 or L4 field updates or header replace features are enabled for this flow);
- **ip_frag_node** - handle to the IP fragmentation node. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case no IP fragmentation is enabled for this flow).

Description

Creates or imports an update type header manipulation object.

The updates performed by this type of header manipulation are protocol specific updates. The FMan is aware of the protocols being manipulated, hence the following checksums are also updated:

Table 255. Checksum updates

op_flags	Updated checksums
IPv4_UPDATE	IPv4 header checksum
UDP_TCP_UPDATE	TCP/UDP checksum only if it is not zero
IPv6_UPDATE	no checksums updated
REPLACE_IPv4_BY_IPv6	no checksums updated
REPLACE_IPv6_BY_IPv4	IPv4 header checksum

TCP/UDP checksum calculation can also be forced using the op_flags set to **UDP_TCP_UPDATE** and the L4 field_flags enabling **CALCULATE_CKSUM** option.

If the res parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM

If the res parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when chain_head parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.2.1.1.6 Function: dpa_classif_set_vlan_hm

Syntax

```
int dpa_classif_set_vlan_hm(
const struct dpa_cls_hm_vlan_params  vlan_params,
int       next_hmd,
int       *hmd,
bool       chain_head,
const struct dpa_cls_hm_vlan_resources *res);

enum dpa_cls_hm_vlan_type {
 DPA_CLS_HM_VLAN_INGRESS,
 DPA_CLS_HM_VLAN_EGRESS,
 DPA_CLS_HM_VLAN_LAST_ENTRY
};

struct dpa_cls_hm_vlan_params {
 enum dpa_cls_hm_vlan_type    type;
 union {
  struct dpa_cls_hm_ingress_vlan_params ingress;
  struct dpa_cls_hm_egress_vlan_params egress;
 };
 void        *fm_pcd;
 bool        reparse;
};

/*
 * Maximum number of VLAN tags supported by the insert header manipulation
 */
#define DPA_CLS_HM_MAX_VLANs      6
/* Standard size of the DSCP-to-VPri mapping table */
#define DPA_CLS_HM_DSCP_TO_VLAN_TABLE_SIZE   32

struct dpa_cls_hm_egress_vlan_params {
 enum dpa_cls_hm_vlan_update_type update_op;
 unsigned int    num_tags;
 struct vlan_header   qtag[DPA_CLS_HM_MAX_VLANs];
 union {
  uint8_t vpri;
  uint8_t dscp_to_vpri[DPA_CLS_HM_DSCP_TO_VPRI_TABLE_SIZE];
 } update;
};

enum dpa_cls_hm_vlan_update_type {
 DPA_CLS_HM_VLAN_UPDATE_NONE,
 DPA_CLS_HM_VLAN_UPDATE_VPri, /* manual VPri update */
 DPA_CLS_HM_VLAN_UPDATE_VPri_BY_DSCP,
 DPA_CLS_HM_VLAN_UPDATE_LAST_ENTRY
};

enum dpa_cls_hm_vlan_count {
 DPA_CLS_HM_VLAN_CNT_NONE,
 DPA_CLS_HM_VLAN_CNT_1QTAG,      /* outer QTag */
 DPA_CLS_HM_VLAN_CNT_2QTAGS,     /* outer most 2 QTags */
 DPA_CLS_HM_VLAN_CNT_3QTAGS,     /* outer most 3 QTags */
 DPA_CLS_HM_VLAN_CNT_4QTAGS,     /* outer most 4 QTags */
 DPA_CLS_HM_VLAN_CNT_5QTAGS,     /* outer most 5 QTags */
```

```
 DPA_CLS_HM_VLAN_CNT_6QTAGS,     /* outer most 6 QTags */
 DPA_CLS_HM_VLAN_CNT_ALL_QTAGS,
 DPA_CLS_HM_VLAN_CNT_LAST_ENTRY
};

struct dpa_cls_hm_ingress_vlan_params {
 enum dpa_cls_hm_vlan_count  num_tags;
};

/* VLAN specific header manipulation low level resources */
struct dpa_cls_hm_vlan_resources {
 void *vlan_node;
};
```

Parameters

- **vlan_params** - VLAN specific header manipulation parameters:
  - **type** - selects the type of the VLAN specific header manipulation: either “ingress” or “egress”; selected from `dpa_cls_hm_vlan_type` enum;
  - **ingress** - parameters for ingress VLAN header manipulations:
    - **num_tags** – number of VLAN tags to remove;

      NOTE

      Only the num_tags=none and num_tags=all are currently supported.
  - **egress** - parameters for egress VLAN header manipulations:
    - **update_op** – the type of desired VLAN tag update operation (if any); update operations are performed only on the outer most VLAN tag; selected from the `dpa_cls_hm_vlan_update_type` enum;
    - **num_tags** – number of VLAN tags to insert; if zero, no VLAN tags will be inserted in the packet and only update operations will be performed on existent VLAN tags; the combination num_tags=0 and update_op=none (nothing to do) doesn't make sense and it is rejected;
    - **qtag** – relevant only if num_tags is not zero; contains an array with the data of the VLAN tags to insert:
    - **update_params** – VLAN tag update parameters if an update_op was selected;
      - **vpri** – new VPri field value if update_op selects manual VPri update;
      - **dscp_to_vpri** – DSCP-to-VPri mapping table to use for VPri field update if update_op selects VPri update by mapping to DSCP;
  - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. res argument is provided), fm_pcd is irrelevant;
  - **reparse** - *TRUE* to force re-parsing of packet headers after this VLAN header update;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **vlan_node** - handle to a header manipulation node with different operations depending on the selected type of VLAN specific header manipulation. In case of VLAN ingress header manipulation this is a VLAN protocol specific removal node. In case of VLAN egress header manipulation this is a header manipulation node which may combine an internal header insert (in case there are VLANs to insert) with a protocol specific VLAN update operation. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

Description

Creates or imports a VLAN specific header manipulation (either ingress or egress) object. VLAN header manipulations apply to both IPv4 as well as IPv6 frames. VLAN tags are added to the Ethernet frame header hence it will work with any type of L3 protocol (even different from IP).

Removal of VLAN tags is performed based on VLAN tag availability in the incoming packet. If the user configured to remove a specific number of VLAN tags and the incoming packets don't have that many VLAN tags, the FMan will remove as many tags as it finds up to the configured number of VLAN tags to remove. If the incoming packet is untagged, no action will be taken and the packet will remain unchanged. This is considered normal operation and no error or other notification will be issued in this case.

If the res parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the res parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when **chain_head** parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.2.1.1.7 Function: dpa_classif_set_mpls_hm

Syntax

```
int dpa_classif_set_mpls_hm(const struct dpa_cls_hm_mpls_params
*mpls_params, int next_hmd, int *hmd, bool chain_head,
const struct dpa_cls_hm_mpls_resources *res);

struct dpa_cls_hm_mpls_params {
 enum dpa_cls_hm_mpls_type  type;
 struct mpls_header  mpls_hdr[DPA_CLS_HM_MAX_MPLS_LABELS];
 unsigned int    num_labels;
 void      *fm_pcd;
 bool      reparse;
};
```

```
/*
 * Maximum number of MPLS labels supported by the insert header
 * manipulation
 */
#define DPA_CLS_HM_MAX_MPLS_LABELS    6

enum dpa_cls_hm_mpls_type {
 DPA_CLS_HM_MPLS_INSERT_LABELS,
 DPA_CLS_HM_MPLS_REMOVE_ALL_LABELS,
 DPA_CLS_HM_MPLS_LAST_ENTRY
};

/* MPLS specific header manipulation low level driver resources */
struct dpa_cls_hm_mpls_resources {
 void *ins_rm_node;
};
```

Parameters

- **mpls_params** - MPLS specific header manipulation parameters;
  - **type** - specifies the type of header manipulation; selected from dpa_cls_hm_mpls_type enum; the “remove all MPLS labels” operation type doesn’t need any other parameters;
  - **mpls_hdr** - the MPLS labels to insert if the operation type is “insert MPLS labels”;
  - **num_labels** - number of MPLS labels to insert; this is relevant only if the operation type is “insert MPLS labels”;
  - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. res argument is provided), fm_pcd is irelevant;
  - **reparse** - *TRUE* to force reparsing of packet headers after this MPLS header update;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; DPA_OFFLD_DESC_NONE can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;
  - **ins_rm_node** - handle to the protocol specific header insert (MPLS) or to the protocol specific header removal (MPLS) node. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

Description

Creates or imports a MPLS specific header manipulation object.

If the res parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the **fm_pcd** handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn’t allocate MURAM.

If the res parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when chain_head parameter is provided as true). When working in this mode the **fm_pcd** handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.2.1.1.8 Function: dpa_classif_free_hm

Syntax

```
int dpa_classif_free_hm(int hmd);
```

Parameters

- **hmd** - descriptor of the header manipulation object to destroy.

Description

Releases a header manipulation object and frees up all related resources allocated for it. The header manipulation operations can be removed in any order. The low level driver resources are removed only when the chain head operation is removed.

The function fails if the header manipulation object is in use (i.e. if it is part of a header manipulation chain which is still attached to an existing DPA Classifier entry action descriptor).

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-EBUSY`, if the header manipulation operation is still in use, or if an FMan low level driver function call has failed.

9.9.3.2.2.1.2 Modify Operations at Runtime

Table 256. API to Modify Header Manipulation Operations at Runtime

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_modify_nat_hm`	Modify the parameters of an existing NAT header manipulation.	• new NAT header manipulation parameters; • flags indicating which parameters of the NAT header manipulation must be modified.	None.

Table continues on the next page...

Table 256. API to Modify Header Manipulation Operations at Runtime (continued)

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_modify_fwd_hm`	Modify the parameters of an existing forwarding type header manipulation.	• new forwarding header manipulation parameters; • flags indicating which parameters of the forwarding header manipulation must be modified.	None.
`dpa_classif_modify_remove_hm`	Modify the parameters of an existing remove type header manipulation.	• new remove header manipulation parameters; • flags indicating which parameters of the remove header manipulation must be modified.	None.
`dpa_classif_modify_insert_hm`	Modify the parameters of an existing insert header manipulation.	• new insert header manipulation parameters; • flags indicating which parameters of the insert header manipulation must be modified.	None.
`dpa_classif_modify_update_hm`	Modify the parameters of an existing update header manipulation.	• new update header manipulation parameters; • flags indicating which parameters of the update header manipulation must be modified.	None.
`dpa_classif_modify_vlan_hm`	Modify the parameters of an existing VLAN specific header manipulation.	• new VLAN specific header manipulation parameters; • flags indicating which parameters of the VLAN specific header manipulation must be modified.	None.
`dpa_classif_modify_mpls_hm`	Modify the parameters of an existing MPLS specific header manipulation.	• new MPLS specific header manipulation parameters; • flags indicating which parameters of the MPLS specific header manipulation must be modified.	None.

9.9.3.2.2.1.2.1 Function: dpa_classif_modify_nat_hm

Syntax

```
int dpa_classif_modify_nat_hm(int    hmd,
const struct dpa_cls_hm_nat_params  *new_nat_params,
int       modify_flags);

enum dpa_cls_hm_nat_modify_flags {
 DPA_CLS_HM_NAT_MOD_FLAGS   = 0x01,
 DPA_CLS_HM_NAT_MOD_SIP    = 0x02,
 DPA_CLS_HM_NAT_MOD_DIP    = 0x04,
 DPA_CLS_HM_NAT_MOD_SPORT   = 0x08,
 DPA_CLS_HM_NAT_MOD_DPORT   = 0x10,
 DPA_CLS_HM_NAT_MOD_IP_HDR   = 0x20
};
```

Parameters

- **hmd** - descriptor of the NAT header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a NAT header manipulation, the function will fail;
- **new_nat_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the `new_nat_params` data structure are valid). Select the flag values from the `dpa_cls_hm_nat_modify_flags` enum and combine them using the "or" logical operand. There are some flag combinations which are illegal:
  - `DPA_CLS_HM_NAT_MOD_IP_HDR` cannot be used in combination with either of:
    1. `DPA_CLS_HM_NAT_MOD_SIP` or
    2. `DPA_CLS_HM_NAT_MOD_DIP`.

Description

Modify the parameters of an existing NAT header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.2.1.2.2 Function: dpa_classif_modify_fwd_hm

Syntax

```
int dpa_classif_modify_fwd_hm(int    hmd,
const struct dpa_cls_hm_fwd_params  *new_fwd_params,
int       modify_flags);

enum dpa_cls_hm_fwd_modify_flags {
 DPA_CLS_HM_FWD_MOD_ETH_MACSA   = 0x01,
 DPA_CLS_HM_FWD_MOD_ETH_MACDA   = 0x02,
 DPA_CLS_HM_FWD_MOD_PPPoE_HEADER  = 0x04,
 DPA_CLS_HM_FWD_MOD_PPP_PID   = 0x08,
```

```
 DPA_CLS_HM_FWD_MOD_IP_FRAG_MTU  = 0x10,
 DPA_CLS_HM_FWD_MOD_IP_FRAG_SCRATCH_BPID = 0x20,
 DPA_CLS_HM_FWD_MOD_IP_FRAG_DF_ACTION = 0x40
};
```

Parameters

- **hmd** - descriptor of the forwarding header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a forwarding header manipulation, the function will fail;
- **new_fwd_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the `new_fwd_params` data structure are valid). Select the flag values from the `dpa_cls_hm_fwd_modify_flags` enum and combine them using the "or" logical operand. There are some flag combinations which are illegal:
  - `DPA_CLS_HM_FWD_MOD_PPP_PID` cannot be used in combination with either of:
    - `DPA_CLS_HM_FWD_MOD_ETH_MACSA`,
    - `DPA_CLS_HM_FWD_MOD_ETH_MACDA` or
    - `DPA_CLS_HM_FWD_MOD_PPPoE_HEADER`.

Description

Modify the parameters of an existing forwarding type header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.2.1.2.3 Function: dpa_classif_modify_remove_hm

Syntax

```
int dpa_classif_modify_remove_hm(int            hmd,
const struct dpa_cls_hm_remove_params    *new_remove_params,
int                          modify_flags);

enum dpa_cls_hm_remove_modify_flags {
    DPA_CLS_HM_RM_MOD_TYPE            = 0x01,
    DPA_CLS_HM_RM_MOD_CUSTOM_OFFSET    = 0x02,
    DPA_CLS_HM_RM_MOD_CUSTOM_SIZE        = 0x04
};
```

Parameters

- **hmd** - descriptor of the remove header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a remove header manipulation, the function will fail;

- **new_remove_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the new_remove_params data structure are valid). Select the flag values from the dpa_cls_hm_remove_modify_flags enum and combine them using the "or" logical operand.

Description

Modify the parameters of an existing remove type header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.2.1.2.4 Function: dpa_classif_modify_insert_hm

Syntax

```
int dpa_classif_modify_insert_hm(int            hmd,
const struct dpa_cls_hm_insert_params    *new_insert_params,
int                           modify_flags);

enum dpa_cls_hm_insert_modify_flags {
    /* Ethernet and PPPoE insert group */
    DPA_CLS_HM_INS_MOD_ETH_HEADER        = 0x01,
    DPA_CLS_HM_INS_MOD_QTAGS_ARRAY    = 0x02,
    DPA_CLS_HM_INS_MOD_PPPoE_HEADER    = 0x04,

    /* PPP insert group */
    DPA_CLS_HM_INS_MOD_PPP_PID        = 0x08,

    /* Custom insert group */
    DPA_CLS_HM_INS_MOD_CUSTOM_OFFSET    = 0x10,
    DPA_CLS_HM_INS_MOD_CUSTOM_DATA    = 0x20
};
```

Parameters

- **hmd** - descriptor of the insert header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not an insert header manipulation, the function will fail;
- **new_insert_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the new_insert_params data structure are valid). Select the flag values from the `dpa_cls_hm_insert_modify_flags` enum and combine them using the "or" ogical operand. The user can only combine flags from a single group (either only Ethernet and PPPoE insert group flags or PPP insert group flags, and so on).

Description

Modify the parameters of an existing insert header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.2.1.2.5 Function: dpa_classif_modify_update_hm

Syntax

```
int dpa_classif_modify_update_hm(int            hmd,
const struct dpa_cls_hm_update_params    *new_update_params,
int                          modify_flags);

enum dpa_cls_hm_update_modify_flags {
    DPA_CLS_HM_UPDATE_MOD_IPHDR            = 0x0001,

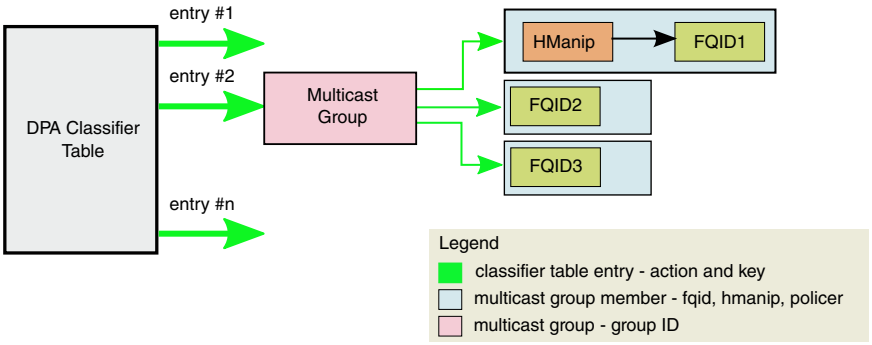
    /* L3 protocol flags group */
DPA_CLS_HM_UPDATE_MOD_SIP            = 0x0002,
    DPA_CLS_HM_UPDATE_MOD_DIP            = 0x0004,
    DPA_CLS_HM_UPDATE_MOD_TOS_TC            = 0x0008,
    DPA_CLS_HM_UPDATE_MOD_IP_ID            = 0x0010,
    DPA_CLS_HM_UPDATE_MOD_L3_FLAGS        = 0x0020,

    /* L4 protocol flags group */
    DPA_CLS_HM_UPDATE_MOD_SPORT            = 0x0040,
    DPA_CLS_HM_UPDATE_MOD_DPORT            = 0x0080,
    DPA_CLS_HM_UPDATE_MOD_L4_FLAGS        = 0x0100,

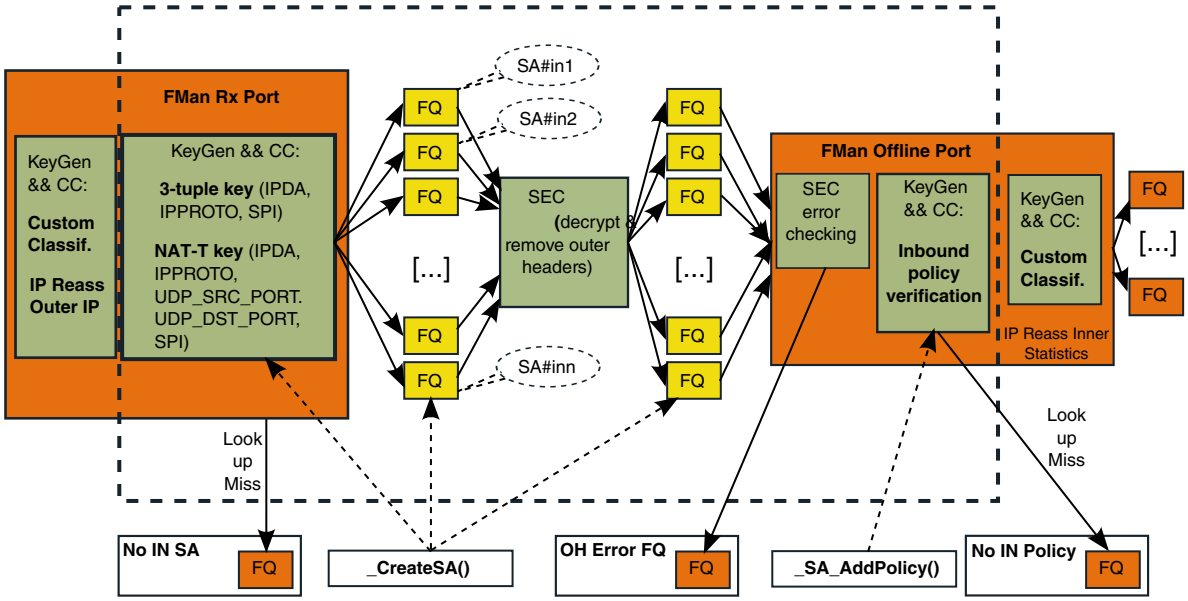
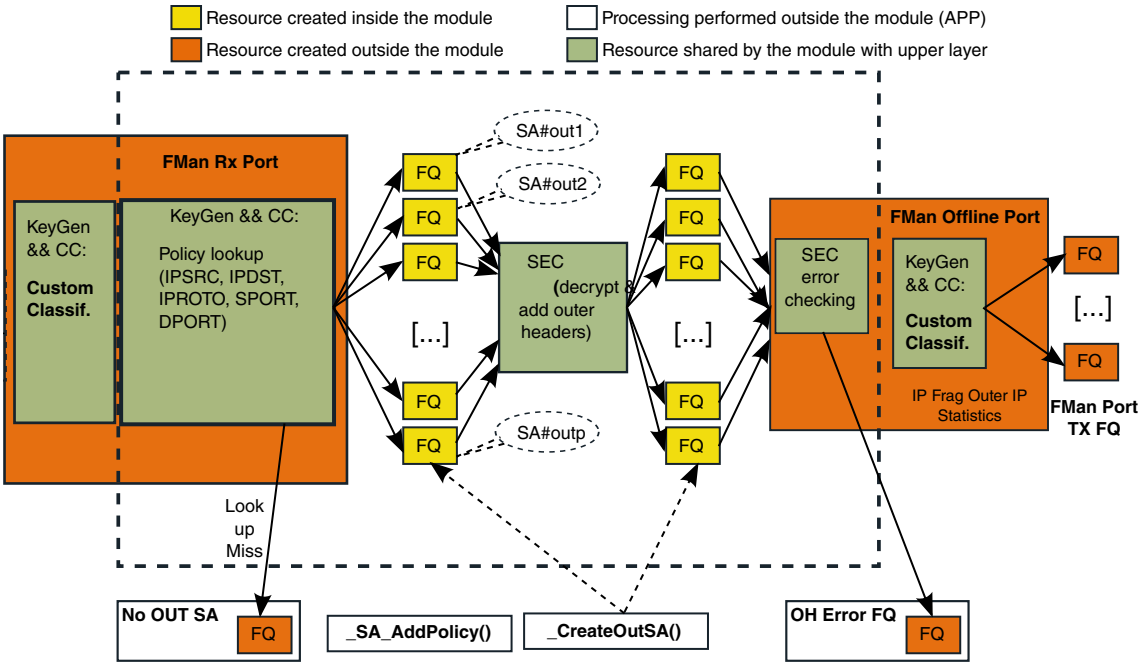
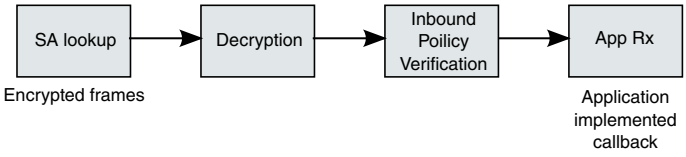
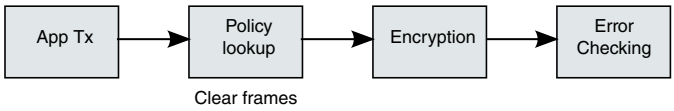
    DPA_CLS_HM_UPDATE_MOD_IP_FRAG_MTU        = 0x0200,
    DPA_CLS_HM_UPDATE_MOD_IP_FRAG_SCRATCH_BPID = 0x0400,
    DPA_CLS_HM_UPDATE_MOD_IP_FRAG_DF_ACTION    = 0x0800
};
```

Parameters

- **hmd** - descriptor of the update header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not an update header manipulation, the function will fail;
- **new_update_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the new_update_params data structure are valid). Select the flag values from the `dpa_cls_hm_update_modify_flags` enum and combine them using the "or" logical operand. Avoid combinations between flags from the L3 protocol group with flags from the L4 protocol group.

Description

Modify the parameters of an existing update header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;

- -EBUSY, if an FMan low level driver function call has failed.

9.9.3.2.2.1.2.6 Function: dpa_classif_modify_vlan_hm

Syntax

```
int dpa_classif_modify_vlan_hm(int                hmd,
const struct dpa_cls_hm_vlan_params        *new_vlan_params,
int                           modify_flags);

enum dpa_cls_hm_vlan_modify_flags {
    DPA_CLS_HM_VLAN_MOD_INGRESS_NUM_QTAGS    = 0x01,

    DPA_CLS_HM_VLAN_MOD_EGRESS_QTAGS    = 0x02,
    DPA_CLS_HM_VLAN_MOD_EGRESS_UPDATE_OP    = 0x04,
    DPA_CLS_HM_VLAN_MOD_EGRESS_VPRI        = 0x08,
    DPA_CLS_HM_VLAN_MOD_EGRESS_DSCP_TO_VPRI_ARRAY = 0x10
};
```

Parameters

- **hmd** - descriptor of the VLAN specific header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a VLAN specific header manipulation, the function will fail;
- **new_vlan_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the new_vlan_params data structure are valid). Select the flag values from the dpa_cls_hm_vlan_modify_flags enum and combine them using the "or" logical operand. The flag DPA_CLS_HM_VLAN_MOD_INGRESS_NUM_QTAGS cannot be combined with any other flags.

Description

Modify the parameters of an existing VLAN specific header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -ENOMEM, if dynamic memory allocations have failed;
- -EBUSY, if an FMan low level driver function call has failed.

9.9.3.2.2.1.2.7 Function: dpa_classif_modify_mpls_hm

Syntax

```
int dpa_classif_modify_mpls_hm(int    hmd,
const struct dpa_cls_hm_mpls_params  *new_mpls_params,
int       modify_flags);

enum dpa_cls_hm_mpls_modify_flags {
 DPA_CLS_HM_MPLS_MOD_NUM_LABELS = 0x01,
 DPA_CLS_HM_MPLS_MOD_HDR_ARRAY  = 0x02,
```

```
};
```

Parameters

- **hmd** - descriptor of the MPLS specific header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a MPLS specific header manipulation, the function will fail;
- **new_mpls_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the `new_mpls_params` data structure are valid). Select the flag values from the `dpa_cls_hm_mpls_modify_flags` enum and combine them using the or logical operand.

Description

Modify the parameters of an existing MPLS specific header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

9.9.3.2.3 Multicast

The multicast module allows creation and management of specific entries into the classifier tables, entries identified as **multicast groups**. A group has **at least one member** identified as an enqueue action. The user can define a maximum number of members inside a group. Members can also have header manipulation chains attached. Header manipulations on any member, except the last member, **must not change the frame's header protocol stack and size**. No such limitation exists on the last member from a group. A virtual storage profile Id can also be set on each member of a group.



Figure 254. Multicast example

After creating a multicast group, new members can be added by filing the `membr_params`. The newly created member is identified by the `md` descriptor. Up to 64 members can be added in a group. Another step after creating a group is to create an entry of type multicast in a classification table. When calling `dpa_classif_table_insert_entry` the group descriptor (`grpd`) will be provided as input parameter for the new entry. If an entry of type multicast is no longer needed it can be removed using the `entry_id` or the call `dpa_classif_table_delete_entry_by_key` (the entry is identified by key). Members of the multicast group can be individually removed by calling `dpa_classif_mcast_remove_member`. The group can be removed by calling `dpa_classif_mcast_free_group`.

9.9.3.2.3.1 Multicast API

There are two selections for the DPA Multicast API:

9.9.3.2.3.1.1 Create and Remove Multicast Groups

Table 257. API to Create and Remove Multicast Groups

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_mcast_create_group`	Creates a multicast group with one initial member. If external group handle is given the function imports an existing group from preallocated resources	• max number of members in multicast group; • handle of the PCD needed to create the group • enqueue parameters of the initial member. • prefilled members – number of members in the imported group.	Descriptor of the newly created multicast group.
`dpa_classif_mcast_free_group`	Removes an existing group.	• group descriptor identifying the multicast group to remove.	None.

9.9.3.2.3.1.1.1 Function: dpa_classif_mcast_create_group

Syntax

```
int dpa_classif_mcast_group_create(struct dpa_cls_mcast_group_params
       *mcast_group_params, int *grpd);

/* Multicast group parameters */
struct dpa_cls_mcast_group_params {
 uint8_t     max_members;
 void      *fm_pcd;
 struct dpa_cls_tbl_enq_action_desc first_member_params;
unsigned int    prefilled_members;
void      *group;
void      *distribution;

};
```

Parameters

- **mcast_group_params** - group parameters
  - **max_members** - maximum number of members in group.
  - **fm_pcd** - handle of the PCD needed when creating a group.
  - **member_params** – enqueue parameters for the member (fqid, policer, header manipulation descriptor, storage profile id).
    - **override_fqid** – nonzero if the action descriptor corresponding to the member is of type new classification result and zero if the action descriptor is of type keep classification result;

- **new_fqid** – if override_fqid is nonzero, this holds the explicit frame queue Id where to place the frame;
- **policer_params** – policing parameters; if NULL, no policing is performed during the enqueue operation.
  - **modify_policer_params** – nonzero if the default policer parameters will be overridden;
  - **shared_profile** - nonzero if this policer profile is shared between ports; relevant only if modify_policer_params is nonzero;
  - **new_rel_profile_id** - this parameter should indicate the policer profile offset within the port's policer profiles or from the SHARED window; relevant only if modify_policer_params is nonzero;
- **hmd** – header manipulation object descriptor, or -1 if no header manipulation is required for the enqueue action; the header manipulation object defined by the provided descriptor must be a header manipulation chain head;
- **spid** – storage profile id. This parameter will specify the storage profile, in order to allocate new external buffers for the frame descriptor received on the member.

- **prefilled_members** – Number of members that already exist in the imported group.
- **group** – External group handle given as input parameter for an import operation.
- **distribution** – External distribution handle. When provided, replicated frames are not enqueued to members' frame queues. They are sent to this distribution.

- **grpd** - Address of an integer variable in which the function will return the group descriptor. This value will be used in other function calls, to identify the group on which operations will be performed.

Description

This function creates or imports a multicast group. The new group is identified by the group descriptor - **grpd**. It can be assigned to a new entry in the classification table using the function `dpa_classif_table_insert_entry`, which will have the type `DPA_CLS_TBL_ACTION_MCAST` for action parameter. If external group handle is provided, preallocated resources will be used – and the existing group will be used to add new members to it. Prefilled members will specify the number of existing members in the imported group.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- `-EINVAL`, if `mcast_group_params` parameter is NULL;
- `-EINVAL`, if `grpd parameter` is NULL;
- `-EINVAL`, if `max_member` is 0 or has a value greater than maximum number of supported members (64);
- `-EINVAL`, if invalid header manip descriptor (`hmd`) for the member defined in the new group. (A new group has at least one member);
- `-EINVAL`, if header manipulation descriptor of the member (`hmd`) is not a chain head;
- `-EINVAL`, if the group could not be created when low level function was invoked;
- `-ENOMEM`, if no more memory for the new multicast group;
- `-ENOMEM`, if no more memory for descriptor table when allocating descriptor for the new group;
- `-ENOMEM`, if no more memory for the member index array used to assign member ids;
- `-ENOSYS`, if policing params were provided for the first group member.

9.9.3.2.3.1.1.2 Function: dpa_classif_mcast_free_group

Syntax

```
int dpa_classif_mcast_free_group(int grpd);
```

Parameters

- **grpd** - Multicast group descriptor. This value identifies a group that was created before this function call.

Description

This function removes an existing group identified by the **grpd** parameter.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- `-EINVAL`, if Invalid group descriptor grpd provided to the function;
- `-EINVAL`, if the group could not be removed when low level function was invoked.

9.9.3.2.3.1.2 Add and Remove Multicast Group Members

Table 258. API to Add and Remove Multicast Group Members

Function Name	Description	Input Parameters	Output Parameters
`dpa_classif_mcast_add_member`	Adds a new member to an existing group	• group descriptor • enqueue parameters of the new member (fqid, policer, header manip desc, spid)	Descriptor of the newly created member.
`dpa_classif_mcast_remove_member`	Removes an existing member from an existing group. The member is identified by its member descriptor.	• group descriptor • member descriptor identifying the member to be removed from the group.	None.

9.9.3.2.3.1.2.1 Function: dpa_classif_mcast_add_member

Syntax

```
int dpa_classif_mcast_add_member(int grpd,
                   struct dpa_cls_tbl_enq_action_desc member_params, int *md);
```

Parameters

- **grpd** - Multicast group descriptor. This value identifies a group that was created before this function call.
- **member_params** - enqueue parameters for the member (fqid, policer, header manipulation descriptor, storage profile.
  - **override_fqid** – nonzero if the action descriptor corresponding to the member is of type new classification result and zero if the action descriptor is of type keep classification result;
  - **new_fqid** – if override_fqid is nonzero, this holds the explicit frame queue Id where to place the frame;

- **policer_params** – policing parameters; if NULL, no policing is performed during the enqueue operation.
  - **modify_policer_params** – nonzero if the default policer parameters will be overridden;
  - **shared_profile** - nonzero if this policer profile is shared between ports; relevant only if modify_policer_params is nonzero;
  - **new_rel_profile_id** - this parameter should indicate the policer profile offset within the port's policer profiles or from the SHARED window; relevant only if modify_policer_params is nonzero;
- **hmd** – header manipulation object descriptor, or -1 if no header manipulation is required for the enqueue action; the header manipulation object defined by the provided descriptor must be a header manipulation chain head;
- **spid** – storage profile id. This parameter will specify the storage profile, in order to allocate new external buffers for the frame descriptor received on the member.
- **md** - Address of an integer variable in which the function will return the member descriptor. This is a reference to an internal index updated by the FMD driver when a member is added or removed from the group.

Description

This function adds a new member to an existing group identified by the **grpd** parameter. The new created member will be identified by the member descriptor md.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- `-EINVAL`, if an invalid group descriptor (grpd) was provided to the function;
- `-EINVAL`, if `member_params` parameter is NULL;
- `-EINVAL`, if `md` is NULL;
- `-EINVAL`, if invalid header manip descriptor (`hmd`) for the new member;
- `-EINVAL`, if header manipulation descriptor (`hmd`) of the member is not a chain head;
- `-EINVAL`, if the multicast member could not be added when low level function was invoked;
- `-ENOSPC`, if the current number of members reached maximum value max_members (defined when the group was created);
- `-ENOSYS`, if policing params were provided.

9.9.3.2.3.1.2.2 Function: dpa_classif_mcast_remove_member

Syntax

```
int dpa_classif_mcast_remove_member(int grpd, int md);
```

Parameters

- **grpd** - Multicast group descriptor. This value identifies a group that was created before this function call.
- **md** - member descriptor which identifies the member inside a group

Description

This function removes an existing member from a group identified by the **grpd** parameter. The member is identified by member descriptor.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- `-EINVAL`, if an invalid group descriptor (`grpd`) was provided to the function;
- `-EINVAL`, if an invalid member descriptor `md` (value is negative or greater than `max_members`) was provided to the function;
- `-EINVAL`, if the multicast member could not be removed when low level function was invoked.

9.9.3.3 DPA IPSec

The DPA IPSec module exports a set of functions which can be used to:

- initialize the DPA IPSec module internal data structures;
- create and configure full inbound IPSec hardware accelerated paths;
- create and configure full outbound IPSec hardware accelerated paths;
- replace expired SAs (after rekeying) without packet loss.



Figure 255. DPA IPSec inbound path architecture overview



Figure 256. DPA IPSec outbound path architecture overview

After the accelerated paths for the inbound (figure above) and the outbound (figure above) are configured, the calling application can send and receive IPSec protected packets just by enqueuing and dequeueing frames. Although, from the calling applications perspective, all paths (inbound or outbound) are configured using similar function calls, the implementation of these hardware accelerated paths differs substantially. The DPA IPSec API is responsible for hiding these implementation details.

9.9.3.3.1 Initialization

Inbound path creation

Initialization is a mandatory step for enabling IPSec offloading. The initialization functions are designed to create and initialize the internal data structures of the module, which will later be used as a base for creating inbound or outbound accelerated IPSec processing paths. These configuration tasks are performed only once and are meant to enable the required hardware blocks, apply an initial configuration and initialize features that do not require dynamic configuration.



Figure 257. Inbound processing decomposition

Creating an inbound accelerated IPSec path means configuring the hardware blocks to classify the encrypted packets, decapsulate and decrypt them, perform required policy checks on the decrypted packet (as stated in the RFCs) and finally deliver the resulting packet to the calling application. The hardware blocks that process the packets before and after decryption and decapsulation, can also be configured to perform IP reassembly. Before decryption IP reassembly acts on the encrypted packets, while after decryption it acts on the inner, decrypted packets.

Because IP reassembly can (presently) only be enabled on a per port basis in FMan, it is beyond the scope of the DPA IPSec API to provide a means to configure and enable this feature. It is the responsibility of the upper layer applications to enable and configure IP reassembly when it initializes the PCD skeleton.

Outbound path creation



Figure 258. Outbound processing decomposition

An accelerated outbound IPSec path is supposed to use the hardware blocks to speed up the processing of an IP packet from the point where it is delivered for IPSec processing by the calling application and till a correct IPSec encapsulated packet has been constructed. Different hardware blocks have to be configured to classify the supplied packet and identify the correct outbound SA, encrypt and encapsulate the packet and finally check for errors and deliver the packet to an offline port. The calling application must implement on this offline port the classification needed to forward the packet. This processing is outside the scope of the DPA IPSec module.

9.9.3.3.2 DPA IPSec API

There are several selections for the DPA IPSec API:

9.9.3.3.2.1 IPSec Engine Startup and Shutdown

Table 259. API for IPSec Engine Startup and Shutdown

Function Name	Description
dpa_ipsec_init	The model's layout is fixed so most of the required configuration parameters are held internally. The function builds the necessary infrastructure for managing future runtime operations.
dpa_ipsec_free	Releases all the resources allocated previously by this module and also the managed (run-time added) ones.

9.9.3.3.2.1.1 Function: dpa_ipsec_init

Syntax

```
int dpa_ipsec_init(const struct dpa_ipsec_params *params,
    int *dpa_ipsec_id);

#define DPA_IPSEC_KEY_FIELD_SIP  0x01
#define DPA_IPSEC_KEY_FIELD_DIP  0x02
#define DPA_IPSEC_KEY_FIELD_PROTO  0x04
#define DPA_IPSEC_KEY_FIELD_DSCP  0x08
#define DPA_IPSEC_KEY_FIELD_SPORT  0x10
#define DPA_IPSEC_KEY_FIELD_ICMP_TYPE 0x10
#define DPA_IPSEC_KEY_FIELD_DPORT  0x20
#define DPA_IPSEC_KEY_FIELD_ICMP_CODE 0x20

enum dpa_ipsec_proto {
 DPA_IPSEC_PROTO_TCP_IPV4 = 0,
DPA_IPSEC_PROTO_TCP_IPV6,
 DPA_IPSEC_PROTO_UDP_IPV4,
 DPA_IPSEC_PROTO_UDP_IPV6,
DPA_IPSEC_PROTO_ICMP_IPV4,
DPA_IPSEC_PROTO_ICMP_IPV6,
```

```
 DPA_IPSEC_PROTO_SCTP_IPV4,
 DPA_IPSEC_PROTO_SCTP_IPv6,
 DPA_IPSEC_PROTO_ANY_IPV4,
 DPA_IPSEC_PROTO_ANY_IPV6,
 DPA_IPSEC_MAX_SUPPORTED_PROTOS
};

enum dpa_ipsec_sa_type {
 DPA_IPSEC_SA_IPV4 = 0,
DPA_IPSEC_SA_IPV4_NATT,
 DPA_IPSEC_SA_IPV6,
 DPA_IPSEC_MAX_SA_TYPE
};

enum dpa_ipsec_data_off {
 DPA_IPSEC_DATA_OFF_NONE = 0,
 DPA_IPSEC_DATA_OFF_1_BURST,
 DPA_IPSEC_DATA_OFF_2_BURST,
 DPA_IPSEC_DATA_OFF_3_BURST
};

struct dpa_ipsec_pol_table {
    int  dpa_cls_td;
    uint8_t key_fields;
};


struct dpa_ipsec_pre_sec_in_params {
    int dpa_cls_td[DPA_IPSEC_MAX_SA_TYPE];
    int gw_change_en;
};

struct dpa_ipsec_pre_sec_out_params {
    struct dpa_ipsec_pol_table table[DPA_IPSEC_MAX_SUPPORTED_PROTOS];
};

struct dpa_ipsec_post_sec_in_params {
    enum dpa_ipsec_data_off data_off;
    uint16_t       tx_ch;
    int            dpa_cls_td;
    bool                      do_pol_check;
    uint8_t                   key_fields;
    bool                      use_ipv6_pol;
    uint16_t                  base_flow_id;
};

struct dpa_ipsec_post_sec_out_params {
    enum dpa_ipsec_data_off data_off;
    uint16_t       tx_ch;
};

struct dpa_ipsec_fqid_range {
    uint32_t start_fqid;
    uint32_t end_fqid;
};

struct dpa_ipsec_params {
 struct dpa_ipsec_pre_sec_in_params   pre_sec_in_params;
 struct dpa_ipsec_post_sec_in_params  post_sec_in_params;
 struct dpa_ipsec_pre_sec_out_params  pre_sec_out_params;
```

```
 struct dpa_ipsec_post_sec_out_params post_sec_out_params;
 void      *fm_pcd;
 uint16_t       qm_sec_ch;
unsigned int   max_sa_pairs;
      struct dpa_ipsec_fqid_range  *fqid_range;
      unsigned int   max_sa_manip_ops;
};
```

Parameters

- **params** - structure containg all the parameters required for initializing a DPA IPSec instance:
  - **pre_sec_in_params** – structure containing the parameters required to configure and initialize the inbound SA lookup block:
    - **dpa_cls_td[..]** – array of descriptors for DPA Classifier tables, indexed by SA type (i.e. IPv4, IPv4 with NAT-T support, IPv6 – `dpa_ipsec_sa_type` enum), which will be used for identifying the SA that should process an incoming frame; the mechanisms used by the DPA IPSec instance to choose the table in which a policy will be inserted.
    - **gw_change_en** – enable the remote gateway address change detection mechanism (currently not supported)
  - **post_sec_in_params** - structure containing the parameters required to configure and initialize the inbound policy verification block:
    - **data_off** – the data offset that will be configured in the frame descriptor of all decrypted frames; configured using the dpa_ipsec_data_off enum; please refer to function description for details;
    - **tx_ch** – the qman channel id associated to the post decryption offline port
    - **dpa_cls_td** - the DPA IPSec module expects to receive from the upper layer a table descriptor for a DPA Classifier Indexed Table object which it will use to determine on which SA a frame was decrypted; it is mandatory that the table descriptor provided here is that of an indexed table
    - **do_pol_check** – enable the inbound policy verification
    - **key_fields** – the policy key structure (which fields of the policy parameters structure should be included in the policy key); relevant only if do_pol_check is TRUE; configured using the DPA_IPSEC_KEY_FIELD_* macros;
    - **use_ipv6_pol** - activate support for IPv6 policies. Allows better MURAM management. Relevant only if do_pol_check = TRUE.
    - **base_flow_id** – the base value from which inbound SA flow ID values can be allocated incrementally;
  - **pre_sec_out_params** - structure containing the parameters required to configure and initialize the outbound policy lookup block:
    - **table[..]** – array of structures defining the parameters for the DPA Classifier tables that will be used for offloading policy lookup; this array is indexed by policy protocol type and IP header version (dpa_ipsec_proto enum); the mechanisms used by the DPA IPSec instance to choose the table in which a policy will be inserted:
    - **dpa_cls_td** – table descriptors for DPA Classifier Exact Match Table objects, which will be used to determine which SA should be used for encrypting the frame;
    - **key_fields** – defines the policy key structure (which fields of the policy parameters structure should be included in the key) used in the table identified by the `dpa_cls_td` parameter; configured using the DPA_IPSEC_KEY_FIELD_* macros;
  - **post_sec_out_params** - structure containing the parameters required to configure and initialize the outbound error checking block:
    - **data_off** – the data offset that will be configured in the frame descriptor of all encrypted frames; configured using the `dpa_ipsec_data_off` enum; please refer to function description for details;
    - **tx_ch** – the qman channel id associated to the post encryption offline port;

- **fm_pcd** - handle to the low level driver PCD
- **qm_sec_ch** – the qman channel id associated to the SEC;
- **max_sa_pairs** – the maximum number of SAs pairs supported by the DPA IPSec module;
- **fqid_range** – pointer to a structure defining the boundaries of a FQID range from which the DPA IPSec instance should allocate fqids for the frame queues it creates internally; if this parameter is NULL, then the DPA IPSec instance will try to acquire fqids using the default allocation mechanism;
- **max_sa_manip_ops** – maximum number of manipulation objects (used for variable IP header length support or DSCP / ECN propagation support) to be preallocated at initialization; this value should be incremented with the number of header manipulations per every outbound policy; if 0, the required manipulation objects will be allocated at runtime;

- **dpa_ipsec_id** - upon successful initialization, this parameter will contain the identifier of this DPA IPSec instance; this identifier must be passed to all the other runtime functions when they are called (currently not supported).

Description

This function is used to initialize a DPA IPSec instance. When this function exits all the required internal structures of the DPA IPSec instance will be initialize and the other API functions can be used to configure offloading for outbound and inbound IPSec flows.

Some of the parameters of this function are handles to DPA Classifier Table objects. These tables must be initialized before this function is called.

The DPA IPSec instance expects to receive from the upper layer one or more table descriptors for DPA Classifier Table objects which it will use to perform SA lookup. Usually a hash table is used for this purpose, because it supports a large number of entries, but an exact match table can also be used.

If not all the tables for SA lookup are necessary (e.g. no IPv6 or NAT-T support is required), then the upper layer can pass invalid descriptors for the tables that will not be used, thus saving resources.

If more than one classifier table is used for SA lookup, the `max_sa_pairs` field will represent the sum of all the SAs that can be classified on the inbound path using all of the tables identified by valid descriptors (see the fields of the `pre_sec_in_params`structure)

If DPA Classifier Hash Tables are used for SA lookup (this is recommended), the value of the `max_sa_pairs` field should be smaller than the sum of the maximum number of entries supported by each hash table (`dpa_cls_td_*`in the `pre_sec_in_params`), because a hash table does not offer a uniform distribution of entries in its buckets. Also, the size of the indexed table in the `post_sec_in_params` structure must be greater or equal to the value of `max_sa_pairs field`.

The order in which the fields of the policy parameters structure should be place in the key is given by the values of the key fields identifiers (the `DPA_IPSEC_KEY_FIELD_*` macros) used for configuring the key structure in the key_fields members of the `pre_sec_out_params` and `post_sec_in_params` structures. The field the lowest identifier value is the first in the key and the field whith the highest identifier value is the last in the key.

The data_off is important to be configured correctly, especially when further processing of the encrypted / decrypted frames is expected to take place. Some of the modules, that will process the frames after they have been encrypted / decrypted by the SEC, require that extra space be available in the data buffers, in front of the actual data (e.g. IP Frag., IP Reass.).

The SEC can be configured not to place the output data at the beginning of the output buffer, but it restricts the values of the offset to be used. The data_off parameter is represented as a multiple of the SEC burst size, the maximum burst size used by the SEC when performing memory accesses. The SEC burst size can be either 32 or 64 bytes and it is controlled by a bit in the SEC control register. The default value for the SEC burst size is 64 bytes. Based on this, the following guidelines must be considered when configuring the `data_off` parameter:

- if classification is to be performed on the frames, after the SEC finishes encrypting / decrypting them a minimum offset of 64 bytes must be configured;
- if IP fragmentation / reassembly is to be performed on the frames, after the SEC finishes encrypting / decrypting them, an offset of 192 bytes should be configured;

- in any other situation an offset of 0 should be configured or a value consistent with the requirements of other applications / modules that are known to process the frames after SEC.

The value of the `base_flow_id` is important to be set correctly when the post decryption offline port is also used to perform other classifications based on flow ID. In these cases, it is recommended that a range of values for flow ID be reserved for all other classifications not related to IPSec, starting from flow ID value 0 and the `base_flow_id` to be equal to the number of flow ID values in that range. In all other cases the value of this field should be 0.

If the gw_change_en field value is FALSE, the remote gateway address change detection mechanism will be disabled. If the value is TRUE, this mechanism will be enabled and the calling application will be notified if an event of this type occurs through the special callback provided at SA creation time. If not callback is provided and such an event occurs, the frame that generated this event will be processed according to the miss action of the inbound SA lookup table.

The max_sa_manip_ops field can be used to preallocate a number of manipulation objects that will be used later (when an SA is offloaded) for the variable IP header length, variable IP version and DSCP / ECN propagation features and thus eliminate the need to allocate MURAM memory at runtime. The preallocated objects are placed in an internally managed pool and the implementation will extract objects from this pool whenever a SA that requires one is offloaded and put an object back into the pool whenever a SA that uses one is removed.

When calculating the value of this field the following aspects should be taken into consideration:

- for each outbound policy with header manipulation enabled one such manipulation object is required;
- for each inbound SA with or without the DSCP / ECN propagation or IP header length or variable IP version feature enabled one such manipulation object is required;
- for each inbound SA with the varianle IP header length feature enabled one such manipulation objet is required;
- for an inbound SA that has enabled both the DSCP / ECN propagation and the variable IP header length features only one manipulation object is required.

If the value of this field is 0 and one of those features is enabled, the IPSec implementation will allocate the required manipulation objects at runtime.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- `-EPERM`, , if a DPA IPSec instance is already initialized (multiple DPA IPSec instances are not supported yet) or the circular queue for inbound flow ID management could not be initialized;
- `-EINVAL`, if invalid function arguments or DPA IPSec initialization parameters were provided;
- `-ENOMEM`, if memory could not be allocated for one of the internal structures;
- `-ENOMEM`, if the internal ID management queues could not be initialized;
- `-EFAULT`, if a bad argument was passed to a internal function;
- `-EDOM`, if the size of the internal ID management queues was exceeded while trying to populate them;
- `-EBUSY`, , if the creation of the CCNodes required for inbound policy verification has failed (as a result of an error in FMD function call);
- all the error codes returned by the `dpa_classifier_create_table` function, which is used for creating tables for inbound policy verification;
- `-ENOSPC`, if the deferred work workqueue used in the rekeying process could not be initialized.

9.9.3.3.2.1.2 Function: dpa_ipsec_free

Syntax

```
int dpa_ipsec_free(int dpa_ipsec_id);
```

Parameters

- **dpa_ipsec_id** - identifier of the DPA IPSec instance that should be freed - (currently not supported).

Description

Releases all the resources allocated previously for this instance of the DPA IPsec API.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- all the error codes returned by the `dpa_ipsec_flush_all_sa function`.

9.9.3.3.2.2 Create, Remove and Manage SAs

Table 260. API for Creating, Removing and Managing SAs

Function Name	Description
`dpa_ipsec_create_sa`	Prepares the accelerated path for IPSec flows that will use this SA to encrypt or decrypt packets. Specific packets will be directed through this flow using policies (outbound) or specific SA selectors (inbound)
`dpa_ipsec_remove_sa`	Releases resources used by an SA.
`dpa_ipsec_flush_all_sa`	Removes all offloaded SAs from the DPA IPsec module
`dpa_ipsec_disable_sa`	Disables an SA before removal. No other packets will be processed using this SA. Resources associated to this SA will not be freed.
`dpa_ipsec_sa_modify`	Modifies all or just a part of the parameters of a previously offloaded SA.
`dpa_ipsec_sa_get_stats`	Returns statistics for a specified SA.
`dpa_ipsec_get_sa_context`	Retrieves information about a SA based on the packet that was processed by that SA.
`dpa_ipsec_sa_get_out_path`	Retrieves information which can be used to apply IPSec processing to a packet without performing the outbound policy lookup.

9.9.3.3.2.2.1 Function: dpa_ipsec_create_sa

Syntax

```
int dpa_ipsec_create_sa(int dpa_ipsec_id,
struct dpa_ipsec_sa_params *sa_params,
int *sa_id);

#define DPA_IPSEC_HDR_COPY_TOS 0x01
#define DPA_IPSEC_HDR_COPY_DF  0x02
#define DPA_IPSEC_HDR_DEC_TTL  0x04
#define DPA_IPSEC_HDR_COPY_DSCP 0x08
#define DPA_IPSEC_HDR_COPY_ECN 0x10

enum dpa_ipsec_direction {
 DPA_IPSEC_INBOUND = 0,     /* Inbound  */
 DPA_IPSEC_OUTBOUND  /* Outbound */
};

struct dpa_ipsec_init_vector {
    uint8_t   *init_vector;
    uint8_t   length;
};

enum dpa_ipsec_arw {
      DPA_IPSEC_ARSNONE = 0, /* No Anti Replay Protection     */
      DPA_IPSEC_ARS32 = 1, /* 32 bit Anti Replay Window size */
      DPA_IPSEC_ARS64 = 3 /* 64 bit Anti Replay Window size */
};

enum dpa_ipsec_cipher_alg {
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_96_MD5_128  ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_96_SHA_160  ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_MD5_128     ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_SHA_160     ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_SHA_256_128 ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_SHA_384_192 ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_SHA_512_256 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_96_MD5_128  ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_96_SHA_160  ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_AES_XCBC_MAC_96  ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_MD5_128     ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_SHA_160     ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_SHA_256_128 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_SHA_384_192 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_SHA_512_256 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_96_MD5_128   ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_96_SHA_160   ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_AES_XCBC_MAC_96   ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_MD5_128      ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_SHA_160      ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_SHA_256_128  ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_SHA_384_192  ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_SHA_512_256  ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_96_MD5_128   ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_96_SHA_160   ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_AES_XCBC_MAC_96   ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_MD5_128      ,
```

```
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_SHA_160      ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_SHA_256_128  ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_SHA_384_192  ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_SHA_512_256
};

struct dpa_ipsec_sa_crypto_params {
enum dpa_ipsec_cipher_alg alg_suite;
    uint8_t          *cipher_key;
    uint8_t          cipher_key_len;
    uint8_t          *auth_key;
    uint8_t          auth_key_len;
};

typedef int (*dpa_ipsec_gw_change_cb)
(int dpa_ipsec_id,
                  int sa_id,
                  struct dpa_ipsec_ip_address new_src_ip);

enum dpa_ipsec_sa_mode {
 DPA_IPSEC_SA_MODE_TUNNEL = 0,
 DPA_IPSEC_SA_MODE_TRANSPORT
};

enum dpa_ipsec_sa_proto {
 DPA_IPSEC_SA_PROTO_ESP = 0,
 DPA_IPSEC_SA_PROTO_AH
};

struct dpa_ipsec_sa_out_params {
    struct dpa_ipsec_init_vector  *init_vector;
    unsigned int        ip_ver;
    uint16_t                      ip_hdr_size;
    uint8_t                       *outer_ip_header;
    uint8_t                       *outer_udp_header;
    uint16_t           post_sec_flow_id;
    uint8_t                       dscp_start;
    uint8_t                       dscp_end;
};

struct dpa_ipsec_sa_in_params {
    enum dpa_ipsec_arw          arw;
    bool      use_var_iphdr_len;
    struct dpa_offload_ip_address  src_addr;
    struct dpa_offload_ip_address  dest_addr;
    bool           use_udp_encap;
    uint16_t      src_port;
    uint16_t      dest_port;
    struct dpa_cls_tbl_action     policy_miss_action;
    struct dpa_cls_tbl_action     post_ipsec_action;
    dpa_ipsec_gw_change_cb    *gw_change_cb;
}

struct dpa_ipsec_sa_params {
    uint32_t      spi;
    bool           use_ext_seq_num;
    uint64_t      start_seq_num;
    unsigned int      l2_hdr_size;
    enum dpa_ipsec_sa_mode  sa_mode;
    enum dpa_ipsec_sa_proto   sa_proto;
```

```
    uint8_t       hdr_upd_flags;
    uint8_t       sa_wqid;
    uint8_t      sa_bpid;
    uint16_t      sa_bufsize;
    bool    enable_stats;
    bool    enable_extended_stats;
    struct dpa_ipsec_sa_crypto_params     crypto_params;
    enum dpa_ipsec_direction              sa_dir;
    union {
        struct dpa_ipsec_sa_in_params  sa_in_params;
        struct dpa_ipsec_sa_out_params sa_out_params;
    };
};
```

Parameters

- **dpa_ipsec_id** - identifier of the DPA IPSec instance - (**currently not supported**)
- **sa_params** - structure holding the parameters for this SA:
  - **spi** – the IPSec security parameter index for this SA
  - **use_ext_seq_num** – specify whether extended sequence numbers should be used
  - **start_seq_num** – the current sequence number for this SA; this value will be automatically be incremented for each packet processed using this SA
  - **l2_hdr_size** – the size of the Ethernet header, including any VLAN information
  - **sa_mode** – transport or tunnel mode; configured using `dpa_ipsec_sa_mode enum` - (currently not supported)
  - **sa_proto** – IPSec protocol used for protecting data (ESP or AH); configured using `dpa_ipsec_sa_proto enum` - (currently not supported)
  - **hdr_upd_flags** – a set of flags describing what header fields should be propagated from the inner header to the outer header and vice versa; these flags can be used to control the copying of the entire TOS field or the DSCP bits or the ECN bits from inner to outer header and vice versa, the copying of the DF bit (only) from the inner header to the outer header and the automatic update (decrement) of the TTL in the outer / inner header; configured using the `DPA_IPSEC_HDR_*` macros;
  - **sa_wqid** - work queue ID for all the queues of this SA
  - **sa_bpid** – the buffer pool from which the SEC should acquire buffers for encrypted / decrypted frames
  - **sa_bufsize** – SEC output frames buffer pool buffer size
  - **enable_stats** – enable support for gathering traffic statistics on this SA
  - **enable_extended_stats** - enable input traffic (extended) statistics on this SA. If *enable_stats* is set to *false*, this setting is ignored.
  - **crypto_params** – protection used by this SA
    - **alg_suite** – identifier for the encryption and authentication algoritms used by this SA; configured using the `dpa_ipsec_cipher_alg enum`;
    - **cipher_key** – address of the IPSec cipher key
    - **cipher_key_len** - length in bytes of the cipher key
    - **auth_key** – address of the IPSec authentication key
    - **auth_key_len** – length in bytes of the authentication key
  - **sa_dir** – SA type (inbound / outbound); configured using the `dpa_ipsec_direction` enum;

- **sa_in_params** – parameters specific for inbound SAs:
  - **arw** - type of anti replay protection requested; configured using the `dpa_ipsec_arw` enum;
  - **use_var_iphdr_len** – enable support for variable IP header length
  - **src_addr** – source IP address in the outer header
  - **dest_addr** – destination IP address in the outer header
  - **use_udp_encap** – specify whether this is a NAT-T SA;
  - **src_port** – source udp port value in the UDP encapsulation header; relevant only if useUdpEncap is true
  - **dest_port** – destination udp port value in the UDP encapsulation header; relevant only if useUdpEncap is true
  - **policy_miss_action** – action descriptor specyfing the action that will be performed for frames that fail inbound policy verification; relevant only if inbound policy verification is enabled;
  - **post_ipsec_action** – action descriptor specifying the default action for frames that have passed inbound policy verification; if inbound policy verification is disabled this is the default action for decrypted frames on this SA
  - **gw_change_cb** – pointer to a callback used by the DPA IPSec module to signal to the upper layer that a remote gateway change event has occurred on this SA - (currently not supported)
- **sa_out_params** – parameters specific for outbound SAs:
  - **init_vector** – address of the initialization vector to be used; if this parameter is NULL the internal random number generator of the SEC will be used to generate an IV
  - **ip_ver** – type of addresses in the outer header (IPv4 or IPv6); configured using the version number (4 or 6);
  - **ip_hdr_size** – size of the outer IP header that will be used to encapsulate the encrypted frames; this must include all IP options
  - **outer_ip_header** – pointer to the buffer containing the IP encapsulation header
  - **outer_udp_header**- pointer to the buffer containing the UDP encapsulation header, used for enabling NAT-T; if this pointer is NULL UDP encapsulation will not be used
  - **post_sec_flow_id** – this value will be attached to this SAs frame queue that comes from the SEC; the upper layer can configure classification based on this value on the post encryption offline port.
  - **dscp_start** – this value represents DSCP range start value. Will be ignored if the DSCP selector is not enabled for this SA
  - **dscp_end** – this value represents DSCP range end value. Will be ignored if the DSCP selector is not enabled for this SA
- **sa_id** - address of an integer variable in which the function will return the SA identifier if it succeeds in offloading the SA; this identifier will be used in further calls to DPA IPSec functions to refer to this particular SA.

Description

This function performs all the necessary steps required to set up a pair of frame queues that can be used to encrypt or decrypt frames according to the parameters of the SA.

For outbound SAs, the calling application must also call the `dpa_ipsec_sa_add_policy` function in order to have a fully functional IPSec hardware accelerated path.

For inbound SAs the function is also responsible for adding an entry in the pre decryption table in order to identify frames that should be decrypted using this SA and direct them to the correct frame queue. A call to this function is sufficient to create a functional inbound hardware accelerated IPSec path, if the use of inbound policy verification is not desired.

In case of error the implementation of this function will attempt to automatically rollback any changes it has performed and free all resources. If this attempt is successful the sa_id parameter will contain an invalid ID (e.g. the value -1). If this attempt fails then the `sa_id` parameter will contain the actual ID which was reserved for this SA and it is the responsibility of the

calling application to call dpa_ipsec_remove_sa with this ID, in order to release any resources that were allocated by dpa_ipsec_create_sa, but could not be released during the automatic rollback process.

In case the SA per DSCP feature needs to be enabled for an outbound SA, dscp_start and dscp_end should be specified. The values will be used to configure each selector that will have the feature enabled.

When configuring the automatic update of inner header fields or outer header fields, functionalities enabled by setting different bits in the hdr_upd_flags field, the following restrictions must be taken into consideration:

- The copying of the entire TOS / DiffServ fields, the automatic decrement of the TTL filed and the copying of the DF bit , enabled by setting the DPA_IPSEC_HDR_COPY_TOS, the DPA_IPSEC_HDR_DEC_TTL and the DPA_IPSEC_HDR_COPY_DF flags, in the hdr_upd_flags field, can be performed only when using the same type of IP header for the inner and outer headers (i.e. IPv4 and IPv4 or IPv6 and IPv6)
- Propagation of the DSCP and / or ECN fields cannot be configured together with the copying of the TOS / DiffServ fields, but this option is available regardless of the IP version of inner and outer.headers.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- -EPERM, if no DPA IPSec instance is initialized;
- -EINVAL, if:
  - invalid function arguments or SA initialization parameters were provided;
  - the calling application requested activation of NAT-T support for an IPv6 SA;
  - the maximum number of inbound SAs that can be offloaded was reached;
  - a bad argument was passed to a internal function;
  - authentication and encryption keys could not be mapped to a job ring device;
  - the size of the calculated key for SA lookup is greater than the maximum key size of the classifier table in which it should be placed;
- -EDOM, if:
  - the maximum number of SAs that can be offloaded was reached;
  - an unused flow ID value could not be retrieved from the internal pool;
- -ERANGE, a new FQID could not be allocated using the default fqid allocation mechanism;
- -ENODEV, if no job ring device could be retrieved for generating the authentication split key;
- -EFAULT, if a bad argument was passed to a internal function;
- -EAGAIN, if the FMD manipulation object required for implementing variable header length support, could not be initialized;
- -ENOMEM, if:
  - there are no more available classifier tables for implementing inbound policy verification;
  - an error occurred during preparation for split key generation;
  - a new FQID could not be allocated from the fqid pool;
- error codes returned by caam_jr_enqueue function;
- error codes returned by the qman_create_fq and qman_init_fq functions;
- error codes returned by the dpa_classif_table_insert_entry (used for inserting inbound SA lookup keys, linking indexed table and inbound policy verification table and configuring the default inbound SA action), dpa_classif_table_get_params and dpa_classif_table_modify_miss_action functions.

9.9.3.3.2.2.2 Function: dpa_ipsec_remove_sa

Syntax

```
int dpa_ipsec_sa_remove(int sa_id);
```

Parameters

- **sa_id** - id of the SA that is to be deleted, as returned by the dpa_ipsec_sa_create function.

Description

This function destroys all the objects associated with the SA including all the policies. In case of error, the calling application can reuse the ID to try again to destroy this SA and all associated resources, by calling the function multiple times. It is the responsibility of the calling application to decide when it should give up on trying to release this SA's resources (if all previous calls to this function have failed).

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EPERM`, if no DPA IPSec instance is initialized;
- `-EINVAL`, if invalid function arguments were provided;
- `-EAGAIN`, if:
  - a lock on the SA could not be acquired;
  - a lock on the SA's child could not be acquired;
  - one of the policies attached to the SA could not be removed;
- `-EINPROGRESS`, if the SA is a child in the rekeying process;
- `-ENOTRECOVERABLE`, if the SA lookup key cannot be removed from the classifier table;
- `-EFAULT`, if a bad argument was passed to a internal function;
- `-EDOM`, if an error occurred while trying to release the SA's ID;
- `-ETIME`, if the SA is in the rekeying process and a timeout occurred while waiting for the SA's FQ TO SEC to be drained;
- `-EIO` , if, the SA is in the rekeying process and the child;s SA FQs could not be scheduled;
- `-EUCLEAN` , if, the SA is in the rekeying process and its resources could not be freed (FQs);
- `-EDQUOT` , if, the SA is in the rekeying process and its resources could not be recycled;
- `-EBUSY`, if a timeout occurred while waiting for the SA's FQs to be drained;
- error codes returned by `dpa_classif_table_delete_entry_by_ref`, if an error occurred during the inbound SA lookup table update process or an error occurred while trying to unlink the policy verification table form the indexed table;
- error codes returned by the `dpa_ipsec_sa_flush_policies`, if an error occurred while trying to remove the policies associated with this SA;
- error codes returned by the `qman_retire_fq` and `qman_oos_fq` functions.

9.9.3.3.2.2.3 Function: dpa_ipsec_flush_all_sa

Syntax

```
int    dpa_ipsec_flush_all_sa(int dpa_ipsec_id);
```

Parameters

- **dpa_ipsec_id** - identifier of the DPA IPSec instance.

Description

This function is used to remove all the SAs offloaded in the specified DPA IPSec instance. The internal resources are recycled. In case of error, the calling application can call this function multiple times until it is successful or it decides to give up on trying to release the resources associated to the SAs that have not been already destroyed by previous calls.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EPERM`, if no DPA IPSec instance is initialized.
- `-EAGAIN`, if one or more SAs could not be removed.

9.9.3.3.2.2.4 Function: dpa_ipsec_disable_sa

Syntax

```
int    dpa_ipsec_sa_disable(int sa_id);
```

Parameters

- **sa_id** - id of the SA that is to be disabled, as returned by the `dpa_ipsec_sa_create` function

Description

This function prepares a SA for removal. After this function returns the SA can no longer be used, because the hardware components were reconfigured in order to deactivate packet processing using this SA.

The frames that have already been classified and placed in the DPA IPSec internal queues, awainting processing, will not be dropped and will continue to be processed in the order they arrived until the queues become empty.

The resources associated with this SA, including the internal queues must be, afterwards, destroyed by the calling application using the `dpa_ipsec_remove_sa` function.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided
- `-EAGAIN`, if, a lock on the SA could not be acquired
- `-EINPROGRESS`, if the SA is currently in the rekeying process
- `-ENOTRECOVERABLE`, if the SA lookup key cannot be removed from the classifier table.
- `-EBADSLT`, if not all the policies associated to an outbound SA could be removed.

9.9.3.3.2.2.5 Function: dpa_ipsec_sa_modify

Syntax

```
int dpa_ipsec_sa_modify(int sa_id,
  struct dpa_ipsec_sa_modify_prm *modify_prm);

enum dpa_ipsec_sa_modify_type {
 DPA_IPSEC_SA_MODIFY_ARS = 0,
DPA_IPSEC_SA_MODIFY_SEQ_NUM,
 DPA_IPSEC_SA_MODIFY_EXT_SEQ_NUM,
DPA_IPSEC_SA_MODIFY_CRYPTO
};

struct dpa_ipsec_sa_modify_prm {
 enum dpa_ipsec_sa_modify_type   type;
 union {
  enum dpa_ipsec_arw    arw;
  uint64_t      seq_num;
  struct dpa_ipsec_sa_crypto_params  crypto_params;
 };
};

enum dpa_ipsec_sa_operation_code {
    DPA_IPSEC_SA_MODIFY_ARS_DONE = 0,
    DPA_IPSEC_SA_MODIFY_SEQ_NUM_DONE,
    DPA_IPSEC_SA_MODIFY_EXT_SEQ_NUM_DONE,
    DPA_IPSEC_SA_MODIFY_CRYPTO_DONE,
    DPA_IPSEC_SA_GET_SEQ_NUM_DONE
};
```

Parameters

- **sa_id** - identifier of the SA for which to modify parameters.
- **modify_prm** - used to define what changes must be done to this SA.
  - **type** – type of parameters; configured using the `dpa_ipsec_sa_modify_prm enum`; currently we support only the `DPA_IPSEC_SA_MODIFY_ARS`.
  - **arw** – anti replay type to change to; relevant only if type is `DPA_IPSEC_SA_MODIFY_ARS`
  - **seq_num** – 32 bit or extended sequence number depending on how the SA was created by dpa_ipsec_create_sa. Only the least significant word is used for 32 bit SEQ
  - **crypto_params** – change the crypto parameters for this SA to the ones specified by this structure; relevant only if type is `DPA_IPSEC_SA_MODIFY_CRYPTO`

Description

This function can be used to modify the parameters of an already offloaded SA while running.

This function is asynchronous basically when returning with error code 0 means that the operation was triggered but not actually finished. When it is finished the post SEC offline port will enqueue to its error frame queue a frame descriptor having user error status (`0x38402102`) and in the payload the SA id of the security association that has been modified allong with the modify operation code. This status is set by the SEC engine when the modify operation is finished and being a user error no other SEC errors have this status.

The upper layer should periodically check the frames from post SEC offline port error frame queue, searching for frames that have SEC user error status (`0x38402102`). Basically when a frame with this status is received for sure the modify operation was finished. Another check can be done by calling again the `dpa_ipsec_sa_modify` function with the SA id set in the frame descriptor's payload and with the same `modify_prm` parameter. If all when well the function should return an –EALREADY error code signifying that the SA has been changed to the specified parameters.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided:
  - invalid function arguments were provided
  - invalid type for modify parameters
  - SA id does not refer to an inbound SA
- `-ENXIO`, could not dma map an addres used by SEC engine.
- `-ETXTBSY`, Failed to enqueue frames to the 'TO SEC FQ' of the SA represented by SA id.
- `-EBUSY`, could not acquire the lock for SA structure represented by the SA id.
- `-EALREADY`, the proposed change is already in place for this SA.
- `-EOPNOTSUPP`, Operation not supported, currently we support changes only of DPA_IPSEC_SA_MODIFY_ARS type.

9.9.3.3.2.2.6 Function: dpa_ipsec_sa_get_stats

Syntax

```
int dpa_ipsec_sa_get_stats(int sa_id, struct dpa_ipsec_sa_stats *sa_stats);

struct dpa_ipsec_sa_stats {
 uint32_t packets_count;
 uint32_t bytes_count;
 uint32_t input_packets;
};
```

Parameters

- **sa_id** - identifier of the SA for which to remove all policies.
- **sa_stats** - pointer to a structure allocated especially to receive the requested statistics:
  - **packets_count** – number of packets processed through this SA
  - **bytes_count** – total number of bytes in all packets that were processed by this SA
  - **input_packets** - the total number of packets (including bad frames) that entered this SA. This statistic information is available only if you have set the *enable_extended_stats* attribute of the SA to *true*. Otherwise this is reported as zero. For an inbound SA this counter translates into the number of packets that were received by this SA, including frames that were later discarded due to decryption errors. For an outbound SA this translates into the total number of packets that were sent on this SA, including packets that failed due to encryption errors.

Description

In case of success the `sa_stats` will be filled with valid statistics information. In order to be able to retrieve statistical information using this function, the `enable_stats flag` must be set when the SA is created. Otherwise statistical information will not be available for the SA and this function will always return an error.

Return Value

The function returns 0 on success or an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided.
- `-EBUSY`, if a lock on the SA could not be acquired.
- `-EPERM`, if the SA was created without statistics support enabled.

9.9.3.3.2.2.7 Function: dpa_ipsec_get_sa_context

Syntax

TBD

Parameters

TBD

Description

This function will be used to retrieve information about the SA that was used to process a packet.

This information can be useful when an error has occurred during IPSec processing and the control application receives the affected packet in a offline port's error frame queue. In this case the control application needs to identify the SA which was used to process that packet.

The function can also be called when inbound policy verification is performed outside the DPA IPSec module (e.g. in software). The module that will perform the inbound policy verification will use the function to retrieve information about what SA was used for decrypting the packet.

Return Value

TBD

9.9.3.3.2.2.8 Function: dpa_ipsec_sa_get_out_path

Syntax

```
int dpa_ipsec_sa_get_out_path(int sa_id, uint32_t *fqid);
```

Parameters

- **sa_id** - identifier of the SA for which to retrieve the frame queue.
- **fqid** - identifier of the frame queue to SEC for the specified SA in order to bypass outbound policy verification and apply IPSec processing.

Description

This function will be used to retrieve information which will allow the upper layer to bypass outbound policy lookup and directly apply IPSec processing on a packet using the SA provided as a parameter to the function. In this case it is the responsibility of the calling application to ensure that the correct SA is used to process the packets.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided.
- `-EBUSY`, if, a lock on the SA could not be acquired.
- `-ENODEV`, if the SA is not used.
- `-EPERM`, if the SA is not an outbound SA.

9.9.3.3.2.2.9 Function: dpa_ipsec_get_stats

Syntax

```
int dpa_ipsec_get_stats(int dpa_ipsec_id, struct dpa_ipsec_stats *stats);

struct dpa_ipsec_stats {
 uint32_t inbound_miss_pkts;
 uint32_t inbound_miss_bytes;
 uint32_t outbound_miss_pkts;
 uint32_t outbound_miss_bytes;
};
```

Parameters

- **dpa_ipsec_id** - *dpa_ipsec* instance Id returned by a previous call to *dpa_ipsec_init*
- **stats** - pointer to a structure allocated especially to receive the global IPSec statistics
  - **inbound_miss_pkts** – number of packets that missed the inbound SA lookup
  - **inbound_miss_bytes** – total number of bytes in all packets that missed the inbound SA lookup
  - **outbound_miss_pkts** - number of packets that missed the outbound policy lookup
  - **outbound_miss_bytes** - total number of bytes in all packets that missed the outbound policy lookup

Description

This function returns the IPSec global statistics for the specified IPSec instance. In case of success the `stats` will be populated with valid statistics information.

Return Value

The function returns 0 on success or an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided.

9.9.3.3.2.3 Add and Remove Policies

Table 261. API to Add and Remove Policies

Function Name	Description
dpa_ipsec_sa_add_policy	Adds a new policy (for inbound or outbound) policy verification. It adds keys and directs packets through a certain flow (SA).
dpa_ipsec_sa_remove_policy	Removes a previously added policy.
dpa_ipsec_sa_flush_policies	Removes all policies associated to a SA.
dpa_ipsec_sa_get_policies	Retrieves all the policies linked to a specified SA.

9.9.3.3.2.3.1 Function: dpa_ipsec_sa_add_policy

Syntax

```
   int dpa_ipsec_sa_add_policy(int sa_id,
                    struct dpa_ipsec_policy_params *policy_params);

struct dpa_ipsec_l4_params {
    uint16_t       src_port;
    uint16_t       src_port_mask;
    uint16_t       dest_port;
    uint16_t       dest_port_mask;
};

struct dpa_ipsec_icmp_params {
    uint8_t      icmp_type;
    uint8_t      icmp_type_mask;
    uint8_t      icmp_code;
    uint8_t      icmp_code_mask;
};

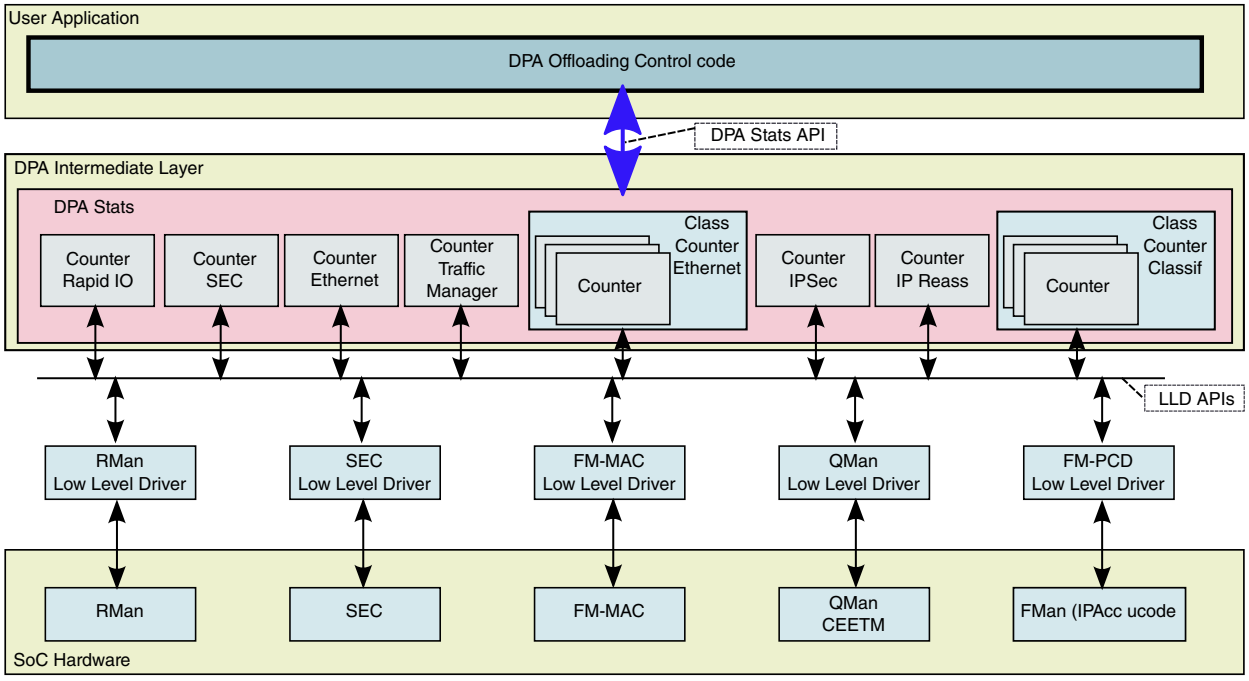
enum dpa_ipsec_df_action {
 DPA_IPSEC_DF_ACTION_DISCARD = 0,
DPA_IPSEC_DF_ACTION_OVERRIDE,
 DPA_IPSEC_DF_ACTION_CONTINUE
};

enum dpa_ipsec_pol_dir_params_type {
 DPA_IPSEC_POL_DIR_PARAMS_NONE = 0,
 DPA_IPSEC_POL_DIR_PARAMS_MANIP,
DPA_IPSEC_POL_DIR_PARAMS_ACT,
};


struct dpa_ipsec_pol_dir_params {
 enum dpa_ipsec_pol_dir_params_type  type;
 union {
  struct     int manip_desc;
  struct dpa_cls_tbl_action  in_action;
 };
```

```
};

struct dpa_ipsec_policy_params {
    struct dpa_ipsec_ip_address    src_addr;
    uint8_t      src_prefix_len;
    struct dpa_ipsec_ip_address    dest_addr;
    uint8_t        dest_prefix_len;
    uint8_t        protocol;
    bool      masked_proto;
    bool      use_dscp;
    union {
      struct dpa_ipsec_l4_params  l4;
  struct dpa_ipsec_icmp_params icmp;
    };
    struct dpa_ipsec_pol_dir_params dir params;
    int      priority;
};
```

Parameters

- **sa_id** - id of the SA that this policy is linked
- **policy_params** - structure containing the selectors for the offloaded policy:
  - **src_addr** – source IP address
  - **src_prefix_len** - the length of the mask to be applied on the source IP address
  - **dest_addr** – destination IP address
  - **dest_prefix_len** - the length of the mask to be applied on the destination IP address
  - **protocol** - the expected value in the IP header protocol field
  - **masked_proto** – enable / disable masking of the protocol field in the policy key;
  - **use_dscp** – enable / disable DSCP value in policy selector;
  - **l4** – structure containing parameters for layer 4 type of policies:
    - **src_port** – source port value; relevant only for selectors for L4 protocols
    - **src_port_mask** – mask to be applied on the source port value;
    - **dest_port** – destination port value; relevant only for selectors for L4 protocols
    - **dest_port_mask** – mask to be applied on the destination port value;
  - **icmp** – structure containing parameters for icmp policies:
    - **icmp_type** – the type field in the ICMP header; relevant only if protoco field is of type ICMP
    - **icmp_type_mask** – mask to be applied on the ICMP type field value value;
    - **icmp_code** – the code field in the ICMP header; relevant only if protocol field is of type ICMP
    - **icmp_code_mask** – mask to be applied on the ICMP code field value;
  - **dir_params** – direction specific parameter for the security policy:
    - **type** – type of parameters; configured using the `dpa_ipsec_pol_dir_params_type` enum; if equal to `DPA_IPSEC_POL_DIR_PARAMS_NONE`, the other fields' values are ignored;
    - **manip_desc** – fragmentation descriptor or header manipulation chain descriptor for policies attached to outbound SAs; relevant only if type is equal to `DPA_IPSEC_POL_DIR_PARAMS_MANIP`:

- **in_action** – the action that should be performed if the decrypted frames match the policy key; if this filed is NULL, the frames that pass inbound policy verification will be handled according to the default SA action configured at SA creation time; if this field is not NULL, it will override, for this policy only, the SA default action; this field is relevant only for policies attached to inbound SAs and if the type field is equal to `DPA_IPSEC_POL_DIR_PARAMS_ACT`;

- **priority** – the priority of this policy.

Description

This function can be used to configure the policies for a SA.

For outbound paths, the use of this function is mandatory, because the policy selectors passed as parameters to it will be used to classify frames and identify the correct SA that should be used for encryption.

For inbound paths the use of this function is mandatory if inbound policy verification was activated for this DPA IPSec instance and is prohibited if it was not activated. In case the inbound policy verification was activated, the policy selectors received as parameters will be used to check if the decrypted frames arrived on the correct SA.

In case the SA per DSCP feature needs to be enabled for an outbound SA selector, use_dscp must be set as true. In this case a number of selectors will be created according to DSCP range specified for the SA.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- EINVAL, if:
  - invalid function arguments were provided
  - the parameters of the classifier table could not be retrieved
  - no classifier table was configured for this type of policy
- EFAULT, if a bad argument was passed to a internal function.
- EBUSY , if a lock on the SA could not be acquired.
- EPERM, if:
  - othe SA is a parent in the rekeying process and is of type outbound
  - othe SA is a child in the rekeying process and is of type inbound
- ENOMEM, if memory could not be allocated for the new policy.
- EAGAIN, if the FMD header manipulation object for fragmentation could not be created.
- error codes returned by the `dpa_classif_table_insert_entry` (used for inserting policy keys).

9.9.3.3.2.3.2 Function: dpa_ipsec_sa_remove_policy

Syntax

```
int dpa_ipsec_sa_remove_policy(int sa_id,
                               struct dpa_ipsec_policy_params *policy_params);
```

Parameters

- **sa_id** - id of the SA that this policy is linked.

- **policy_params** - structure with policy selectors, used for identify the policy that must be deleted; for detailed information regarding the policy selectors please see the description of the parameters for `dpa_ipsec_sa_add_policy` function

Description

This function is used to remove a previously offloaded policy. The policy to be removed is identified by a policy parameters structure that is identical to the one that was passed in the `dpa_ipsec_sa_add_policy` function when the policy was offloaded.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if invalid function arguments were provided.
- `EBUSY`, if:
  - a lock on the SA could not be acquired
  - the FMD manipulation object used for fragmentation could not be removed
- `EFAULT`, if a bad argument was passed to a internal function
- `EPERM`, if:
  - the SA is a parent in the rekeying process and is of type outbound
  - the SA is a child in the rekeying process and is of type inbound
- `EDOM`, if a corresponding policy was not found in the SA's list of associated policies.
- error codes returned by the `dpa_classif_table_delete_entry_by_ref` (used for removing policy keys).

9.9.3.3.2.3.3 Function: dpa_ipsec_sa_flush_policies

Syntax

```
int    dpa_ipsec_sa_flush_policies(int sa_id);
```

Parameters

- **sa_id** - identifier of the SA for which to remove all policies..

Description

This function is used to remove all the policies for the given SA. The internal resources are recycled.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if invalid function arguments were provided.
- `EBUSY`, if, a lock on the SA could not be acquired.
- `EFAULT`, if a bad argument was passed to a internal function.
- `EAGAIN`, if one or more policies could not be removed.
- error codes returned by the `dpa_classif_table_delete_entry_by_ref` (used for removing policy keys).

9.9.3.3.2.3.4 Function: dpa_ipsec_sa_get_policies

Syntax

```
int    dpa_ipsec_sa_get_policies(int sa_id,
           struct dpa_ipsec_policy_params *policy_params,
           int    *num_pol);
```

Parameters

- **sa_id** - identifier of the SA for which to retrieve all policies.
- **policy_params** - placeholder for the array of policy structures associated to the specified SA; the array of policy parameters will be allocated internally by this function.
- **num_pol** - number of policy parameters structures in the returned policy array.

Description

This function is used to retrieve all the policies associated to a given SA. The array that will hold the retrieved policies must be allocated by the calling application.

In order to determine the correct size for this array, the calling application must first call the function with the `policy_params` parameter set to NULL. In this case the function will return in the `num_pol` parameter the number of policies that are linked to the specified SA. Then the calling application can allocated a policy array in which there is enough space for all the policies linked to the SA. After calling the function again, this time passing the address of the newly allocated array as the `policy_params` parameter, it will receive the information about all the policies linked to the specified SA.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if invalid function arguments were provided.
- `EBUSY`, if, a lock on the SA could not be acquired.
- `EFAULT`, if a bad argument was passed to a internal function.
- `EAGAIN`, if there are more policies in the SA list than the maximum number of policies that can be copied in the provided buffer.

9.9.3.3.2.4 Rekeying

Table 262. API for IPSec Rekeying

Function Name	Description
`dpa_ipsec_sa_rekeying`	Performs all the required actions for changing and expired SA with a newly negotiated SA, while ensuring no packets are lost during this process.

9.9.3.3.2.4.1 Function: dpa_ipsec_sa_get_rekeyings

Syntax

```
int     dpa_ipsec_sa_rekeying(int sa_id,
                              struct dpa_ipsec_sa_params *sa_params,
dpa_ipsec_rekey_event_cb *rekey_event_cb,
bool auto_rmv_old_sa, int *new_sa_id);



typedef int (*dpa_ipsec_rekey_event_cb)
(int dpa_ipsec_id,
                   int sa_id,
                   int error);
```

Parameters

- **sa_id** - id of the SA that has expired and needs to be replaced.
- **sa_params** - structure containing the parameters of the new SA; for a detailed description of these parameters see the description of `dpa_ipsec_create_sa`;
- **rekey_event_c** - callback used to report the status of the rekeying process, when it is completed; this callback is only supported in Kernel space
  - **dpa_ipsec_id** – id of the IPSec instance in which the SA rekeying process is performed;
  - **sa_id** – id of the SA for which the rekeying process was started;
  - **error** – status of the rekeying process;
- **auto_rmv_old_sa** - valid only for inbound rekeying. Auto remove old SA when encrypted traffic starts flowing on the new SA.
- **new_sa_id a** - address of an integer variable where the id of the newly created SA will be stored.

Description

This function is used to replace an expired SA with a new valid SA.

The rekeying procedure was designed in such a way that there is no service interruption, no packets are lost and the links between the policies and the old SA are automatically transferred to the new SA. Furthermore the rekeying process ensures that the packet order is preserved.

The `auto_rmv_old_sa` option has meaning only when rekeying an inbound SA as follows:

- If TRUE the DPA IPSEC will create the new SA and remove the old SA when it has received a frame encrypted with the new SA. This is done disregarding that the SA might be in its lifetime period.
- If FALSE the DPA IPSEC will create the new SA and let the old SA in place. Practically the two SA's will run in parallel. It is the responsibility of the user to call `dpa_ipsec_remove_sa` when the old SA expires (hardlimit reached). This option may be used when the user wants to receive frames encrypted with old SA which were reordered by the network and arrive after the new SA is already in use.

This function is the only one that uses an asynchronous error reporting mechanism. If an error occurs during the process of replacing the old SA with the new SA, which is an asynchronous process, the function will use the `dpa_ipsec_rekey_event_cb` to report the error to the calling application. If no error occurred during the rekeying process, the same callback will be used to signal successful completion of this task (i.e. error = 0).

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- -EINVAL, if:
  - no DPA IPSec instance is initialized
  - invalid function arguments or SA initialization parameters were provided
  - the calling application requested activation of NAT-T support for an IPv6 SA
  - authentication and encryption keys could not be mapped to a job ring device
  - the size of the calculated key for SA lookup is greater than the maximum key size of the classifier table in which it should be placed
- -EBUSY , if a lock on the SA could not be acquired.
- -EEXIST, if a bad argument was passed to a internal function.
- -EDOM, if a bad argument was passed to a internal function.
- -ERANGE, if a bad argument was passed to a internal function.
- -ENODEV, if a bad argument was passed to a internal function.
- -EFAULT, if a bad argument was passed to a internal function.
- -ENOMEM, if a bad argument was passed to a internal function.
  - an error occurred during preparation for split key generation
  - a new FQID could not be allocated from the fqid pool
- -EAGAIN, if there are more policies in the SA list than the maximum number of policies that can be copied in the provided buffer.
- error codes returned by caam_jr_enqueue function.
- error codes returned by the qman_create_fq and qman_init_fq functions.
- error codes returned by the dpa_classif_table_insert_entry (used for inserting inbound SA lookup keys) and dpa_classif_table_get_params functions
- -EUSERS, if the function dpa_classif_table_modify_entry_by_ref, used for modifying the actions associated to the outbound security policies, failed and only some of the security policies were transferred to the new SA
- error codes returned by the qman_schedule_fq function

The following error codes can be returned in the error parameter of the dpa_ipsec_rekey_event_cb callback:

- for outbound SAs:
  - –EUCLEAN, if during the rekeying process the removal of the TO SEC FQ of old SA failed. The upper layer has to call the dpa_ipsec_remove_sa at a later time (not from callback) to try again to free the old SA resources. The new SA is active.
  - –EDQUOT, if recycling of old SA's memory resources failed. The upper layer has to call the dpa_ipsec_remove_sa function at a later time (not from callback) to try again to recycle these resources.
- for inbound SAs:
  - –ENOTRECOVERABLE, if the removal of the old SA's table entry in in the SA lookup classifier table has failed. If this error is received the upper layer must take into consideration that the system is in a vulnerable state, because it will still accept packets that match the old SA's parameters, and it should consider rebooting the system.
  - –EUCLEAN, if during the rekeying process the removal of the TO SEC FQ of old SA failed. The upper layer has to call the dpa_ipsec_remove_sa at a later time (not from callback) to try again to free the old SA resources. The new SA is active.
  - –EDQUOT, if recycling of old SA's memory resources failed. The upper layer has to call the dpa_ipsec_remove_sa function at a later time (not from callback) to try again to recycle these resources.

9.9.3.4 DPA Statistics

The DPA Stats module exports a set of functions that can be used to:

- initialize the DPA Stats module;
- create a DPA Stats single counter;
- create a DPA Stats class counter;
- retrieve the values for a series of counters.

9.9.3.4.1 Initialization

Initialization is a mandatory step in using the DPA Stats component. The initialization function has the purpose of creating and initializing internal data structures that will be further used at runtime. The allocated internal structures will be later

Counter creation

Counter creation is an important step, as the counter is the basic element used to retrieve specific information. There are two types of counters that can be created: a single counter and a class counter.

A **single counter** is meant to retrieve information from one source, which can be either a hardware resource or a software resource.

A **class counter** has the purpose of retrieving information from multiple sources of the same type.

Counters are of different types; as such counter creation is responsible of configuring the interface with different hardware blocks or software components in order to be able to obtain the corresponding values.During counter creation no hardware block or software component is initialized, that is the responsibility of the user software. The counter relies on an already initialized hardware or software component and will only configure the mechanism it needs to gain the desired information. It is also the user responsibility to insure synchronization between the initialized hardware and software components and the created counters. As such a counter needs to be removed if the underlying hardware or software component was destroyed.



Figure 259. DPA Stats architecture

Counter retrieve

The user has the possibility of retrieving the values for one or multiple counters. For each counter, the retrieve operation means a direct interaction with the hardware or software component. As such, the response time of the retrieve operation depends on the number of counters the request is made of and the underlying hardware or software component response time.

The result of a retrieve operation is written by the DPA Stats component in a memory area provided by the user application and the user application will be informed on the number of bytes written either when the calling function returns or via a callback function.

9.9.3.4.2 DPA Statistics API

There are several selections for the DPA Statistics API:

9.9.3.4.2.1 Initialize and Free Statistics

Table 263. API to Initialize and Free Statistics

Function Name	Description	Input Parameters	Output Parameters
`dpa_stats_init`	Configures and initializes a DPA Stats instance.	• maximum number of counters that will be managed by the DPA Stats instance. • pointer to the memory area where the values of the counters will be written by the DPA Stats instance. • length of the memory area expressed in number of bytes.	Identifier of the newly created DPA Stats instance.
`dpa_stats_free`	Releases all resources associated with a DPA Stats instance and destroys it.	• identifier of the DPA Stats instance to destroy.	None.

9.9.3.4.2.1.1 Function: dpa_stats_init

Syntax

```
int dpa_stats_init(const struct dpa_stats_params *params, int
                   *dpa_stats_id);

struct dpa_stats_params
{
 unsigned int max_counters;
 void    *storage_area;
 unsigned int storage_area_len;
};
```

Parameters

- **params** - data structure containing the parameters needed to initialize the DPA Stats;
  - max_counters: maximum number of counters supported by the DPA Stats module.
  - storage_area: pointer to the memory area where the values of the counters will be written by the DPA Stats instance after a retrieve operation was successful.
  - storage_area_len: length of the memory area expressed in number of bytes.
- **dpa_stats_id** - a location where the function will return the identifier of the DPA Stats instance, in case of success; this id will be further used to create counters that are attached to the DPA Stats instance.

Description

This function is used to initialize a DPA Stats instance. The purpose of the function is to allocate and initialize all the internal structures that will be further needed by the DPA Stats instance.

The storage area can be a pointer to a memory allocated using C programming standard library routines or a pointer to a memory area allocated using NXP's "shmem" application programming interface. Memory allocated using "shmem" is a contiguous physical memory that is accessible both in user-space application and in kernel-space driver implementation through corresponding user-space and kernel space virtual addresses. This type of memory allows the DPA Stats driver to write the statistics values in the storage area using a "zero copy" approach and is an optimized solution to transfer information from kernel-space to user-space.

Memory area allocated in user-space using C programming routines is not mapped in kernel-space so the transfer between kernel-space DPA Stats driver and user-space application is performed using memory copy from kernel-to-user space and vice-versa.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if one of the function arguments or parameters was incorrect;
- `ENOMEM`, if there is no more memory to create an internal data structure needed by the DPA Stats module.
- `EPERM`, if the component is already initialized.
- `ENOSPC`, if the operation of creating a single threaded workqueue failed.
- `EDOM`, if the number of provided counters is above the maximum allowed.

9.9.3.4.2.1.2 Function: dpa_stats_free

Syntax

```
int dpa_stats_free(int dpa_stats_id);
```

Parameters

- **id** - identifier of the DPA Stats instance to destroy.

Description

Releases all resources associated with a DPA instance and destroys the instance.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if the provided instance identifier is not valid;
- `EDOM`, if the provided counter identifier could not be released.

9.9.3.4.2.2 Create, Remove or Modify Counters

Table 264. API to Create Remove or Modify Counters

Function Name	Description	Input Parameters	Output Parameters
`dpa_stats_create_counter`	Configures and initializes a DPA Stats counter. The counter will be used to address a single hardware or software resource.	• identifier of the DPA Stats instance the counter will belong to. • the type of counter that needs to be created • parameters needed to create the counter, depending of the counter type • pointer to a location where the function will return an unique identifier for the counter in case of success.	Identifier for the counter. The returned identifier will be further used to address the counter.
`dpa_stats_create_class_counter`	Configures and initializes a DPA Stats counter. The counter will be used to address multiple hardware or software resources of the same time.	• identifier of the DPA Stats instance the class counter will belong to. • the type of class counter that needs to be created. • parameters needed to create the class counter, depending of the class counter type. • pointer to a location where the function will return an unique identifier for the class counter in case of success.	Identifier for the class counter in case of success. The returned identifier will be further used to address the class counter.

Table continues on the next page...

Table 264. API to Create Remove or Modify Counters (continued)

Function Name	Description	Input Parameters	Output Parameters
`dpa_stats_modify_class_counter`	Modifies a member of a certain class counter. The class member can be invalidated or updated, by providing a new source.	• identifier of the DPA Stats class counter for which the member will be updated. • the type of the member that needs to be modified. • parameters needed to modify the class member. • the index of the member in the class counter.	None.
`dpa_stats_remove_counter`	Removes a single counter or a class counter. The resources occupied by the counter are marked as empty.	• identifier of the DPA Stats counter or class counter that will be removed.	None.

9.9.3.4.2.2.1 Function: dpa_stats_create_counter

Syntax

```
int dpa_stats_create_counter(int dpa_stats_id,
 const struct dpa_stats_cnt_params *params,int *dpa_stats_cnt_id);

struct dpa_stats_cnt_params
{
 enum dpa_stats_cnt_type type;
 union
{
  struct dpa_stats_cnt_eth   eth_params;
  struct dpa_stats_cnt_reass   reass_params;
struct dpa_stats_cnt_frag   frag_params;
struct dpa_stats_cnt_plcr   plcr_params;
  struct dpa_stats_cnt_classif_tbl  classif_tbl_params;
struct dpa_stats_cnt_classif_node  classif_node_params;
  struct dpa_stats_cnt_ipsec   ipsec_params;
  struct dpa_stats_cnt_traffic_mng  traffic_mng_params;
 };
};

enum dpa_stats_cnt_type
{
 DPA_STATS_CNT_ETH = 0,
 DPA_STATS_CNT_REASS,
DPA_STATS_CNT_FRAG,
DPA_STATS_CNT_POLICER,
DPA_STATS_CNT_CLASSIF_TBL,
DPA_STATS_CNT_CLASSIF_NODE,
 DPA_STATS_CNT_CLASSIF,
 DPA_STATS_CNT_IPSEC,
 DPA_STATS_CNT_TRAFFIC_MNG,
```

```
 DPA_STATS_CNT_RAPIDIO
};

enum dpa_stats_cnt_sel
{
 DPA_STATS_CNT_NUM_OF_BYTES = 0,
 DPA_STATS_CNT_NUM_OF_PACKETS,
 DPA_STATS_CNT_NUM_ALL
};

enum dpa_stats_cnt_eth_sel
{
 DPA_STATS_CNT_ETH_DROP_PKTS    = 0x00000001,
 DPA_STATS_CNT_ETH_BYTES    = 0x00000002,
 DPA_STATS_CNT_ETH_PKTS     = 0x00000004,
 DPA_STATS_CNT_ETH_BC_PKTS    = 0x00000008,
 DPA_STATS_CNT_ETH_MC_PKTS    = 0x00000010,
 DPA_STATS_CNT_ETH_CRC_ALIGN_ERR   = 0x00000020,
 DPA_STATS_CNT_ETH_UNDERSIZE_PKTS   = 0x00000040,
 DPA_STATS_CNT_ETH_OVERSIZE_PKTS   = 0x00000080,
 DPA_STATS_CNT_ETH_FRAGMENTS    = 0x00000100,
 DPA_STATS_CNT_ETH_JABBERS    = 0x00000200,
 DPA_STATS_CNT_ETH_64BYTE_PKTS   = 0x00000400,
 DPA_STATS_CNT_ETH_65_127BYTE_PKTS   = 0x00000800,
 DPA_STATS_CNT_ETH_128_255BYTE_PKTS   = 0x00001000,
 DPA_STATS_CNT_ETH_256_511BYTE_PKTS   = 0x00002000,
 DPA_STATS_CNT_ETH_512_1023BYTE_PKTS  = 0x00004000,
 DPA_STATS_CNT_ETH_1024_1518BYTE_PKTS = 0x00008000,
DPA_STATS_CNT_ETH_OUT_PKTS    = 0x00010000,
 DPA_STATS_CNT_ETH_OUT_DROP_PKTS   = 0x00020000,
DPA_STATS_CNT_ETH_OUT_BYTES     = 0x00040000,
DPA_STATS_CNT_ETH_IN_ERRORS     = 0x00080000,
DPA_STATS_CNT_ETH_OUT_ERRORS     = 0x00100000,
 DPA_STATS_CNT_ETH_IN_UNICAST_PKTS   = 0x00200000, DPA_STATS_CNT_ETH_OUT_UNICAST_PKTS   =
0x00400000,
 DPA_STATS_CNT_ETH_ALL     = 0x00800000
};

enum dpa_stats_cnt_eth_id
{
 DPA_STATS_ETH_1G_PORT0 = 0,
 DPA_STATS_ETH_1G_PORT1,
 DPA_STATS_ETH_1G_PORT2,
 DPA_STATS_ETH_1G_PORT3,
DPA_STATS_ETH_1G_PORT4,
 DPA_STATS_ETH_10G_PORT0
};

struct dpa_stats_cnt_eth_src
{
 uint8_t     engine_id;
 enum dpa_stats_cnt_eth_id  eth_id;
};

struct dpa_stats_cnt_eth
{
 struct dpa_stats_cnt_eth_src  src;
 unsigned int    cnt_sel;
};
```

```
enum dpa_stats_cnt_reass_gen_sel
{
 DPA_STATS_CNT_REASS_TIMEOUT        = 0x00000001,
 DPA_STATS_CNT_REASS_RFD_POOL_BUSY   = 0x00000002,
 DPA_STATS_CNT_REASS_INT_BUFF_BUSY   = 0x00000004,
 DPA_STATS_CNT_REASS_EXT_BUFF_BUSY   = 0x00000008,
 DPA_STATS_CNT_REASS_SG_FRAGS        = 0x00000010,
 DPA_STATS_CNT_REASS_DMA_SEM        = 0x00000020,
 DPA_STATS_CNT_REASS_NON_CONSISTENT_SP = 0x00000040,
 DPA_STATS_CNT_REASS_GEN_ALL        = 0x00000080
};

enum dpa_stats_cnt_reass_ipv4_sel
{
 DPA_STATS_CNT_REASS_IPv4_FRAMES   = 0x00000100,
 DPA_STATS_CNT_REASS_IPv4_FRAGS_VALID  = 0x00000200,
 DPA_STATS_CNT_REASS_IPv4_FRAGS_TOTAL  = 0x00000400,
 DPA_STATS_CNT_REASS_IPv4_FRAGS_MALFORMED  = 0x00000800,
 DPA_STATS_CNT_REASS_IPv4_FRAGS_DISCARDED  = 0x00001000,
 DPA_STATS_CNT_REASS_IPv4_AUTOLEARN_BUSY  = 0x00002000,
 DPA_STATS_CNT_REASS_IPv4_EXCEED_16FRAGS  = 0x00004000,
 DPA_STATS_CNT_REASS_IPv4_ALL    = 0x00008000
};

enum dpa_stats_cnt_reass_ipv6_sel
{
 DPA_STATS_CNT_REASS_IPv6_FRAMES   = 0x00010000,
 DPA_STATS_CNT_REASS_IPv6_FRAGS_VALID  = 0x00020000,
 DPA_STATS_CNT_REASS_IPv6_FRAGS_TOTAL  = 0x00040000,
 DPA_STATS_CNT_REASS_IPv6_FRAGS_MALFORMED  = 0x00080000,
 DPA_STATS_CNT_REASS_IPv6_FRAGS_DISCARDED  = 0x00100000,
 DPA_STATS_CNT_REASS_IPv6_AUTOLEARN_BUSY  = 0x00200000,
 DPA_STATS_CNT_REASS_IPv6_EXCEED_16FRAGS  = 0x00400000,
 DPA_STATS_CNT_REASS_IPv6_ALL    = 0x00800000
};

struct dpa_stats_cnt_reass
{
 void  *reass;
 unsigned int cnt_sel;
};

enum dpa_stats_cnt_frag_sel
{
 DPA_STATS_CNT_FRAG_TOTAL_FRAMES  = 0x00000001,
 DPA_STATS_CNT_FRAG_FRAMES   = 0x00000002,
 DPA_STATS_CNT_FRAG_GEN_FRAGS   = 0x00000004,
 DPA_STATS_CNT_FRAG_ALL    = 0x00000008
};

struct dpa_stats_cnt_frag
{
 void  *frag;
 unsigned int cnt_sel;
};

enum dpa_stats_cn_plcr_sel
{
 DPA_STATS_CNT_PLCR_GREEN_PKTS   = 0x00000001,
DPA_STATS_CNT_PLCR_YELLOW_PKTS   = 0x00000002,
```

```
 DPA_STATS_CNT_PLCR_RED_PKTS    = 0x00000004,
 DPA_STATS_CNT_PLCR_RECOLOR_YELLOW_PKTS  = 0x00000008,
 DPA_STATS_CNT_PLCR_RECOLOR_RED_PKTS  = 0x00000010,
 DPA_STATS_CNT_PLCR_ALL     = 0x00000020
};

struct dpa_stats_cnt_plcr
{
 void  *plcr;
 unsigned int cnt_sel;
};

enum dpa_stats_cnt_classif_sel {
 DPA_STATS_CNT_CLASSIF_BYTES   = 0x00000010,
 DPA_STATS_CNT_CLASSIF_PACKETS  = 0x00000020,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE1  = 0x00000040,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE2  = 0x00000080,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE3  = 0x00000100,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE4  = 0x00000200,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE5  = 0x00000400,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE6  = 0x00000800,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE7  = 0x00001000,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE8  = 0x00002000,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE9  = 0x00004000,
 DPA_STATS_CNT_CLASSIF_RMON_RANGE10  = 0x00008000,
};

struct dpa_stats_cnt_classif_tbl
{
 int       td;
 const struct dpa_offload_lookup_key *key;
 unsigned int     cnt_sel;
};

enum dpa_stats_classif_node_type
{
 DPA_STATS_CLASSIF_NODE_HASH = 0,
 DPA_STATS_CLASSIF_NODE_INDEXED,
 DPA_STATS_CLASSIF_NODE_EXACT_MATCH
};

struct dpa_stats_cnt_classif_node
{
 void       *cc_node;
enum dpa_stats_classif_node_type  ccnode_type;
 struct dpa_offload_lookup_key  *key;
 unsigned int     cnt_sel;
};

struct dpa_stats_cnt_ipsec
{
 int     sa_id;
 enum dpa_stats_cnt_sel  cnt_sel;
};

enum dpa_stats_cnt_traffic_mng_src
{
 DPA_STATS_CNT_TRAFFIC_CLASS = 0,
 DPA_STATS_CNT_TRAFFIC_CG
};
```

```
struct dpa_stats_cnt_traffic_mng
{
 enum dpa_stats_cnt_traffic_mng_src  src;
 void       *traffic_mng;
 enum dpa_stats_cnt_sel    cnt_sel;
};
```

Parameters

- **dpa_stats_id** - identifier of the DPA Stats instance the counter belongs to.
- **params** - structure holding the parameters needed to create a counter:
  - **type**: the type of the counter that needs to be created: Ethernet counter, Reassembly counter, Fragmentation Counter, Policer Counter, Classifier Table counter, Classification Node Counter, IPSec counter, Traffic Manager counter or RapidIO counter (selected from `dpa_stats_cnt_type` enum).
  - **eth_params**: structure holding the parameters needed to create an Ethernet counter
    - **src**: structure holding the parameters needed to identify the source of the Ethernet counter
      - **engine_id**: identifier of the engine the Ethernet interface belongs to
      - **eth_id**: identifier of the Ethernet interface, identifier that is relative to the engine the interface belongs to
    - **cnt_sel**: selection from Ethernet available counters: enum `dpa_stats_cnt_eth_sel`
  - **reass_params**: structure holding the parameters needed to create an IP Reassembly counter
    - **reass**: handle of IP Reassembly object
    - **cnt_sel**: selection from Reassembly available counters, enum `dpa_stats_cnt_reass_gen_sel`, `dpa_stats_cnt_reass_ipv4_sel` or `dpa_stats_cnt_reass_ipv6_sel`
  - **frag_params**: structure holding the parameters needed to create an IP Fragmentation counter
    - **frag**: handle of IP Fragmentation object
    - **cnt_sel**: selection from Fragmentation available counters, enum `dpa_stats_cnt_frag_sel`
  - **plcr_params**: structure holding the parameters needed to create a Policer counter
    - **plcr**: handle of Policer object
    - **cnt_sel**: selection from Policer available counters, enum `dpa_stats_cn_plcr_sel`
  - **classif_tbl_params**: structure holding the parameters needed to create a Classifier Table counter
    - **td**: descriptor of the table used to perform the Classification
    - **key**: pointer to the key descriptor. If *key* is NULL, DPA Stats will provide the table *miss* condition statistics in this counter.
      - **byte**: pointer to an array of bytes representing the key
      - **mask**: the mask to store for this key
      - **size**: the size of the key
    - **cnt_sel**: selection from Classifier Table available counters, enum `dpa_stats_cnt_classif_sel` or enum `dpa_stats_cnt_frag_sel`
  - **classif_node_params**: structure holding the parameters needed to create a Classification Node counter
    - **cc_node**: handle of the Cc node used to perform the Classification
    - **ccnode_type**: the type of FMAN Classification Node

- **key**: pointer to the key descriptor. If *key* is NULL, DPA Stats will provide the CC node's *miss* condition statistics in this counter.
  - **byte**: pointer to an array of bytes representing the key
  - **mask**: the mask to store for this key
  - **size**: the size of the key
- **cnt_sel**: selection from Classification Node available counters, enum `dpa_stats_cnt_classif_sel`

- **ipsec_params**: structure holding the parameters needed to create an IPSec counter
  - **sa_id**: identifier of the Security Association
  - **cnt_sel**: single selection from IPSec available counters, enum `dpa_stats_cnt_sel`
- **traffic_mng_params**: structure holding the parameters needed to create a Traffic Manager counter
  - **src**: the type of the source used for the Traffic Manager counter
  - **traffic_mng**: depending on the Traffic Manager source, the 'traffic_mng' has a different meaning: it represents a pointer to a structure of type 'qm_ceetm_cq' in case the traffic source is a "Class Queue" or a pointer to a structure of type 'qm_ceetm_ccg' in case the traffic source is a "Class Congestion Group"
  - **cnt_sel**: single selection from Traffic Manager counter, `dpa_stats_cnt_sel` enum
- **rapidio_params**: structure holding the parameters needed to create a RapidIO counter

- **dpa_stats_cnt_id** - address of an integer variable in which the function will return the counter identifier if it succeeds in creating it; this identifier will be used in further calls to DPA Stats functions to refer to this particular counter.

Description

This function performs all the necessary steps required to create and initialize a single counter. A single counter means a counter that is used to retrieve information from one source, source that can be either a hardware resource or a software resource.

During counter creation no memory is allocated, but instead the function uses internal structures that were allocated during DPA Stats instance initialization. On success, the function returns a unique counter identifier that should be further used on calls that refer to that counter.

The first parameter of the structure used to create a counter is the type of the counter. Depending on the type of the counter, the user needs to provide the parameters necessary for that type of counter.

For the Ethernet counter, the user can select any combinations of counters from the enumeration `dpa_stats_cnt_eth_sel`.

For the Reassembly counter, the user can have combination of selections from the three groups of counters: counters that are common both to IPv4 and IPv6 protocols, counters specific to IPv4 protocol and counters specific to IPv6 protocol: `dpa_stats_cnt_reass_gen_sel`, `dpa_stats_cnt_reass_ipv4_sel` and `dpa_stats_cnt_reass_ipv6_sel`.

For the Fragmentation and Policer counter, the user can have combination of selections from the corresponding enumerations: `dpa_stats_cnt_frag_sel` and `dpa_stats_cnt_plcr_sel`.

In terms of Classification, there are two types of Classification counters: a Classification Table counter and a Classification Node counter. For a Classification Table counter, the user can retrieve the statistics values for a specific table entry, statistics values that can be any selection or combination of selection from `dpa_stats_cnt_frag_sel` or `dpa_stats_cnt_classif_sel`. The user should be aware that when the same fragmentation object is applied on more than one Classifier Table entries, the statistics values represent the values for the entire number of entries on which the same fragmentation object is applied.

For IPSec and Traffic Manager counters, the user has the possibility of selection from the enum `dpa_stats_cnt_sel`, which return the number of bytes, number of frames or both of them but the returned value has a different meaning, depending on the type of counter and source of the counter.

The DPA Stats component verifies every valid source provided during counter creation by trying to retrieve the corresponding statistics.

The value for a single statistics value is of 4 bytes, but in case multiple statistics values were selected by combining them the value returned will be 4 bytes multiplied with the number of statistics selected.

Return Value

This function returns 0 for success or an error code otherwise. The returned error codes are the following:

- `EPERM`, if no DPA Stats instance is initialized.
- `EINVAL`, if one of the function arguments or parameters was incorrect.
- `EDOM`, if the number of previously created counters reached the maxium preconfigured number of counters.

9.9.3.4.2.2.2 Function: dpa_stats_create_class_counter

Syntax

```
int dpa_stats_create_class_counter(int dpa_stats_id,
  const struct dpa_stats_cls_cnt_params *params, int *dpa_stats_cnt_id);

struct dpa_stats_cls_cnt_params
{
 int     class_members;
 enum dpa_stats_cnt_type type;
 union
{
  struct dpa_stats_cls_cnt_eth     eth_params;
  struct dpa_stats_cls_cnt_reass    reass_params;
struct dpa_stats_cls_cnt_frag    frag_params;
struct dpa_stats_cls_cnt_plcr    plcr_params;
struct dpa_stats_cls_cnt_classif_tbl  cls_tbl_params;
struct dpa_stats_cls_cnt_classif_node cls_node_params;
  struct dpa_stats_cls_cnt_classif    cls_params;
  struct dpa_stats_cls_cnt_ipsec    ipsec_params;
  struct dpa_stats_cls_cnt_traffic_mng  traffic_mng_params;
  struct dpa_stats_cls_cnt_rapidio    rapidio_params;
 };
};

struct dpa_stats_cls_cnt_eth
{
 struct dpa_stats_cnt_eth_src  *src;
 unsigned int    cnt_sel;
};

struct dpa_stats_cls_cnt_reass
{
 void   **reass;
 unsigned int cnt_sel;
};

struct dpa_stats_cls_cnt_frag
{
 void   **frag;
 unsigned int cnt_sel;
};

struct dpa_stats_cls_cnt_plcr
{
```

```
 void   **plcr;
 unsigned int cnt_sel;
};

enum dpa_stats_classif_key_type
{
 DPA_STATS_CLASSIF_SINGLE_KEY = 0,
 DPA_STATS_CLASSIF_PAIR_KEY
};

struct dpa_offload_lookup_key_pair
{
 struct dpa_offload_lookup_key *first_key;
 struct dpa_offload_lookup_key *second_key;
};

struct dpa_stats_cls_cnt_classif_tbl
{
 int td;
 enum dpa_stats_classif_key_type   key_type;

 union {
  struct dpa_offload_lookup_key      **keys;
  struct dpa_offload_lookup_key_pair **pairs;
 };
 unsigned int cnt_sel;
};

enum dpa_stats_classif_node_type
{
 DPA_STATS_CLASSIF_NODE_HASH = 0,
 DPA_STATS_CLASSIF_NODE_INDEXED,
 DPA_STATS_CLASSIF_NODE_EXACT_MATCH
};

struct dpa_stats_cls_cnt_classif_node
{
 void              *cc_node;
enum dpa_stats_classif_node_type    ccnode_type;
 struct dpa_offload_lookup_key      **keys;
 unsigned int       cnt_sel;
};

struct dpa_stats_cls_cnt_ipsec
{
 int     *sa_id;
 enum dpa_stats_cnt_sel  cnt_sel;
};

struct dpa_stats_cls_cnt_traffic_mng
{
 enum dpa_stats_cnt_traffic_mng_src  src;
 void       **traffic_mng;
 enum dpa_stats_cnt_sel        cnt_sel;
};

struct dpa_stats_cls_cnt_rapidio;
```

Parameters

- **dpa_stats_id** - identifier of the DPA Stats instance the class counter belongs to.
- **params** - structure holding the parameters needed to create a counter:
  - **class_members**: the number of members or sources of the same type the class counter provides information for
  - **type**: the type of the class counter that needs to be created: Ethernet counter, Reassembly counter, Fragmentation Counter, Policer Counter, Classification Table, Classification Node counter, IPSec counter, Traffic Manager counter or RapidIO counter (selected from dpa_stats_cnt_type enum).
  - **eth_params**: structure holding the parameters needed to create an Ethernet class counter
    - **src**: an array of structures holding the parameters needed to identify multiple sources. Each structure identifies one Ethernet source:
      - **engine_id**: identifier of the engine the Ethernet interface belongs to
      - **eth_id**: identifier of the Ethernet interface, identifier that is relative to the engine the interface belongs to
    - **cnt_sel**: selection of Ethernet available counters, `dpa_stats_cnt_eth_sel enum`
  - **reass_params**: structure holding the parameters needed to create an IP Reassembly counter
    - **reass**: an array of IP Reassembly objects
    - **cnt_sel**: selection of counters from Reassembly available counters, `dpa_stats_cnt_reass_gen_sel`, `dpa_stats_cnt_reass_ipv4_sel` and `dpa_stats_cnt_reass_ipv6_sel` enums
  - **frag_params**: structure holding the parameters needed to create an IP Fragmentation counter
    - **frag**: an array of IP Fragmentation objects
    - **cnt_sel**: selection of counters from Fragmentation available counters, dpa_stats_cnt_frag_sel enum
  - **plcr_params**: structure holding the parameters needed to create a Policer counter
    - **plcr**: an array of Policer objects
    - **cnt_sel**: selection of counters from Policer available counters, dpa_stats_cn_plcr_sel enum
  - **cls_tbl_params**: structure holding the parameters needed to create a Classification Table counter
    - **td**: descriptor of the table used to perform the Classification
    - **key_type**: the type of the key used to identify an entry in the Classification
    - **keys**: Pointer to an array of keys, where each element of the array can be either a key that identifies a specific entry or NULL in order to obtain the statistics for the *miss* entry.
    - **pairs**: Pointer to an array of "pair-keys" where each element of the array can either be a "pair-key" that identifies a specific entry, or NULL. It is allowed to specify a *miss* table relationship in each of the key pairs by providing NULL for that specific key (i.e. first or second key). If one wants to get statistics for the *miss-miss* pair, you can provide NULL for both keys inside the key pair, or you can simply provide NULL for the pair itself in this array.
    - **cnt_sel**: selection from Classification Table available counters, dpa_stats_cnt_classif_sel and dpa_stats_cnt_frag_sel enums
  - **cls_node_params**: structure holding the parameters needed to create a Classification Node counter
    - **cc_node**: handle of the Cc node used to perform the Classification
    - **ccnode_type**: the type of FMAN Classification Node
    - **keys**: Pointer to an array of keys, where each element of the array can be either a key that identifies a specific entry or NULL in order to obtain the statistics for the *miss* entry.
    - **cnt_sel**: selection from Classification Node available counters, `dpa_stats_cnt_classif_sel` enum
  - **ipsec_params**: structure holding the parameters needed to create an IPSec counter

  - **sa_id**: identifier of the Security Association
  - **cnt_sel**: single selection from IPSec available counters, dpa_stats_cnt_sel enum
- **traffic_mng_params**: structure holding the parameters needed to create a Traffic Manager counter
  - **src**: the type of the source used for the Traffic Manager counter
  - **traffic_mng**: depending on the Traffic Manager source, the 'traffic_mng' has a different meaning: it represents an array of pointers to structures of type 'qm_ceetm_cq' in case the traffic source is a "Class Queue" or an array of pointers to structures of type 'qm_ceetm_ccg' in case the traffic source is a "Class Congestion Group"
  - **cnt_sel**: single selection from Traffic Manager counter, `dpa_stats_cnt_sel` enum
- **rapidio_params**: structure holding the parameters needed to create a RapidIO counter

- **dpa_stats_cnt_id** - address of an integer variable in which the function will return the counter identifier if it succeeds in creating it; this identifier will be used in further calls to DPA Stats functions to refer to this particular counter.

Description

This function performs all the necessary steps required to create and initialize a class counter. A class counter means a counter that is used to retrieve information from multiple sources of the same type, source which can be either a hardware resource or a software resource.

During class counter creation no memory is allocated but instead the function uses internal structures that were allocated during DPA Stats instance initialization. On success, the function returns a unique counter identifier that should be further used on calls that refer to that counter.

The first parameter of the structure used to create a counter is the type of the counter. Depending on the type of the counter, the user needs to provide the parameters necessary for that type of counter.

For the Ethernet counter, the user can select any combinations of counters from the enumeration `dpa_stats_cnt_eth_sel`.

For the Fragmentation and Policer counter, the user can have combination of selections from the corresponding enumerations: `dpa_stats_cnt_frag_sel` and `dpa_stats_cnt_plcr_sel`.

In terms of Classification, there are two types of Classification counters: a Classification Table counter and a Classification Node counter. For a Classification Table counter, the user can retrieve the statistics values for multiple table entries, statistics values that can be any selection or combination of selection from `dpa_stats_cnt_frag_sel` or `dpa_stats_cnt_classif_sel`. The user should be aware that when the same fragmentation object is applied on more than one Classifier Table entries, the statistics values represent the values for the entire number of entries on which the same fragmentation object is applied. In order to identify multiple table entries, the user application needs to provide an array of keys, keys which can be either single keys or pair of keys. When a pair of keys is used, the first key will identify one entry from the table, while the second key identifies an entry connected with the first entry on a “next” action.

For IPSec and Traffic Manager counters, the user has the possibility of selection from the enum `dpa_stats_cnt_sel`, which return the number of bytes, number of frames or both of them but the returned value has a different meaning, depending on the type of counter and source of the counter.

For two types of class counter, meaning Classification Table and IPSec, the user can provide an invalid source, in which case the returned value of the associated statistics will be 0. In this way, the user application can provide during creation only a subset of valid sources and update the rest of the sources during run-time. For a Classication Table counter an invalid source means a NULL pointer for a specific key, in case the key type is single, or a NULL pointer for the first key, in case the key type is pair of keys. For an IPSec counter, an invalid source means an invalid security association identifier, whose value is equal to -1.

The DPA Stats component verifies every valid source provided during counter creation by trying to retrieve the corresponding statistics.

The value for a single statistics value is of 4 bytes, but in case multiple statistics values were selected by combining them, the value returned will be 4 bytes multiplied with the number of statistics selected.

Return Value

This function returns 0 for success or an error code otherwise. The returned error codes are the following:

- `EPERM`, if no DPA Stats instance is initialized.
- `EINVAL`, if one of the function arguments or parameters was incorrect.
- `EDOM`, if the number of previously created counters reached the maxium preconfigured number of counters.

9.9.3.4.2.2.3 Function: dpa_stats_modify_class_counter

Syntax

```
int dpa_stats_modify_class_counter(int dpa_stats_cnt_id,
 const struct dpa_stats_cls_member_params *params, int member_index)

struct dpa_stats_cls_member_params
{
 enum dpa_stats_cls_member_type type;
 union
{
  struct dpa_offload_lookup_key *key;
  struct dpa_offload_lookup_key_pair *pair;
  int sa_id;
 };
};
```

Parameters

- **dpa_stats_cnt_id** - identifier of the class counter the member belongs to.
- **params** - structure holding the parameters needed to update a member of a class:
  - **type**: the type of the class member that needs to be updated: single key, pair key or security association identifier (selected from `dpa_stats_cls_member_type` enum)
  - **key**: a key descriptor to identify a specific table entry, or NULL to identify the table *miss* counter
  - **pair**: a pair of key descriptors to identify a specific table entry. It is allowed to specify a miss table relationship in each of the key pairs by providing NULL for that specific key (i.e. first or second key). If one wants to get statistics for the *miss-miss* pair, you can provide NULL for both keys inside the key pair, or you can simply provide NULL for the pair.
  - **sa_id**: a security association identifier
- **member_index** - the position of the member in the counter's class.

Description

This function is used to modify a member of a class counter of type Classification Table or IPSec, identified through the position of the member in the class. The user application can invalidate a specific member, in which case the returned statistics value is 0, can set a valid source (key, pair or sa id) in case the member of the class was created with an invalid source or can even update a valid source.

A source is considered invalid in the following cases:

- single key: if the provided key byte and key mask are NULL
- pair key: if the provided first key byte and mask are NULL
- sa id: if the provided security association identifier is -1

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if the provided class counter identifier is not valid;

9.9.3.4.2.2.4 Function: dpa_stats_remove_counter

Syntax

```
int dpa_stats_remove_counter(int dpa_stats_cnt_id);
```

Parameters

- **dpa_stats_cnt_id** - of the counter (single or class counter) it needs to be removed.

Description

This function is used to remove a previously created DPA Stats single or class counter. The memory occupied by the internal structures of this counter is not released, but instead it is marked as empty and can be used the next time a counter is created. After a counter is removed, it can no longer be used to retrieve values for it.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if the provided instance identifier is not valid;
- `EDOM`, if the provided counter identifier could not be released.

9.9.3.4.2.3 Retrieve or Reset Counters

Table 265. API to Retrieve or Reset Counters

Function Name	Description	Input Parameters	Output Parameters
dpa_stats_get_counters	Retrieves the values for multiple single or class counters.	• an array of single or class counters identifiers for which to retrieve values. • the size of the array of requested counters. • flag to specify if requested counters should be reset after their values are retrieved. • offset in the memory area provided at DPA Stats initialization where to write the counter values. • pointer to a location where the function will return the number of bytes that should be written by the DPA Stats in the storage area in case the operation was successful • pointer to a callback function to be called by the DPA Stats module when counter values were written in the storage area or NULL in case the request is a synchronous operation	The number of bytes written by the DPA Stats in the storage area in case the operation was successful.
dpa_stats_reset_counters	Reset the values for multiple single or class counters.	• an array of single or class counters identifiers for which to reset the statistics. • the size of the array of counters to be reset.	None.

9.9.3.4.2.3.1 Function: dpa_stats_get_counters

Syntax

```
int dpa_stats_get_counters(struct dpa_stats_cnt_request_params params,
 int *cnts_len, dpa_stats_request_cb request_done);

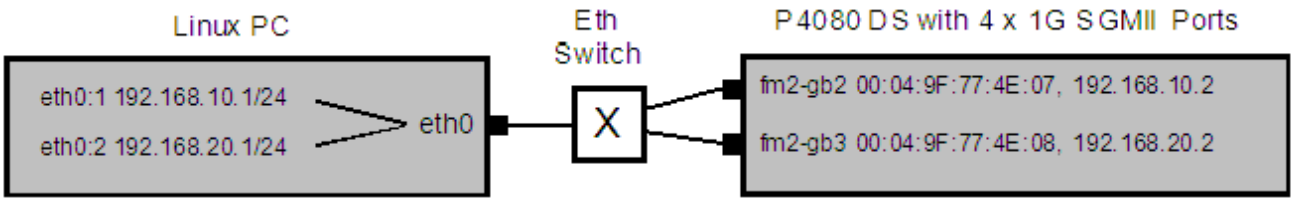
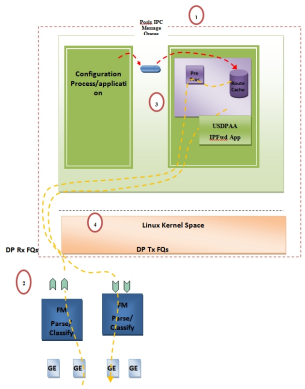
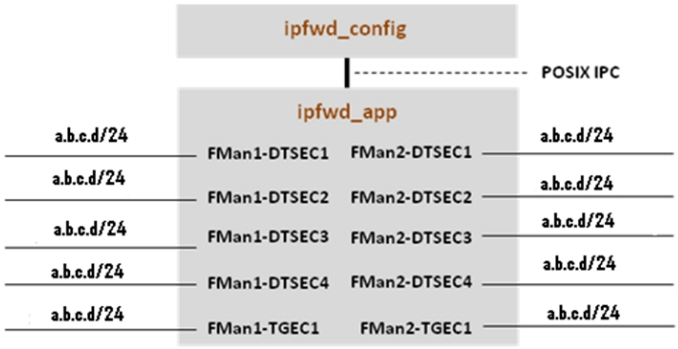
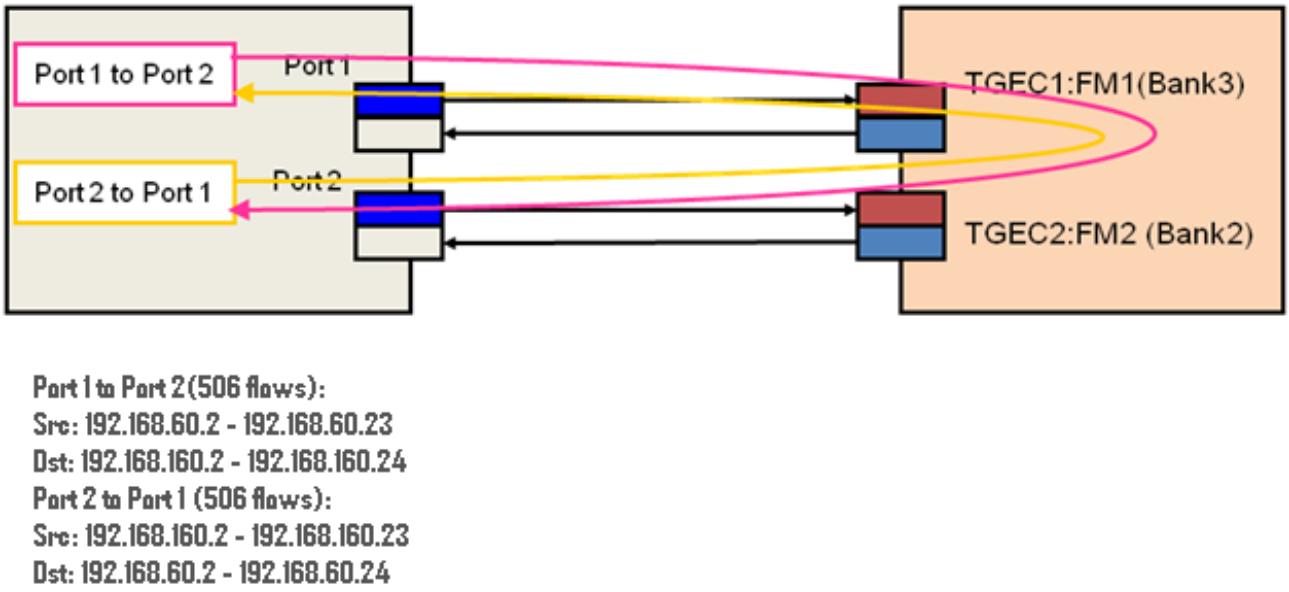
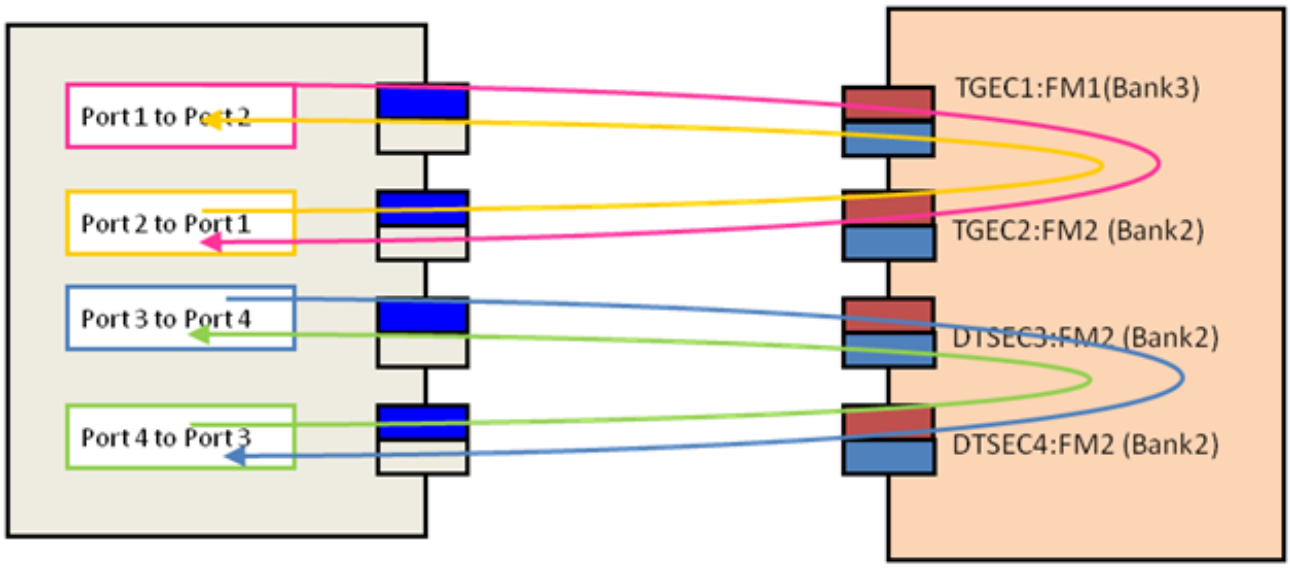
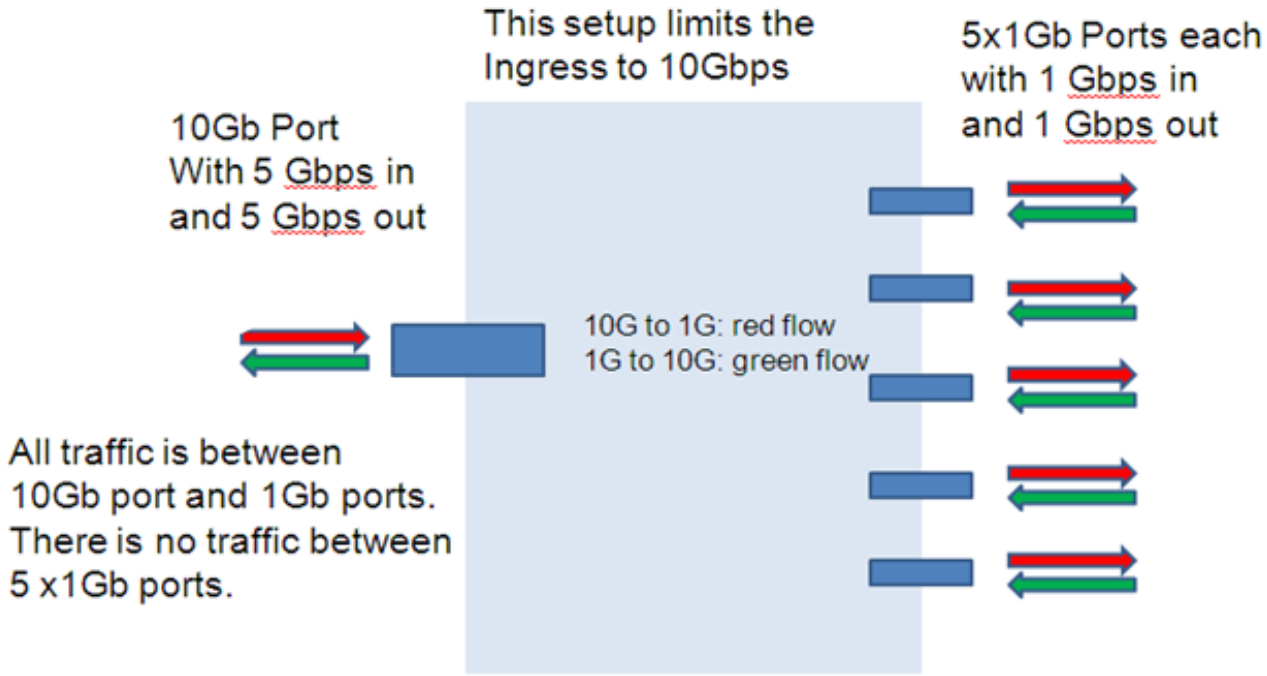
typedef void (*dpa_stats_request_cb)(int dpa_stats_id, unsigned int   storage_area_offset, unsigned
int cnts_written, int bytes_written);

struct dpa_stats_cnt_request_params
```

```
{
 int    *cnts_ids;
 unsigned int cnts_ids_len;
 bool    reset_cnts;
 unsigned int storage_area_offset;
};
```

Parameters

- **params** structure holding the parameters needed to retrieve the values for a group of counters
  - **cnts_ids**: pointer to an array of counter identifiers for which to retrieve values
  - **cnts_ids_len**: the size of array of counters to retrieve values
  - **reset_cnts**: flag to specify if counters provided in the cnts_ids array should be reset after the retrieve operation
  - **storage_area_offset**: offset in the storage area where to receive the values of the counters
- **cnts_len**: address of an integer variable in which the function will return the number of bytes occupied by the requested counters values
- **request_done** - pointer to a callback function that will be called by the DPA Stats module when the retrieve operation finished writing the counter values in the storage area or NULL in case the retrieve operation is synchronous
  - **dpa_stats_id**: DPA Stats instance identifier
  - **storage_area_offset**: offset in the storage area where the retrieve operation wrote the counter values
  - **cnts_written**: number of counters that were written in the storage area
  - **bytes_written**: number of bytes that were written in the storage area or the error code in case an error occurred

Description

This function is used to retrieve values for a group of counters. The statistics will be retrieved in the order given by the counter identifiers position in the array cnts_ids.

Each counter statistics value represents cumulative counters and is stored on 4 bytes. The user application has the possibility of resetting the counters values after the retrieve operation, by enabling the reset_cnts flag.

For a counter of type Classification Table or IPSec, in case the user application explicitly provided an invalid source, the corresponding statistics value will be 0 and the retrieve operation is considered successful.

A counters retrieve request can be either synchronous or asynchronous. The difference between them is made through the `request_done` callback. In case the corresponding parameter is NULL, the request will be synchronous and the counters values will be available in the storage area when the function returns. In case an error occurred during the retrieve operation, the function will return the corresponding error. If the user application provided the request_done callback, the retrieve operation is asynchronous and the callback will be called by the DPA Stats module as soon as the counters values are written in the storage area. In case of an error, the number of bytes_written will have a negative value and it will the store the error code.

The user application needs to be aware of the length of the counters values before making the call to the retrieve operation. The only check the DPA Stats module performs is to verify if the offset provided by the call and the size of the counters goes beyond the length of the storage area. If that is not the case, the counters values are written starting with the storage_area_offset and will override any information that might be available in the memory area it needs to write.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if invalid function arguments were provided;

- ENOENT, if an error occurred during statistics retrieve for an Ethernet counter;
- ESRCH, if an error occurred during statistics retrieve for a Reassembly counter;
- EINTR, if an error occurred during statistics retrieve for a Fragmentation counter;
- EIO, if an error occurred during statistics retrieve for a Classifier Table counter;
- ENXIO, if an error occurred during statistics retrieve for a Classification Node counter;
- E2BIG, if an error occurred during statistics retrieve for an IPSec counter.

9.9.3.4.2.3.2 Function: dpa_stats_reset_counters

Syntax

```
int dpa_stats_reset_counters(int *cnts_ids, unsigned int cnts_ids_len);
```

Parameters

- **cnts_ids** - pointer to an array of counter identifiers for which to reset the statistics.
- **cnts_ids_len** - the size of array of counters identifiers to reset.

Description

This function is used to reset the statistics for multiple single or class counters.

Return Value

The function returns 0 on success and an error code otherwise. The returned error code is the following:

- **EINVAL**, if invalid function arguments were provided;

9.9.3.4.2.4 Create or Remove Sampling Group

Table 266. API to Create or Remove Sampling Group

Function Name	Description	Input Parameters	Output Parameters
dpa_stats_create_sampling_group	Create a sampling group.	TBD	None.
dpa_stats_remove_sampling_group	Remove a sampling group.	TBD	None.

9.9.3.4.2.4.1 Function: dpa_stats_create_sampling_group

Syntax

TBD

Parameters

TBD

Description

This function will allow creating a sampling group for a number of counters. The purpose of a sampling group is to perform counters sampling at a specific frequency in order to assure counter rollover. By creating sampling groups, the application will always receive cumulative counters that take in account also the number of rollovers.

Return Value

TBD

9.9.3.4.2.4.2 Function: dpa_stats_create_sampling_group

Syntax

TBD

Parameters

TBD

Description

This function is used to remove a previously created sampling group.

Return Value

TBD

9.9.3.5 Network Function Layer

The *Network Function Layer* is a DPAA offloading driver extension which provides a service level API in Linux user space to enable NXP customers to easily configure hardware acceleration for specific standard network services on different NXP devices directly from their user space applications. The Network Function API is **platform agnostic** and is presently available for two families of NXP devices:

- the **Layerscape** family - LS2085A
- the **QorIQ** family equipped with DPAA 1.x network functions accelerator - P devices, B devices and T devices

On QorIQ devices the Network Function Layer (a.k.a. **NF Layer** is based on the current DPAA 1.x offloading drivers.

The Network Function Layer includes the following types of network sevices. The network services supported on DPAA 1.x devices are marked in the table below.

Table 267. Network Function Layer Services

Service Name	Supported on DPAA 1.x?
Firewall	
IP Forwarding (IPv4 and IPv6)	yes
IPSec	yes
Interface and Network Namespace Management	

Network Function API runtime services work identically across the entire list of supported NXP devices.

As far as the initialization is concerned, each NXP devices family has its specific mechanism for initializing its network hardware accelerator. Due to the major differences in design and concepts, NF API could not align the initialization across device families in a clear and easy-to-use way. As a consequence, each type of device has its own initialization sequence before starting to use NF API.

In QorIQ DPAA 1.x based devices, the offloading application / integration layer is in charge of the platform initialization. The following is a most commonly used initialization sequence.

1. Interpret PCD configuration file and create PCD model (using *fmclib*, for instance)
2. Initialize network interfaces (typically using *USDPAA*)
3. Initialize the acceleration infrastructure (queues, buffer pools, classification tables, etc.) (typically using *USDPAA*)
4. Update the PCD model with runtime information
5. Apply the PCD model and configure FMan (using *fmclib* and *fmlib*, for instance)
6. Provide initialization data to the NF Layer
7. Use the NF Layer for runtime network function services offloading configuration

9.9.3.6 References

Table 268. References

Index	Title
1	USDPAA PPAC User Guide
2	QMan/BMan API Guide
2	Frame Manager Configuration Tool Reference Manual
2	Frame Manager Configuration Tool: Examples

9.9.4 USDPAA PPAC User Manual

9.9.4.1 USDPAA PPAC Users Manual

The Packet Processing Application Core (PPAC), as a component of the USDPAA software framework that directly access queue manager (QMan) and buffer manager (BMan), helps develop and maintain common packet processing code between multiple USDPAA sample applications developed by NXP.

This document provides the following information.

- PPAC and PPAM detail.
- Layout of a PPAC/PPAM application.
- User controls to alter PPAC/PPAM application behavior.

Common packet processing code reduces maintenance costs, keeping common packet processing centralized and supporting updates done once rather than per application. Application-specific packet processing code is part of the application Packet Processing Application Module (PPAM). Together, the application specific PPAM and the common PPAC make up the complete application.

Users of USDPAA are not required to use the PPAC/PPAM model. Indeed, NXP only intends to extend PPAC capabilities to meet needs for sample USDPAA applications rather than for other generic usage.

This document assumes familiarity with the following concepts and documentation.

- QMan/BMan API Reference Manual
- USDPAA User Guide
- Linux user-space programming (POSIX, pthreads, etc.)

9.9.4.2 Overview of PPAC

The “reflector” application, developed as a small, stand-alone application for the USDPAA driver components, converted to a PPAC/PPAM application.

Origins of PPAC.

Early in USDPAA development, the “reflector” application was developed as a small, stand-alone application written on top of the USDPAA driver components. This application was eventually converted to a PPAC/PPAM application. Currently, there exists in the SDK another application, called hello_reflector, which is a new version of that stand-alone application.

the reflector application is very simple in terms of its packet processing logic (it flips Ethernet and IPv4 headers in regular IPv4 packets and forwards them back out the interface they arrive on, whilst discarding everything else), it facilitated and continues to facilitate the development, testing, and benchmarking of many generic aspects of packet-processing applications, not least of which are:

- parsing and enacting device configuration.
- driver initialisation.

Goals of the PPAC Users Manual.

With the extension of configurability and addition of features to the basic reflector application, and the requirement for other USDPAA packet-processing applications that share many of the same “general” requirements mentioned above, it was determined that the generic packet-processing application logic in reflector should be separated from the logic implementing the reflector-specific packet-processing “decision”.

Definition of PPAC and PPAM

The division of the packet-processing applications into a common, general infrastructure and application-specific components gives rise to the following two acronyms that now make up any PPAC-based application.

- PPAC: Packet-Processing Application Core. This is the generic application framework, implementing all of the “generic aspects of packet-processing applications” listed above. In the simplest case (the reflector application), this is essentially everything except the logic that flips the packet headers and selects the input interface as the forwarding destination. In particular, this includes the main() function .
- •PPAM: Packet-Processing Application Module This specialises PPAC into a “real application” by implementing the missing piece – namely the “packet-processing” specifics of the desired application. Each PPAM is compiled and linked with the PPAC component, but produces a stand-alone (and PPAM-specific) application binary.

Object Orientation

For some of the discussion that will follow, it may be helpful to use an object-oriented metaphor where PPAC is a base-class for packet-processing applications such that the packet-processing function is a pure virtual method (which is to say, non-existent and so must be implemented by derived classes). In this metaphor, PPAC-based applications are instantiations of derived classes, called PPAMs. Now all of this is implemented in C and the object-oriented metaphor should not be taken too literally, but it may help as an illustrative tool.

Limitation of Scope

PPAC supports a limited set of features; the application framework does not provide a solution for all conceivable USDPAA-based applications. PPAC provides as much consistency as possible for the packet-processing applications that are bundled with the USDPAA component of the SDK. That is, PPAC delivers a consistent user experience in both the look-and-feel and the elimination of code duplication. The scope and complexity of PPAC is intentionally limited to the current development, testing, and benchmarking requirements of USDPAA and the applications bundled with it. In particular, it is more important that PPAC (and PPAM) code be sufficiently comprehensible that application authors can understand how it works, than to provide a complete infrastructural solution for all potential USDPAA-based applications. With this in mind, please see the

"Chronology of a DPAA application" section, which attempts to provide pointers to key sections of PPAC code specifically to help with the development of non-PPAC applications.

Performance Challenges

Any abstraction underlying USDPAA applications needs to avoid imposing additional levels of function calls or other indirection. Eg. for 64 byte packets, when using between 1 and 4 CPUs on a p4080 DS, the peak processing rate of reflector application averages out to approximately 170 CPU cycles per-packet. An ad-hoc experiment to add just one extra level of indirection to that processing path introduced an additional overhead of ~20 cycles, causing a 12% performance degradation.

The same concern for function calls also applies to the data structure relationships between the PPAC and PPAM components. Use of pointers to relate discontiguous data-structures would necessarily lead to additional indirection and latency in the fast-path processing logic.

As it happens, splitting the fast-path logic of a data-path application into an abstraction layer (PPAC) and an application-specific layer (PPAM) whilst unifying the corresponding data-structures and without introducing any additional level of function calls is not impossible, but nor is it trivial.

9.9.4.3 Overview of PPAC Method and Implementation

the PPAC/PPAM interface is implemented by a strategic use of inlining.

Method

The reasons mentioned above should help explain why the PPAC/PPAM interface is implemented by a strategic use of inlining. It has been kept as simple as reasonably possible, but is clearly less straightforward than it would be if the afore-mentioned performance considerations were not a factor. The resulting implementation is organized in such a way that the compiler is able to inline the fast-path code of PPAC and PPAM together, as though they were written as a single (or "flat") application. As such, the PPAM version of reflector has no performance degradation relative to earlier non-PPAM versions, despite the PPAC/PPAM split. Note that all non-performance-critical code is compiled and linked conventionally within the PPAC library, without inlining. The application PPAM determines for itself how much code to implement within the inlining scope.

From both the data-structure and function-call points of view, the inlining relationship between PPAC and PPAM needs to resolve a circular dependency:

- PPAM within PPAC: the fast-path PPAC packet-processing logic needs to "compile in" the PPAM packet-processing logic – i.e. to have the compiler expand the PPAM-specific packet-processing logic "inline". Similarly the PPAC data-structure representing the interface and FQ state being handled needs to encompass any corresponding PPAM-specific state. I.e. the combined data-structures need to be declared statically rather than linked via indirection.
- PPAC within PPAM: the PPAM packet-processing logic will invariably need to issue one or more enqueue/transmit operations (or a release/drop operation) as a result of its packet-processing decision, and these operations must necessarily be coordinated by PPAC (otherwise generic mechanisms like buffer-management, flow-control, [etc] cannot be coordinated). So any PPAC-provided functions to be used by the PPAM packet-processing logic will need to be expanded "inline".

Thus, for at least a subset of the PPAC and PPAM code, there will be two layers of inlining; PPAC within PPAM within PPAC. The following diagram shows the behaviour of a PPAC-based application where maximum inlining of the packet-processing code is used (this is the case with the reflector application).

By way of example, the reflector application presents a very minimal PPAM component, essentially the "header-flipping" function, and so it presents only the one C source file that is used to inline the PPAC machinery and compile within that "inlining" scope. The IPFwd application on the other hand implements significant components of a network stack and routing logic, so much of its logic (including some that is used in the packet-processing fast-path) is conventional C code without inlining. The fact remains that the PPAC abstraction itself does not contribute any additional functional indirection, in order to retain the property of providing no performance overhead relative to an equivalent application written without the use of such an abstraction.

The following diagram illustrates a PPAC-based application where only a subset of the packet-processing code is inlined. This example proposes that each FQ carries packets for one or more network flows (due to classification and hashing in FMan), and that the PPAM associates each such FQ with state to implement a "recently-processed flow cache" (or "RPFC" as we'll call it for the purposes of this illustration), which in its simplest form would be a single-entry. It's worth noting that if

network traffic is classified/hashed into multiple Rx FQs, then the probability increases that consecutive packets on a given FQ are from the same flow. If the packet matches a flow in the RPFC then packet-processing will complete entirely within the inlined/optimised code. Furthermore, if FQ context-stashing is enabled, then this RPFC state would be pre-positioned in the CPU's L1 or L2 cache and so packet-processing would complete without any memory latencies. The “slow path” is the case where the packet does not match in the RPFC, the processing of the packet will then involve code that is not inlined into the packet-handler, and will likewise use state that is not necessarily present in processor cache.

Implementing a New PPAM

The upcoming sections provide a guide to the noteworthy PPAC files, data-structures, and interfaces, and also enumerate those required of a minimal PPAM application. Additionally, the “reflector” application should be used as a reference example to understanding this interface and for starting out new PPAM applications, and indeed could be copied and modified to begin doing such work due to its simplicity.

As was mentioned at the start of this document, such descriptions are provided to enable creation of quick sample applications that require the same common packet processing as other USDPAA sample applications. The following sections also help users understand the implementation of PPAC/PPAM sample applications so that they can extract any useful capability there into their own non-PPAC application.

About the Upcoming Document Sections

In the upcoming sections of this document, the PPAC/PPAM specifics will be introduced in the following order;

- PPAC files: this section will document the files that make up PPAC and describe their characteristics.
- PPAM files: this section documents the minimal set of files that a PPAM (i.e. PPAC-based application) needs to provide.
- Packet-processing data structures: this section documents the data types used in PPAC-based applications, which necessarily includes the sub-types to be implemented by PPAMs.
- PPAM-provided functions: this section describes how PPAM can implement its data-types and the corresponding handler functions that are invoked by the PPAC infrastructure.
- PPAC-provided functions: this section describes those PPAC functions that can be called from within PPAM implementations. Most interestingly, this describes the interfaces available to PPAM packet-handling logic for the transmitting or dropping of frames.

9.9.4.4 PPAC Files

The PPAC abstraction comprises the following five (5) files.

File: apps/include/ppac_interface.h

This header declares the network interface structure, “struct ppac_interface”, which combines the PPAC and PPAM structures into one, thus it requires that the PPAM declarations already be defined. This type encompasses all Rx and Tx FQs.

File: apps/include/ppac.h

This header declares a variety of definitions required by both PPAC and PPAM logic, which includes:

- pre-compiler symbols controlling parameters and options for PPAC compilation.
- pre-declarations of PPAM functions to be called by PPAC code.
- pre-declarations of PPAC functions to be called by PPAM code.
- pre-declarations of PPAC functions called between the code that is inlined into PPAM and the code that is compiled in to the PPAC library.
- PPAC inline code that needs to be available at a wider scope than that of apps/include/ppac.c.
- pre-declarations of global variables required by inline functions.
- pre-declarations of global variables representing weakly-linked PPAC constants (ie. that a PPAM can override).

File: 3.3apps/include/ppac.c

This “header” contains the fast-path PPAC code that needs to be expanded/inlined with the corresponding fast-path PPAM code. It also contains any other code that needs to exist at the same scope as the fast-path code. The main characteristic of this file is that it must be included exactly one time, by the PPAM code, and it must be included after PPAM has declared its own fast-path hooks (the included PPAC fast-path logic calls these PPAM hooks). This file includes:

- PPAC-defined QMan callbacks for handling dequeued frames from each class of Rx FQ. (These invoke PPAM-specific hooks which are thereby inlined into a single function by the compiler.)
- the interface manipulation code (setup, teardown, enable, disable) that binds these fast-path callbacks to their corresponding FQ objects and provides fast-path as well as setup and teardown hooks to PPAM.

File: apps/ppac/main.c

This file contains the PPAC code that can be compiled independently of PPAM and made available as a linkable library (libusdpaa_ppac.a). This includes:

- global constants and variables that are not required directly from inline fast-path logic.
- FQ, buffer pool, and CGR manipulation code (setup, teardown, etc).
- thread management (including IPC).
- buffer pool management.
- global setup and teardown (invokes the per-interface setup/teardown logic contained within apps/include/ppac.c)

File: apps/ppac/ppac.lds

This file is a linker script to be used when compiling PPAM applications. The CLI implementation in PPAC allows both PPAC and PPAM code to declare and implement commands to be added to the interface, and the underlying definitions compile these into a dedicated linker section. By linking the resulting application with this linker-script, the location and length of this linker section is known to PPAC CLI code. (The alternative to using this approach is to have PPAC and PPAM dynamically build a list of CLI command handlers, which requires a coordinated initialisation phase, whereas this linker-based mechanism is static.)

Use of this linker script is shown in the PPAM section covering <app-dir>/Makefile.am.

9.9.4.5 PPAM Files

The PPAM application may be composed of as little as one C file, and of course may be significantly more complex. The main requirement is that exactly one C file include the PPAC inline machinery contained in apps/include/ppac.c.

Other C files may include the other headers in order to share PPAC definitions as widely as required.

The three (3) files listed here correspond to those implemented by the reflector PPAM (which uses two files in order to separate some definitions into a header for the sake of clarity). A far more elaborate PPAM example can be found in the case of the IPFwd PPAM.

File: <app-dir>/ppam_interface.h

This file is defined by reflector and IPFwd PPAMs for the sake of clarity - there is no technical or other reason for such a header to exist separately from any other PPAM files. In reflector, this header contains the PPAM-specific data-structures that are pre-requisites for the definition of PPAC data-types contained in ppac_interface.h. i.e. the PPAM code includes ppam_interface.h prior to including ppac.c (which is what requires and includes ppac_interface.h). PPAM code can just as easily implement the same definitions elsewhere, so long as they are defined prior to the one-off inclusion of ppac.c.

File: <app-dir>/*.c

This section refers to the primary PPAM C file that includes the PPAC inline machinery. It must ensure that the required data-structures are defined prior to the inclusion of ppac_interface.h, and it must ensure that the required PPAM hooks are declared before inclusion of ppac.c. In the case of reflector, the data-structures are defined in ppam_interface.h as explained above, and apps/reflector/reflector.c defines and implements the PPAM hooks as inline functions prior to including ppac.c. (They only need to be pre-declared before inclusion of ppac.c if you prefer to implement them afterwards. If you don't wish to use inline functions to save function jumps, then they can be implemented in other C files and resolved by the linker.)

The PPAM must also define the symbols that are required by PPAC in order to link, though this does not necessarily need to be done within the primary PPAM C file that is including the PPAC inline machinery. In the reflector case however, these symbols are in fact instantiated in the same C file, for simplicity's sake. These symbols relate to the CLI, and as reflector adds no commands to the base commands implemented within PPAC, so the required PPAM symbols have trivial implementations.

File: <app-dir>/Makefile.am

This section presumes that the PPAM application is being built by the USDPAA build system. This need not be the case, but an external build-system would still need to address the same requirements in its own way. The following definitions are used to build and link the reflector application;

```
bin_PROGRAMS = reflector

AM_CFLAGS := -I$(TOP_LEVEL)/apps/include

reflector_SOURCES := reflector.c
reflector_LDADD := usdpaa_ppac usdpaa_syscfg usdpaa_qbman usdpaa_fman \
                   usdpaa_dma_mem usdpaa_of
reflector_LDFLAGS := $(LIBXML2_LDFLAGS) $(LIBEDIT_LDFLAGS) \
                     -T $(TOP_LEVEL)/apps/ppac/ppac.lds
```

The key points here are:

- the PPAC logic contained in apps/include/ needs to be accessible via includes of <ppac.c>, <ppac.h>, and <ppac_interface.h>.
- the application needs to link with libusdpaa_ppac.a, and by dependence needs to link with the other named libraries that provide USDPAA configuration parsing and driver functionality.
- the linker flags need to provide any dependencies for linking with Gnome Libxml2. In the SDK containing USDPAA, this is set to point to the installation directory for system libraries and to resolve with the “xml2”, “z” (zip), and “m” (math) libraries.
- the linker flags need to provide any dependencies for linking with Editline. In the SDK containing USDPAA, this is set to point to the installation directory for system libraries and to resolve with the “edit”, and “curses” (ncurses) libraries.
- the ppac.lds linker script should be used to add the array-delimitation definitions required by the PPAC CLI implementation.

When implementing a new PPAM application, the above model can be followed – simply create a new sub-directory within the build-tree, add that directory's name to the SUBDIRS definition in parent directory's Makefile.am file, and copy-paste-and-modify the above “reflector” definitions to your requirements:

- if your directory is in fact to host sub-directories with their own libraries and/or applications, then add a SUBDIRS definition to your Makefile.am file (for further information about this, look at the IPFwd reference application).
- change all occurrences of “reflector” to the name of your desired application (or if your directory is only going to host sub-directories with their own targets, then remove everything except the SUBDIRS definition described in the preceding point).
- change the “..._SOURCES” attribute to describe the C files that need to be compiled. The above example for reflector only lists one C file because that's all it contains, but less trivial applications will have more (the list should be space-separated, with back-slashes for multi-line entries).
- Extend “..._LDADD” and “..._LDFLAGS” attributes if required.

9.9.4.6 Packet-processing data structures

A descriotion of the data-structures that are used within PPAC (and PPAM) for handling network interfaces and the FQs they contain.

Within each such PPAC type, the data-structure contains a PPAM-specific equivalent, allowing PPAM to “sub-class” PPAC types for its application-specific purposes (to use the object-oriented metaphor mentioned in section 2.2.2). In section 6, these PPAM types will be described, as will the handler functions PPAM can provide for them (i.e. the “pure virtual methods” that can be implemented in “derived classes”, to continue the object-orientation imagery).

File: struct ppac_interface

(Defined in apps/include/ppac_interface.h.) This structure represents a network interface and associated state. In keeping with the object-oriented metaphor, this structure includes a PPAM-defined structure within it, “struct ppam_interface”, which the PPAM can use to define and maintain interface-wide state of its own.

This structure contains the following state:

- a pointer to information about the configuration of the FMan ethernet port corresponding to this “interface”.
- a PPAM-defined “struct ppam_interface” object for use by PPAM processing.
- an array of QMan FQ objects representing the Tx FQs for the interface.
- objects representing the unique Rx FQs associated with the interface . These structures are described subsequently.
- a linked-list of PCD ranges, each represented by the "struct ppac_pcd_range" type. Each PCD range is an array of objects representing the "hashed" Rx FQs for a policy of the interface. In high-performance datapath applications, FMan distributes most traffic to this range of FQs based on parse and classification processing.

File: struct ppac_rx_{error|default|hash}

These structures represent individual FQs that FMan enqueues frames to during Rx processing and that are dequeued to software portals for application processing. The “error” FQ receives frames due to any errors in FMan Rx processing, otherwise frames are either enqueued to one of many “hash” FQs (due to FMan parse and classify processing) or to the “default” FQ. For each structure type, the PPAC structure includes a PPAM-defined structure within it, “struct ppam_rx_*”, which the PPAM can use to define and maintain FQ-wide state of its own.

These structures contain the following state:

- the “struct qman_fq” object used with the QMan driver for manipulating the hardware FQ object as well as handling dequeued frames. The nature of the QMan driver API is such that this object is also the “FQ context” that gets stashed to processor cache if the FQ is configured for any stashing at all. This object is intentionally the first element in the per-FQ PPAC structure, because the application can control how many cache lines of “FQ context” should be stashed during dequeue operations, so any adjacent state following this object can also benefit from dequeue stashing.
- a PPAM-defined “struct ppam_rx_*” object for use by PPAM processing. As mentioned in the previous item, QMan dequeue processing can be configured to stash this state to processor cache during the dequeue operation in hardware, potentially avoiding or minimising the possibility for cache-misses in fast-path processing.

File: struct ppac_tx_{error|confirm}

These structures are similar to those described for “struct ppac_rx_*”, so the details will not be repeated here. The key difference is that these FQs are enqueued to by FMan during Tx processing and so represent the consequence of an attempt by software to forward already-processed traffic, rather the consequence of Rx processing of frames that software has not yet seen.

The “error” FQ naturally will receive any frames where FMan or QMan encountered an error during transmit (implying that the packet was not transmitted successfully).

The “confirm” FQ is intended to return frames back to software that have been successfully transmitted, but this is not a mode of operation that PPAC currently enables (nor are there currently any options to turn this on).

For current PPAC application purposes, all transmitted frames are composed of buffers sourced from BMan pools, and so the Tx FQs are configured to make FMan autonomously release all buffers from the transmitted frames back to the pools they belong to. Use of a “confirm” FQ is unnecessary here and would just degrade system performance. If an application is going to transmit traffic from buffers that do not originate from BMan buffer pools, then it may be necessary to reconfigure the Tx FQs to use the confirmation feature.

9.9.4.7 PPAM-provided Functions

PPAC/PPAM-based applications are driven by PPAC itself which implements the main() function as well as the run-to-completion functions that execute in the worker threads.

Here are listed the functions that can be implemented by a PPAM application. Application-specific behaviour of PPAM applications is determined by the way in which PPAMs implemen the interfaces that PPAC expects to exist. A subset of these interfaces are performance-critical, so as previously described they can be implemented so as to benefit from compiler inlining

When considering the object-oriented metaphor, these PPAM-provided functions correspond to pure virtual methods in the PPAC base-class that the PPAM derived class should implement

Many of these functions naturally map to the PPAM-specific structures that are embedded in each of the corresponding PPAC data structures defined in the previous section, so will be documented in the same sequence. The exceptions are the “global initialisation” and “polling” handlers, which are not scoped to any particular interface or FQ. (In object-oriented language, these latter handlers would be considered “class methods”.)

Global Initialisation

The handlers described in this section are called when initialising and cleaning the application process itself and the individual worker threads created within that process. The PPAC library declares weakly-linked versions of these interfaces, so a PPAM is not required to implement its own versions unless it needs such hooks (though if it does the PPAM-provided versions will be called instead of the PPAC-provided fallbacks).

Process Initialization and Cleanup

The “init” function is called when the application starts up (prior to creation of any network interface structures or any worker threads) and the “finish” function is called when the application is exiting (after all worker threads and network interfaces are destroyed). Any state managed by these hooks should be implemented using (non-thread-local) global variables.

```
int ppam_init(void) ;
void ppam_finish(void);
```

If ppam_init() returns non-zero, that is considered failure and application initialisation will be abandoned.

Worker Thread Initialisation and Cleanup

These functions are called when a worker thread starts up or is being destroyed. Any state managed by these hooks should be implemented using thread-local global variables, i.e. using the “__thread” gcc attribute.

```
int ppam_thread_init(void);
void ppam_thread_finish(void);
```

As usual, if the initialisation function returns non-zero, that is considered failure and so creation of the worker thread will be abandoned. Note also that thread-local QMan/BMan portals are already initialised when ppam_thread_init() is called and are not destroyed until after ppam_thread_finish() has returned, so it is safe to use portal-dependent QMan/BMan APIs within these functions.

Polling Handlers

The run-to-completion loop of each worker thread (implemented by PPAC) can optionally call a PPAM-provided polling function to allow PPAM applications t perform general-purpose processing. E.g. to implement background processing tasks, generate frames for transmit that are not in reaction to a received frame (IPC), etc. As with the global initialisation handlers, PPAC defines a weakly-linked version of this polling function, so PPAMs are not required to implement it unless they need it. There is also a thread-local global variable that PPAM code can be set non-zero whenever it wishes the run-to-completion loop to call this polling handler.

```
extern __thread int ppam_thread_poll_enabled;
int ppam_thread_poll(void);
```

If ppam_thread_poll_enabled is zero (the default), the ppam_thread_poll() function is not called from within the run-to-completion loop (minimising overhead in the case that no polling hook is required). Note that the weakly-linked version of

ppam_thread_poll() implemented by PPAC will intentionally kill the application, because it is illegal to set ppam_thread_poll_enabled non-zero unless the PPAM application implements its own ppam_thread_poll() function.

File: struct ppam_interface

This is the PPAM-specific object representing a network interface. It and its PPAC-wrapper form the “parent” for the objects that representing the individual FQs A PPAM must implement the following hooks for this object;

Initialisation

This function is called as the network interface is initialised but prior to all Rx or Tx FQ objects being initialised, so this hook is called before all ppam_rx_*_init() or ppam_tx_*_init() hooks are called for object that belong to the this interface. The parameters provide a pointer to the PPAM-specific interface state (which is uninitialised on entry and should be initialised to the PPAM's requirements), a pointer to the configuration information for this network interface, and also indicates to PPAM the numbe of Tx FQs that will be initialised for this interface;

```
int ppam_interface_init(struct ppam_interface *p,
                        const struct fm_eth_port_cfg *cfg,
                        unsigned int num_tx_fqs);
```

If this function returns non-zero, it will be interpreted as an error code and initialisation of the interface will be abandoned.

Cleanup

This function is called as the network interface is being cleaned up and after all Rx and Tx FQ objects have been destroyed, so this hook is called after all ppam_rx_*_finish() and ppam_tx_*_finish() hooks are called for objects that belong to this interface.

```
void ppam_interface_finish(struct ppam_interface *p);
```

Tx FQ Enumeration

The ppam_interface_init() function described earlier notifies the PPAM of the number of Tx FQs that will be initialised for this interface. So the purpose of this function is to notify the PPAM as each individual Tx FQID is (dynamically) allocated and initialised for this purpose. I.e. the PPAM can use the initial ppam_interface_init() hook to know how many Tx FQs the interface will have (e.g. in order to allocate a sufficiently large array, or to verify that an existing static array is large enough) and then use the subsequent ppam_interface_tx_fqid() hooks to obtain each such FQID. This process preceeds all the Rx FQ hooks described below.

```
void ppam_interface_tx_fqid(struct ppam_interface *p,
                            unsigned idx,
                            uint32_t fqid);
```

9.9.4.8 PPAM Rx and Tx

These are the PPAM-specific objects representing the Rx-related FQs of a network interface. A PPAM must implement the following hooks for these objects;

PPAM Rx Initialization

These functions are called just prior to the corresponding Rx-related FQ being initialised by PPAC. The parameters provide a pointer to the PPAM-specific FQ state (which is uninitialised on entry and should be initialised to the PPAM's requirements), a pointer to the PPAM-specific state for the interface this FQ belongs to (which will have already been initialised by the PPAM's own ppam_interface_init() function), and a pointer to the QMan dequeue-stashing configuration that will be used to initialise the FQ (the PPAM can modify this structure before returning in order to modify the stashing configuration to be applied to the FQ). Note that the “hash” function also provides an index parameter, as the “rx_hash” FQs form arrays called "PCD ranges". If the PPAM logic needs to know the number (and order) of these ranges, and the number of "hash" FQs within each of them, it can determine that from the ::list field of the "cfg" parameter passed to the ppam_interface_init() hook.

```
int ppam_rx_error_init(struct ppam_rx_error *p,
                       struct ppam_interface *_if,
```

```
                        struct qm_fqd_stashing *stash_opts);
int ppam_rx_default_init(struct ppam_rx_default *p,
                         struct ppam_interface *_if,
                         struct qm_fqd_stashing *stash_opts);
int ppam_rx_hash_init(struct ppam_rx_hash *p,
                      struct ppam_interface *_if,
                      unsigned idx,
                      struct qm_fqd_stashing *stash_opts);
```

If these functions return non-zero, it will be interpreted as an error code and initialisation of the interface will be abandoned.

PPAM Rx Cleanup

These functions are called as the network interface is being cleaned-up, each such hook is called just prior to the corresponding FQ object being destroyed. The interface-wide ppam_interface_finish() hook is last to be called once all FQs belonging to the interface have been cleaned up. The parameters match those to the corresponding _init() functions, with the exception that no stashing configuration is provided (because the FQ is being destroyed, not initialised).

```
void ppam_rx_error_finish(struct ppam_rx_error *p,
                          struct ppam_interface *_if);
void ppam_rx_default_finish(struct ppam_rx_default *p,
                            struct ppam_interface *_if);
void ppam_rx_hash_finish(struct ppam_rx_hash *p,
                         struct ppam_interface *_if,
                         unsigned idx);
```

PPAM Rx Packet Processing

These functions essentially represent the PPAM's “fast-path”, particular in the “hash” case. When processing dequeued frames, the QMan driver invokes the callback associated with the FQ object, which is implemented by PPAC as part of the “inlined” component of the code. These PPAC-implemented callbacks handle any “infrastructure” responsibilities for processing the dequeued frame and the FQ it was dequeued from (e.g. providing support for order preservation or restoration, flow-control, etc), but they defer to these PPAM-specific packet-processing hooks to examine the packets and determine what to do with them. The parameters provide a pointer to the PPAM-specific FQ state, a pointer to the PPAM-specific state for the interface this FQ belongs to (except in the “hash” case), and a pointer to the QMan DQRR entry containing the frame descriptor and other status about the dequeue operation for the frame (and the FQ it was dequeued from, e.g. whether the dequeue left empty).

If these hooks are implemented as inlines, then the compiler is able expand the PPAC and PPAM packet-processing logic into a single function layer, and many also make other optimisations possible (such as not having to prepare and pass parameters to the PPAM hooks if they're unused, etc).

```
void ppam_rx_error_cb(struct ppam_rx_error *p,
                      struct ppam_interface *_if,
                      const struct qm_dqrr_entry *dqrr);
void ppam_rx_default_cb(struct ppam_rx_default *p,
                        struct ppam_interface *_if,
                        const struct qm_dqrr_entry *dqrr);
void ppam_rx_hash_cb(struct ppam_rx_hash *p,
                     const struct qm_dqrr_entry *dqrr);
```

Note that the “hash” callback hook does not provide the PPAM-specific state for the interface, nor does it provide the index of the hash callback. These two omissions are primarily justified by the observation that this is the most performance-defining code-path and so should not include any potentially-unnecessary overheads, whether or not the packet-processing is fully inlined. Also, the “error” and “default” FQs are global to the interface and so packets that arrive on them represent events that are in some way “exceptional”, whereas packet-processing on the “hash” FQs should be the normal case and less likely to depend on interface-global state. If this is not the case and the “hash” callback does require the interface state and/or the index, then that state should be added to the “struct ppam_rx_hash” type and initialised during the ppam_rx_hash_init() hook,

ensuring that it does not need to be “computed” by PPAC for passing as a parameter, and may also benefit from stashing of the “struct ppam_rx_hash” structure during dequeue operation.

It is the responsibility of these callbacks to determine what to do with the frame and in doing so should commit to the corresponding action before returning. The APIs available for this are described in a later section called “PPAC-provided functions”, but in essence they provide “drop” and “forward” mechanisms.

PPAM Tx

As with the description of these data-structures in the previous section (“Packet-processing data structures”), the PPAM-specific functions for them are very similar to those described for “struct ppac_rx_*” and so will not be repeated here. As mentioned in the data-structure discussion, the key difference is that these FQs are enqueued to by FMan during Tx processing and so represent the consequence of an attempt by software to forward already-processed traffic, rather the consequence of Rx processing of frames that software has not yet seen. Please see that data-structure discussion for more specifics.

PPAM Tx Initialization

```
int ppam_tx_error_init(struct ppam_tx_error *p,
                       struct ppam_interface *_if,
                       struct qm_fqd_stashing *stash_opts);
int ppam_tx_confirm_init(struct ppam_tx_confirm *p,
                         struct ppam_interface *_if,
                         struct qm_fqd_stashing *stash_opts);
```

PPAM Tx Cleanup

```
void ppam_tx_error_finish(struct ppam_tx_error *p,
                          struct ppam_interface *_if);
void ppam_tx_confirm_finish(struct ppam_tx_confirm *p,
                            struct ppam_interface *_if);
```

PPAM Packet Processing

```
void ppam_tx_error_cb(struct ppam_tx_error *p,
                      struct ppam_interface *_if,
                      const struct qm_dqrr_entry *dqrr);
void ppam_tx_default_cb(struct ppam_tx_confirm *p,
                        struct ppam_interface *_if,
                        const struct qm_dqrr_entry *dqrr);
```

9.9.4.9 PPAC Provided Functions

Describes the functions that PPAC makes available for use from PPAM application code.

PPAC Packet Processing

The PPAM packet-processing hooks must adhere to a simple set of rules when processing dequeued frames in order to ensure sane functioning of the system. For every packet examined, exactly one of two mutually-exclusive actions must be taken prior to returning from the packet-processing hook; “drop” or “forward”. (The “forward” APIs are called “send”, for the sole reason that the latter has fewer letters.) The “drop” case is very simple, while the “forward” case presents some options..

Note that these APIs assume that no intermediary processing of the frame is required, and in particular do not prescribe any way of dealing with accelerator hardware (cryptographic, pattern-matching, or otherwise). See section 7.2 for more information.

PPAC Packet Processing - Drop a Frame

If a PPAM packet-processing hook has examined a frame and determined that it should not be transmitted, it must drop it using the following API. The parameter is the frame-descriptor to be dropped, which would usually be “&dqrr->fd”, where “dqrr” is the parameter passed to the packet-processing hook.

```
void ppac_drop_frame(const struct qm_fd *fd);
```

Note that this function is actually implemented as part of the PPAC inline machinery, and so just as the PPAM hook can be inlined into the PPAC packet-processing code, this “drop” action will likewise be inlined into the processing chain, and permit the compiler to look for any possible optimisations.

PPAC Packet Processing - Forward a Frame

If a PPAM packet-processing hook has examined a frame and determined that it should be forwarded, it must use the following API exactly once. The parameters specify the frame-descriptor to be forwarded (see the ppac_drop_frame() description above for more info about this parameter), as well as the FQID of the Tx FQ to which it should be forwarded. Please refer back to the “PPAM-provided functions”, and specifically the “Tx FQ enumeration” subsection to recall how a PPAM is notified of the Tx FQIDs for each of the network interfaces. This information provided during initialisation should be organised by the PPAM implementation in order for it to be able to determine target FQIDs when processing packets.

```
void ppac_send_frame(u32 fqid, const struct qm_fd *fd);
```

As with dropping a frame, this function is also implemented by PPAC as an inline, ensuring that it can be compiled (and potentially optimised) into the packet-processing logic.

PPAC Packet Processing - Forwarding Multiple Frames

If a frame needs to be forwarded multiple times (e.g. for multicast), ppac_send_frame() must still be called exactly once. Supplementary transmit actions are achieved by using the following API;

```
void ppac_send_secondary_frame(u32 fqid, const struct qm_fd *fd);
```

The parameters are the same as for ppac_send_frame().

PPAC Packet Processing - Dealing with Accelerators

As mentioned in the previous section, the current packet-processing functionality in PPAC provides no support for accelerators. It is likely that future versions of PPAC will provide something in this regard, but for now interactions with accelerators is presumed to be outside the scope of the PPAC infrastructure, and so PPAMs will communicate with accelerators via their driver interfaces. The upshot of this is that any flow-control or other infrastructural support within PPAC will be “blind” to frames and frame queues going between the application and the accelerator hardware.

If a PPAM packet-processing function chooses not to make a drop-or-forward decision immediately, then it must do so at a later time in order to avoid leaking the frame (and all buffer resources associated with it). This delayed transmit action will presumably occur from within the dequeue callback for the FQ that brings the accelerator's response back from the accelerator.

PPAC Packet Processing - Accelerator Limitations

The use of accelerators, or indeed any other deferral by the PPAM packet-handler to issue a drop-or-forward action “at a later time” infers some limitations on PPAC functions.

- ORP (Order Restoration) support must be disabled in PPAC, the infrastructure for ORP relies on the drop or forward PPAC interfaces being called from within the PPAM's packet-handling function.
- Order-preservation support must also be disabled in PPAC, for much the same reasons as for ORP.

PPAC Packet Processing - Application Generated Frames

It is possible for applications to generate new frames for transmission rather than strictly forwarding received frames. E.g. the polling hooks available to PPAM (see section 6.2) might be used by the PPAM application to generate test traffic, send work-items to accelerators, respond to IPC requests from other threads or applications, or manage the migration of traffic between USDPAA and non-USDPAA interfaces (e.g. wireless). The limitations described in section 7.2.1 with respect to

accelerators apply here if the transmissions are to go out the USDPAA-managed network interfaces. Moreover, if the transmissions are targeted to those network interfaces, then the frame descriptors must reference buffers that are sources from BMan buffer pools (as the network hardware will autonomously release them back to BMan post-transmission).

PPAC Packet Processing - Application Terminated Frames

Frames that are received via the PPAC-provided packet-handling interface may be fully-consumed by the PPAM application, the assumption is not that they are strictly for forwarding. However, the current PPAC implementation does present some limitations in this regard;

- the frames that are not forwarded must instead be "dropped" from the PPAC perspective (by calling the ppac_drop_frame() function), meaning that the BMan buffers within them will be deallocated back to their respective buffer pools.
- if the application intends to use the packet contents after the packet has been "dropped", its contents must be copied to other memory before doing so. (It is possible that future PPAC enhancements will provide a means to avoid this limitation.)
- if the application intends to delay dropping the frame until it has finished with it, meaning that it will not be dropped before returning from the packet-handling routine, then the same limitations that are mentioned in section 7.2.1 apply here (for the same reasons).

9.9.4.10 PPAC Setting

PPAC-based applications are compiled using a certain set of options that are, for the most part, defined in the header located at apps/include/ppac.h. The following sections describe the most useful options for modification, if alternative application behaviour is desired.

PPAC Order Preservation

Order preservation is a functionality of the QMan software portal interface that allows processing of Rx FQs across multiple portals to retain order when transmitted corresponding Tx FQs. The technique only applies to frames dequeued from a given Rx FQ that are all transmitted out the same Tx FQ (which is the case for "reflector", and also the case for "IPFwd" when frames are from the same flow). The mechanism requires two QMan features, "HOLDACTIVE" and "enqueue DCA". The former ensures that a FQ that has been dequeued to a software portal from should remain bound to that portal until all the corresponding DQRR entries have been consumed. The latter ensures that a DQRR entry is consumed by QMan itself once it has dispatched the corresponding enqueue (Tx) command. Together, for any given Rx/Tx FQ pair, the processing via pool-channels and multiple CPUs does not allow frame processing to get out of order. (For more information on this feature, consult the QMan/BMan API Guide.)

Use of "HOLDACTIVE" is mutually exclusive with another QMan option "AVOIDBLOCK", which is selected by default in PPAC. So to enable order-preservation, one must change the settings from :

```
#undef PPAC_HOLDACTIVE
#undef PPAC_ORDER_PRESERVATION
#define PPAC_AVOIDBLOCK
to;
#define PPAC_HOLDACTIVE
#define PPAC_ORDER_PRESERVATION
#undef PPAC_AVOIDBLOCK
```

PPAC Order Restoration

Order restoration is a feature of the QMan hardware that allows frame enqueue operations to be re-assembled in an ORP ("Order Restoration Point") prior to enqueuing onto the destination FQ. (ORPs are in fact FQ objects in hardware that are configured for use as restoration contexts). The subject of ORP (and its various behavioural parameters) is beyond the scope of this document. Please consult the QMan Reference Manual for more detail.

To enable order restoration, one must change the setting from;

```
#undef PPAC_ORDER_RESTORATION
to;
#define PPAC_ORDER_RESTORATION
```

PPAC Monitoring Rx/Tx Fill-Levels via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. The CGRs are not configured to perform any flow-control (i.e. no tail-drop nor WRED options are enabled), so this simply allows the user to monitor the overall fill-level of frame queues in the system, in particular to determine whether build-up is occurring before or after the software-processing phase. The thresholds used for the CGRs are determined from the two constants also defined in ppac.h, PPAC_CGR_RX_PERFQ_THRESH and PPAC_CGR_TX_PERFQ_THRESH. These constants, combined with the number of FQs, define the thresholds for CGR congestion-entry. QMan in turn defines the CGR exit-threshold to be 7/8 the entry-threshold (to provide hysteresis), and so these combined entry/exit events will be logged to stdout independently for the Rx and Tx CGRs.

An extra command, “cgr”, becomes available in the CLI when this feature is compiled in, which will query and display all the fields of both CGRs. Note however that this option introduces extra latency, and more critically, extra contention within the system. This is because QMan must lock the CGR for each enqueue and dequeue event that relates to it, meaning that the forwarding of a single packet through the system requires 2 lock/unlock pairs for each CGR (thus 4 lock/unlock pairs). A small but noticeable performance degradation should be expected when running in this mode. (Real-world use of CGRs would not subscribe all Rx/Tx FQs from all interfaces to a single CGR, so this scalability issue should not be a concern for production software usage of CGR-based congestion management.)

To enable this feature, change:

```
#undef PPAC_CGR
to;
#define PPAC_CGR
```

PPAC Other Settings

Many other settings used by PPAC (and thus PPAC-based apps) are defined in ppac.h but they are less intended for ad-hoc manipulation than those mentioned above. However a curious user may wish to examine some of these, and perhaps search out their usage within the source-code, in order to see how they are used and explore some of the driver interfaces and application design in this way.

9.9.4.11 PPAC Buffers

When starting up, PPAC applications seed the 3 buffer pools that the FMan is configured to use for Rx processing.

To improvement restartability, the pools are first drained of any stale contents, after which they are seeded using allocations from the “/dev/fsl_usdpaa_shmem” DMA device, which is described in the USDPAA User Guide. The buffer pool IDs that must be used, and the size of the buffers they must each hold, is required to match the settings in FMan – these are currently hardcoded in the include/internal/conf.h header, and must be kept in-sync with the FMan settings.

The three buffer pools initialized and used by PPAC (and the default USDPAA configuration of FMan) have the following attributes;

Table 269. Buffer Pool attributes

Buffer Pool ID (BPID)	Buffer Size (used by Fman)	Number of Buffers
7	320	0
8	704	0
9	1728	0x2000

When processing frames, buffers are allocated by FMan on Rx and then released by FMan after Tx.

The current default buffer pool settings from include/internal/conf.h follow;

```
#define DMA_MEM_PATH            "/dev/fsl-usdpaa-shmem"
#define DMA_MEM_BP1_BPID        7
#define DMA_MEM_BP1_SIZE        320
#define DMA_MEM_BP1_NUM         0 /* 0*320==0 (0MB) */
#define DMA_MEM_BP2_BPID        8
#define DMA_MEM_BP2_SIZE        704
#define DMA_MEM_BP2_NUM         0 /* 0*704==0 (0MB) */
#define DMA_MEM_BP3_BPID        9
#define DMA_MEM_BP3_SIZE        1728
#define DMA_MEM_BP3_NUM         0x2000 /* 0x2000*1728==13.5MB */
#define DMA_MEM_BPOOL \
        (DMA_MEM_BP1_SIZE * DMA_MEM_BP1_NUM + \
        DMA_MEM_BP2_SIZE * DMA_MEM_BP2_NUM + \
        DMA_MEM_BP3_SIZE * DMA_MEM_BP3_NUM) /* 13.5MB */
```

And the application implements the seeding of these buffers according to the following structure in apps/ppac/main.c;

```
/* Seed buffer pools according to the configuration symbols */
const struct ppac_bpool_static {
        int bpid;
        unsigned int num;
        unsigned int sz;
} ppac_bpool_static[] = {
        { DMA_MEM_BP1_BPID, DMA_MEM_BP1_NUM, DMA_MEM_BP1_SIZE},
        { DMA_MEM_BP2_BPID, DMA_MEM_BP2_NUM, DMA_MEM_BP2_SIZE},
        { DMA_MEM_BP3_BPID, DMA_MEM_BP3_NUM, DMA_MEM_BP3_SIZE},
        { -1, 0, 0 }
};
```

9.9.4.12 Chronology of a DPAA application

Provides pointers to key locations in the PPAC code where tasks are performed that any USDPAA-based application would need to implement, in particular for applications that will not be based on PPAC or anything similar.

For more information on each of the named APIs, such as parameters, return codes, semantics, [etc], please consult the relevant documentation (e.g. the “USDPAA User Guide”, the “Queue Manager, Buffer Manager API Reference Manual”, etc.).

DPAA Application - Global Initialization

This section covers the steps performed prior to initialising USDPAA threads, i.e. prior to binding pthreads to QMan and BMan portals.

DPAA Application - Global Initialization for Device Tree Parsing

The USDPAA “of” driver needs to parse the device-tree before any dependent driver or application layers are activated. This is performed via of_init(), as seen early in apps/ppac/main.c:main(). This does not do any configuration of devices, it simply parses the information in the device-tree in order to make it available for the various device drivers that will subsequently initialise. (Note that “of” refers to the “Open Firmware” standard upon which u-boot and linux device-tree handling is based, and the APIs for manipulating such device-trees are typically prefixed with “of_” in those environments too.)

DPAA Application - Global Initialization for Network Device Configuration Parsing

The application-level “usdpaa_netcfg” interface parses the device-tree and the two XML files used with FMC to program the FMan devices. This interface is in turn built on top of the lower-level interface associated with the USDPAA “FMan” driver (and initialisation of “usdpaa_netcfg” will implicitly initialise the “FMan” driver layer). See include/usdpaa/{usdpaa_netcfg.h,fman.h}, which is used by PPAC in apps/ppac/main.c:main() where it calls usdpaa_netcfg_acquire().

USDPAA apps should choose to bypass the “usdpaa_netcfg” layer and optionally the “FMan” driver layer too if it wishes to implement its own parsing of the configuration, or even obtain the configuration information from another source.

DPAA Application - Global Initialization for QMan and BMan

The USDPAA QMan and BMan drivers currently initialise some global resources using compiled-in constants. This setup is driven by the qman_global_init() and bman_global_init() APIs. This is done in apps/ppac/main.c:main().

DPAA Application - DMA Memory Initialization

The USDPAA “dma_mem” driver initialises and manages access to a memory-mapped region of physically-contiguous memory with trivial virtual/physical address conversion and no page-faulting. This is via the dma_mem_setup() API, as called in apps/ppac/main.c:main().

DPAA Application - Thread Initialization

This section covers the setup of USDPAA threads.

DPAA Application - Portal Interrupts

A running thread can request the allocation and configuration of a thread-affine QMan or BMan portal using the qman_thread_init()/bman_thread_init() APIs. This is done in apps/ppac/main.c:worker_fn(), after the thread has been spawned from apps/ppac/main.c:worker_new(). Note that this specifies to the QMan/BMan drivers the CPU that the portals need to be affine to, but it is up the application when (or even if) to bind the thread to the nominated CPU. In the case of PPAC, this is performed immediately prior to initialising the portals, via the pthread_setaffinity_np() API.

DPAA Applicaiton - Thread-local Enqueue Objects

Rather than using distinct QMan FQ objects for each FQID that it wishes to enqueue to, PPAC instead declares a thread-local, global FQ object for enqueuing to any FQID. This is done in apps/ppac/main.c:worker_fn(), search for “local_fq”. This means that PPAC and PPAM logic only needs to know and specify the FQIDs of the various Tx FQs for each interface rather than tracking objects for each such FQ. Moreover, it also avoids the sharing of Tx FQ objects across multiple threads (and thus across CPUs) unnecessarily, which is to avoid the potential for cache-coherency overheads. This has its disadvantages too – when PPAC performs an enqueue, it must directly substitute the FQID within the thread-local FQ object. Also, in the event of any ERNs (enqueue rejection notifications), the PPAC-implemented callback cannot rely on the FQ object to identify the FQID (or interface) which relates to the failed enqueue, so would have to interpret that information from the FQID contained in the notification. The PPAC currently does no such demuxing and simply drops any frames that are rejected.

DPAA Applicaiton - Portal dequeue mapping of pool-channels

The network interface configuration mentioned earlier, obtained via usdpaa_netcfg_acquire(), can be used to identify the QMan pool-channels that the network interfaces' Rx FQs are scheduled to. Using this information, the application can determine the pool-channels it wishes the initialised portal to dequeue from. The PPAC code computes this mask in apps/ppac/main.c:main() (search for “sdqcr”), and then each thread applies it to its QMan portal in apps/ppac/main.c:worker_fn() via the qman_static_dequeue_add() API.

DPAA Application - Thread run-to-completion loop

This section covers the run-to-completion loop of a USDPAA application thread once it has been initialised. For USDPAA, the QMan and BMan portals are usually set in run-to-completion mode, meaning that none of the portal interrupt sources are enabled and all portal duties are configured to be processed by polling instead of interrupt-handling. The expectation is therefore that the application thread will repeatedly “poll” the portals within the core-loop of its execution.

DPAA Application - Fast Path Processing

For QMan, the run-to-completion processing is split into two halves. The “fast” part deals with processing of the portal's dequeue ring (DQRR), because this is the most speed-critical mechanism and also the most hardware-optimised. In particular, with portal ring-stashing enabled (as it is by default in USDPAA), processing of this ring rarely involves any cache-stalls, whether there is dequeue work available to be processed or not, so it can be executed quickly and in a tight-loop without any large fixed overheads. (Processing that involves reading a cache-inhibited register, on the other hand, will always have a minimum overhead due to the latency of issuing a read to hardware and waiting for the response.)

So the core loop of PPAC-based application threads call qman_poll_dqrr() frequently, such that other kinds of processing are far less frequent. Or put another way, most iterations of the thread's run-to-completion loop do nothing except call qman_poll_dqrr(). See apps/ppac/main.c:main(), the calls to qman_poll_dqrr().

DPAA Application - Slow Path Processing

For QMan, the other side of run-to-completion processing encompasses “everything else”, i.e. all portal processing except DQRR. This is because polling for other processing duties, even if there is no work to be done, will require a minimal overhead in order to interrogate hardware. In the case of BMan, there is no “fast” path as such, as there is no analogy for DQRR. The speed-critical processing of releasing buffers to (and acquiring buffers from) buffer pools does not require maintenance from the run-to-completion loop (they are command based and self-maintaining).

So PPAC-based application threads calls qman_poll_slow() and bman_poll_slow() within the run-to-completion loop, but use a throttling scheme to ensure that their overheads are not incurred on every iteration. See apps/ppac/main.c:main().

DPAA Application - Packet Processing

The task of examining a packet (or frame) and determining the appropriate action is triggered by the “Fast path processing” mentioned above. When qman_poll_dqrr() is called and dequeued frames are discovered and handled, callbacks will be invoked for the FQ objects to which the dequeued frames belonged. So packet-processing occurs within callbacks registered with FQ objects. In the case of PPAC, the main example for the performance-critical case is in apps/include/ppac.c:cb_dqrr_rx_hash(). However, this handler defers all the significant work to the PPAM hook, as indeed that is what PPAMs are - they implement the packet-handling specifics. As can be seen from cb_dqrr_rx_hash(), it calls ppam_rx_hash_cb() to process the packet. See apps/reflector/reflector.c:ppam_rx_hash_cb(). This in turn defers all work to apps/reflector/reflector.c:reflect_cb() (because in some configurations, the same processing is performed on the “rx_default” FQ too, so the implementation is factored out).

This packet processing uses dma_mem_ptov(qm_fd_addr()) to extract the frame address from the frame-descriptor (whether the frame descriptor also specifies an offset or not) and converts it from a device-physical to a usable pointer/virtual address. The packet is then examined, and the outcome is either that the packet is dropped via ppac_drop_frame() or it is forwarded via ppac_send_frame().

Note also that in the case of the frame being forwarded, the FQID to which it is transmitted is derived from the PPAM's own FQ state that was initialised during the network interface initialisation phase. In particular, the Tx FQIDs for each interface are captured as the interface initialises, so that when the Rx FQs for the same interface are initialised, the PPAM-specific state immediately makes a fixed assignment for each Rx FQ to one of the Tx FQs. (As a “reflector” this is possible because all packets need to go back out the interface they come in on. This is quite different to “IPFwd” and other forwarding applications of course.)

9.9.4.13 A non-PPAC comparison, “hello_reflector”

A stand-alone version of reflector was created in order to illustrate how to move from a PPAM-based prototype to a stand-alone code-base for development of stand-alone user applications. To emphasise its “hello world” nature, this application is called “hello_reflector”.

"hello reflector" is located in apps/hello_reflector/. Many of the features of the PPAC/PPAM version are missing from the stand-alone version in order to keep it simple (and besides which, those features are provided from the PPAC infrastructure rather than from the reflector PPAM itself, so re-implementing those features in a new, stand-alone form would not illustrate anything useful). The “hello_reflector” code is also intended to be self-documenting, so there is no user guide for it nor will it be discussed in great deal here. However, comparing the reflector PPAM code with hello_reflector should make it clear that the packet-processing is fundamentally the same. Furthermore, the single hello_reflector C file implements all the initialisation described in section 10 outside of PPAC, so should be instructive for those looking to write stand-alone (non-PPAC) USDPAA applications.

9.9.5 USDPAA Reflector and PPAC User Guide SDK

9.9.5.1 Introduction

The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document describes the "reflector" application and the PPAC abstraction on which it is built, both contained in the USDPAA package. This application serves as a reference example for working with the USDPAA interface, as well as providing

a benchmark for USDPAA system performance. More elaborate PPAC-based applications (such as "ipfwd") have their own user guide documents, but do not repeat the PPAC discussion found here.

Furthermore, a stripped-down "hello_reflector" application exists within USDPAA that is implemented without using the PPAC abstraction. This application is missing many of the features of PPAC, but is intended to provide a usable comparison between PPAC-based and stand-alone applications, eg. for customers prototyping with PPAC and then looking to develop stand-alone production code. As "hello_reflector" is very simplistic, the source code is self-documenting and it will not be discussed in any detail here. A short guide to executing the "hello_reflector" application will be provided at the end of this document.

A variant of hello_reflector called hello_reflector "short circuit" is also provided which basically tests the sanity of the hardware path for the packet flow by this way without any processing by the core on the received packets.

9.9.5.1.1 Intended audience

This document is intended for software developers and system architects who work with the USDPAA framework.

The material is technical in nature. The reader is assumed to be familiar with:

- General Linux software development, operation, and configuration for Power architecture devices in particular.
- Familiarity with the concept of device drivers and their role in operating systems.
- Linux network administration and use.
- The NXP Linux SDK for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.
- The USDPAA User Guide document (this document covers only the reflector application).
- Linux UIO (User space I/O) driver infrastructure (USDPAA drivers for QMan/BMan portals use the standard UIO mechanisms for memory-mapping and interrupt-handling).

The P4080 was the first NXP SoC to incorporate the DPAA. As such, it is used in many examples. However, USDPAA is intended to apply in the same manner to all DPAA-based SoCs.

9.9.5.1.2 Change history

Table 270. Change History

Version	Updates
1.0	Creation of reflector UG for phase1 release, based on the general UG prepared for the phase0 release. - tidy up of text - describe phase1-specific additions; PPAC/PPAM, IRQ mode, etc.
1.1	Add section on testing reflector using a Linux PC.
1.2	Fix minor typographical errors.
1.3	Updates for v1.0 release. - split PPAC-common material from reflector specifics - corrected and updated buffer pool descriptions

Table continues on the next page...

Table 270. Change History (continued)

Version	Updates
1.4	Updates for v1.1 release - document "hello_reflector" - document cmd-line args and environment-variables - update sample console output - update buffer definitions

9.9.5.2 Overview of reflector

The reflector application is a simple packet-processing USDPAA application. The main purpose of USDPAA's user space drivers (as opposed to more conventional kernel-mediated device access) is raw I/O performance, and the reflector application focuses on this. Reflector is basically all I/O - it receives and sends frames, but does very little processing on them. It demonstrates the I/O capabilities of the DPAA hardware and corresponding user-space drivers.

9.9.5.3 Overview of PPAC

The source code to reflector has been reorganized into two parts; the "PPAC" (Packet-Processing Application Core) and a "PPAM" (Packet-Processing Application Module). The idea behind this is that many packet-processing applications (particularly any variations on the theme of network forwarding) would likely only differ from reflector in the way they make their "forwarding decision". The essential difference is the logic that looks at the packet and determines what to do with it. The PPAM portion implements this application specific logic. On the other hand, the PPAC component represents the common infrastructure to support such PPAMs; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers, [etc]. For USDPAA demonstration applications, the use of a common PPAC component allows for easier code development and maintenance and a uniform application look and feel.

Users are not obliged to use PPAC for their USDPAA application development. Moreover users could choose to implement PPAC-based applications (eg. for quick prototyping) and then later port their work to a stand-alone product. A stand-alone version of reflector, called "hello_reflector", has been provided to illustrate this principle. Please note that the emphasis in "hello_reflector" is on simplicity, so it does not (re)implement stand-alone equivalents of all the features and flexibility of the PPAC environment.

Additionally, if the user only wants to first check the sanity of the hardware path for the packet flow and later wants to send the packets to the core for the processing, he can first run hello_reflector in "short circuit" mode and if the traffic is received back on the hardware path, he can now run hello_reflector and so involve the core in the packet processing.

In "short circuit" hello reflector, packets are dequeued from the interface and received on Rx FQs. These packets are then sent out from the QMAN channel on which Tx FQs are scheduled.

9.9.5.4 PPAC details

However at an implementation level, there is a limit to the degree of abstraction one can obtain. E.g. for 64 byte packets, using between 1 and 4 CPUs on a p4080 DS, reflector processing averages out to approximately 170 CPU cycles per-packet. Adding just one extra level of indirection to that processing path can add enough overhead to make a visible performance difference to the application (one such ad-hoc experiment showed an additional overhead of ~20 cycles, causing a 12% performance degradation). For this reason, the PPAC/PPAM interface is implemented by a strategic use of inlining. The resulting PPAC implementation and interface is organized in such a way that the compiler is able to inline the fast-path code of PPAC and PPAM together, as though they were written as a single (or "flat") application. Indeed, the PPAM version of reflector, despite the PPAC/PPAM split, has no performance degradation relative to earlier non-PPAM versions.

9.9.5.4.1 IRQ mode for sleeping when idle

A new feature of PPAC (and thus all PPAM applications like reflector and "ipfwd") is support for interrupts and sleeping. As USDPAA's QMan and BMan drivers use the standard Linux UIO subsystem, the behavior of interrupt-handling is in keeping with UIO semantics. Note that the qman_irqsource_*() and bman_irqsource_*() APIs must be used prior to entering a blocking read(), select(), poll(), [etc] to configure the relevant QMan/BMan portals to report activity via interrupt. Failure to do so may cause the application to block indefinitely waiting for interrupts that won't occur because the portals are configured for polling. Similarly, when leaving IRQ/blocking mode to run in polling mode (and in particular to post-process the events that caused interrupts), one must again use the "irqsource" APIs, this time to configure the portals to be processed by polling APIs rather than interrupts.

For an illustration of PPAC's use of IRQ mode, see apps/ppac/main.c, specifically the core loop of the worker_fn() function. This loop migrates from polling mode to a blocking "IRQ mode" built around select() whenever reflector has looped a certain number of times without any forward progress. If input traffic is significantly below processing capacity, then it should be possible to observe reflector going in to (and out of) blocking select()s, even if the traffic is never completely stopped. In such cases, any latency associated with sleeping and scheduling is absorbed by system buffering long enough for reflector to wake up and resume polling mode (and catch up on the buffered backlog). This mechanism allows reflector to share CPUs with other tasks more effectively (and consume less power) provided the throughputs are low or busty enough to make this acceptable.

9.9.5.4.2 Buffers

When starting up, PPAC applications seed the 3 buffer pools that the Fman is configured to use for Rx processing. To improvement restartability, the pools are first drained of any stale contents, after which they are seeded using allocations from the "/dev/fsl_usdpaa_shmem" DMA device, which is described in the USDPAA User Guide.

The three buffer pools initialized and used by PPAC (and the default USDPAA configuration of Fman) have the following attributes.

Table 271. Buffer Pool attributes

Pool ID	Buffer Size (used by Fman)	Number of Buffers
7	320	0
8	704	0
9	1728	0x2000

When processing IPv4 frames, buffers are allocated by FMan on Rx and then released by FMan after Tx.

The source code specifies the buffer pool requirements in include/internal/conf.h:

```
#define DMA_MEM_BP1_BPID        7
#define DMA_MEM_BP1_SIZE        320
#define DMA_MEM_BP1_NUM         0 /* 0*320==0 (0MB) */
#define DMA_MEM_BP2_BPID        8
#define DMA_MEM_BP2_SIZE        704
#define DMA_MEM_BP2_NUM         0 /* 0*704==0 (0MB) */
#define DMA_MEM_BP3_BPID        9
#define DMA_MEM_BP3_SIZE        1728
#define DMA_MEM_BP3_NUM         0x2000 /* 0x2000*1728==13.5MB) */
#define DMA_MEM_BPOOL \
          (DMA_MEM_BP1_SIZE * DMA_MEM_BP1_NUM + \
          DMA_MEM_BP2_SIZE * DMA_MEM_BP2_NUM + \
          DMA_MEM_BP3_SIZE * DMA_MEM_BP3_NUM) /* (13.5MB) */
```

And the application implements the seeding of these buffers according to the following structure in apps/ppac/main.c:

```
/* Seed buffer pools according to the configuration symbols */
const struct ppac_bpool_static {
          int bpid;
          unsigned int num;
          unsigned int sz;
} ppac_bpool_static[] = {
          { DMA_MEM_BP1_BPID, DMA_MEM_BP1_NUM, DMA_MEM_BP1_SIZE},
          { DMA_MEM_BP2_BPID, DMA_MEM_BP2_NUM, DMA_MEM_BP2_SIZE},
          { DMA_MEM_BP3_BPID, DMA_MEM_BP3_NUM, DMA_MEM_BP3_SIZE},
          { -1, 0, 0 }
};
```

9.9.5.4.3 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired;

9.9.5.4.3.1 Order preservation

Order preservation is a functionality of the QMan software portal interface that allows processing of Rx FQs across multiple portals to retain order when transmitted corresponding Tx FQs. The technique only applies to frames dequeued from a given Rx FQ that are all transmitted out the same Tx FQ (which is the case for "reflector", and also the case for "ipfwd" when frames are from the same flow). The mechanism requires two QMan features, "HOLDACTIVE" and "enqueue DCA". The former ensures that a FQ that has been dequeued to a software portal from should remain bound to that portal until all the corresponding DQRR entries have been consumed. The latter ensures that a DQRR entry is consumed by QMan itself once it has dispatched the corresponding enqueue (Tx) command. Together, for any given Rx/Tx FQ pair, the processing via pool-channels and multiple CPUs does not allow frame processing to get out of order. (For more information on this feature, consult the QMan/BMan API Guide.)

Use of "HOLDACTIVE" is mutually exclusive with another QMan option "AVOIDBLOCK", which is selected by default in PPAC. So to enable order-preservation, one must change the settings from;

```
#undef PPAC_2FWD_HOLDACTIVE
#undef PPAC_2FWD_ORDER_PRESERVATION
#define PPAC_2FWD_AVOIDBLOCK
```

to;

```
#define PPAC_2FWD_HOLDACTIVE
#define PPAC_2FWD_ORDER_PRESERVATION
#undef PPAC_2FWD_AVOIDBLOCK
```

9.9.5.4.3.2 Monitoring Rx/Tx fill-levels via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. The CGRs are not configured to perform any flow-control (ie. no tail-drop nor WRED options are enabled), so this simply allows the user to monitor the overall fill-level of frame queues in the system, in particularly to determine whether build-up is occuring before or after the software-processing phase. The thresholds used for the CGRs are determined from the two constants also defined in ppac.h, PPAC_CGR_RX_PERFQ_THRESH and PPAC_CGR_TX_PERFQ_THRESH. These constants, combined with the number of FQs, define the thresholds for CGR congestion-entry. Qman in turn defines the CGR exit-threshold to be 7/8 the entry-threshold (to provide hysteresis), and so these combined entry/exit events will be logged to stdout independently for the Rx and Tx CGRs.

An extra command, "cli", becomes available in the CLI when this feature is compiled in, which will query and display all the fields of both CGRs. Note however that this option introduces extra latency, and more critically, extra contention within the system. This is because Qman must lock the CGR for each enqueue and dequeue event that relates to it, meaning that the forwarding of a single packet through the system requires 2 lock/unlock pairs for each CGR (thus 4 lock/unlock pairs). A small but noticable performance degradation should be expected when running in this mode. (Real-world use of CGRs would not subscribe all Rx/Tx FQs from all interfaces to a single CGR, so this scalability issue should not be a concern for production software usage of CGR-based congestion management.)

To enable this feature, change;

```
#undef PPAC_CGR
```

to;

```
#define PPAC_CGR
```

9.9.5.4.3.3 Other settings

Many other settings used by PPAC (and thus PPAC-based apps) are defined in ppac.h but they are less intended for ad-hoc manipulation than those mentioned above. However a curious user may wish to examine some of these, and perhaps search out their usage within the source-code, in order to see how they are used and explore some of the driver interfaces and application design in this way.

9.9.5.5 Running reflector

The following comments about executing reflector apply to all PPAM applications - PPAMs may of course define new command-line (and environment-variable) behaviour, but the following behaviour is all generic to PPAC (reflector does not implement any extensions of its own).

After logging in to the p4080 DS environment as "root", the SOURCE_THIS file in the "/root" directory can be used to simplify running "fmc" (to configure FMan for the required network device configuration) and starting up reflector, as can be seen from the following:

```
login: root
Password:
[root@p4080 root]# cat /root/SOURCE_THIS
cd /usr/etc
fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
reflector
[root@p4080 root]# source /root/SOURCE_THIS
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@4, Tx Channel = 40, FMAN = 0, Port ID = 0
Found /fsl,dpaa/ethernet@7, Tx Channel = 63, FMAN = 1, Port ID = 2
Found /fsl,dpaa/ethernet@8, Tx Channel = 64, FMAN = 1, Port ID = 3
Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0
Qman: FQID allocator includes range 512:128
Bman: BPID allocator includes range 56:8
Configuring for 4 network interfaces and 4 pool channels
FSL dma_mem device mapped (phys=0xe0000000,virt=0x70000000,sz=0x1000000)
Thread uid:0 alive (on cpu 1)
Release 0 bufs to BPID 7
Release 0 bufs to BPID 8
Release 8192 bufs to BPID 9
reflector starting
Thread uid:0 alive (on cpu 1)
reflector>
```

Reflector starts up with a single thread running on CPU 1 by default, with all the network interfaces enabled. The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 can not be removed (except by shutting down the application).

Reflector can alternatively be started as a single thread on a different CPU by executing it with the desired CPU as its sole argument:

```
[root@p4080 root]# reflector 5
```

Alternatively, multiple threads can be started by specifying a range of CPUs:

```
[root@p4080 root]# reflector 3..7
```

By default, reflector is compiled to load the XML configuration and PCD files passed to the "fmc" tool in the afore-mentioned SOURCE_THIS script. Indeed, it is important that reflector always load the same configuration as is passed to "fmc". If "fmc" is run with different XML inputs, eg. when running on another board than p4080ds and/or when using a different SERDES configuration, then reflector must be instructed to load non-default XML files to match. This can be achieved either by setting the DEF_PCD_PATH and DEF_CFG_PATH environment variables, or by using short or long command-line arguments: Eg. the following 3 examples achieve the same thing:

```
[root@p4080 root]# reflector -c my_cfg.xml -p my_pcd.xml

[root@p4080 root]# reflector --fm-config my_cfg.xml --fm-pcd my_pcd.xml

[root@p4080 root]# export DEF_CFG_PATH=my_cfg.xml
[root@p4080 root]# export DEF_PCD_PATH=my_pcd.xml
[root@p4080 root]# reflector
```

Note also that PPAC implements a CLI for all PPAM applications, which means that it expects a console to be present on 'stdin'. If the reflector process is backgrounded, or run as a daemon, then the process will pause waiting for it to receive control of console input. If the user wishes to launch reflector or any other PPAM without the use of the console, they should instruct it ignore input and not run the CLI by passing the "-n" or "--non-interactive" command-line arguments:

```
[root@p4080 root]# reflector --non-interactive &
```

9.9.5.6 PPAC (and reflector) CLI commands

The following commands are illustrated in the context of reflector (carrying on from the session started in the previous section), but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

```
reflector> add 2
```

To add threads on a range of CPUs:

```
reflector> add 3..6
```

To list the threads (this also queries each thread, verifying that they aren't blocked):

```
reflector> list
Thread uid:0 alive (on cpu 1)
Thread uid:1 alive (on cpu 2)
Thread uid:2 alive (on cpu 3)
Thread uid:3 alive (on cpu 4)
Thread uid:4 alive (on cpu 5)
Thread uid:5 alive (on cpu 6)
```

To remove a thread by its UID:

```
reflector> rm uid:2
Thread uid:2 killed (cpu 3)
```

To remove a thread running on a given CPU:

```
reflector> rm 4
Thread uid:3 killed (cpu 4)
```

If more than one thread is running on a given CPU (only possible if the device-tree has been updated to reserve multiple portals for USDPAA on the same CPU), then this removes the first thread found running on the named CPU. A range of CPUs can likewise be specified:

```
reflector> rm 5..6
Thread uid:4 killed (cpu 5)
Thread uid:5 killed (cpu 6)
```

To perform a controlled shutdown of reflector (this includes disabling the network ports):

```
reflector> quit
```

9.9.5.7 Running hello_reflector

The stand-alone "hello_reflector" application, which is not PPAC-based, can be executed in exactly the same environment as the PPAC-based "reflector". Instead of executing "reflector", one executes "hello_reflector" instead:

```
[root@p4080 root]# hello_reflector
```

Hello_reflector always starts up on CPU 0, and by default it only starts a single thread. Multiple threads can be started by providing the "-n" argument, in which case that many CPUs will be used starting with CPU 0 and counting upwards.

```
[root@p4080 root]# hello_reflector -n 5
```

Hello_reflector does not implement a CLI, so the only way to gracefully shut it down is to enter a Ctrl-C on the controlling console, or equivalently issue a SIGINT signal to the hello_reflector process.

Alternative configuration and PCD files can be provided to hello_reflector by using the "-c" and "-p" arguments respectively. These should always match the XML files that are passed to "fmc":

```
[root@p4080 root]# hello_reflector -p my_pcd.xml -c my_cfg.xml
```

9.9.5.8 Running hello_reflector (short circuit)

Issue following command to run hello_reflector in "short circuit" mode:

```
[root@p4080 root]# hello_reflector -c /usr/etc/usdpaa_config_p4_serdes_0xe.xml -
p /usr/etc/usdpaa_policy_hash_ipv4.xml –n 1 –sc
```

To test sc mode, stop the process (ctrl + Z) and feed the traffic to FMAN port (using spirent), you will still see the rx traffic due to the fact that the destination channel for RX queues is tx_channel.

Follow these steps precisely :

1. Hang up process by ctrl+z, this will avoid using cpu and will not impact traffic flow.

2. To quit process, you have to recover process by 'fg %1' then press 'ctrl+c' to quit.

9.9.5.9 Testing reflector

Functional testing of the "reflector" application is possible by connecting any subset of the P4080 USDPAA network interfaces to a conventional computer, assumed to be a Linux PC using an ethernet switch as shown in the figure below.



Figure 260. Testing Reflector

The figure assumes that the default USDPAA SerDes 0xe reset configuration word (RCW) is used. This configuration provides 2 x 10 Gbps and 2 x 1 Gbps ethernet interfaces to the USDPAA application. The test case in the figure assumes that only the 2 x 1 Gbps interfaces will be used. These are the two interfaces on the SGMII riser card that are closest to the P4080DS motherboard. This test will work even if 10 Gbps XAUI riser cards are not fitted in the P4080DS.

See the main USDPAA User Guide for more information on network interfaces.

To perform the test, boot the P4080 and run the example application as described in section Running reflector on page 1695.

The IP and ethernet MAC addresses used below are examples. You can change them as long as you are consistent. The most important thing is to be sure of the MAC addresses on the P4080DS board. Again, see the main USDPAA User Guide.

On the Linux PC, create eth0:1 through eth0:3 via, for example:

```
sudo ifconfig eth0:1 192.168.10.1 netmask 255.255.255.0
sudo ifconfig eth0:2 192.168.20.1 netmask 255.255.255.0
```

The reflector application does not respond to ARP requests, so static ARP entries for all of the P4080 USDPAA interfaces must be created on the Linux PC:

```
sudo arp -s 192.168.10.2 00:04:9F:77:4E:07
sudo arp -s 192.168.20.2 00:04:9F:77:4E:08
```

Then, from the Linux PC ping one of the P4080 interfaces, e.g. "ping 192.168.10.2". This causes the following sequence:

1. Linux PC sends ICMP echo request to the switch.
2. At least for the first ping, the switch will flood the request to all P4080 ports. It will be dropped by all but the correct one.
3. The P4080 receives the frame, swaps the IP and MAC addresses, and sends the frame back out the MAC it came in on.
4. The Linux PC receives the frame, an ICMP echo request.
5. The Linux PC responds to the request.
6. The response is received by the P4080 which swaps the IP and MAC addresses and sends the frame back out on the MAC it came in on.
7. The Linux receives the response.
8. The original ping is now complete. The Linux PC just "pinged itself via the P4080".

It is possible to also test with different packet types using this same technique. For example, have a telnet server running on the Linux PC and then do "telnet 192.168.10.2" from the Linux PC. The Linux PC will telnet to itself via the P4080. One can also use ssh in the same way.

To test performance, it is best to used dedicated ethernet traffic generation equipment such as a Spirent Test Center. Such equipment can inject packets into the P4080 at known and very high rates and measure the rate of the packets that egress from the P4080.

9.9.6 NXP USDPAA IPFWD User Manual Rev. 1.2

9.9.6.1 About this Book

The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA IPFwd application
- Execution steps for USDPAA IPFwd application from the NXP yocto package on the P4080DS/P3041DS/P5020DS/ T4240QDS/B4860QDS/B4420QDS.

Conventions

This document uses the following conventions:

Courier is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

9.9.6.2 Introduction

9.9.6.2.1 Purpose

This document describes the USDPAA IPv4 forwarding application. This application documents the USDPAA IPv4 demonstration applications forwarding flow and its performance measured on a P4080DS.

9.9.6.3 Overview

The USDPAA IPv4 forwarding (IPFwd) application is a multi-threaded application that routes IPv4 packets from one ethernet interface to another on all QorIQ platforms. The routing is done based on the source IP address and destination IP address in the frame. Any combination of the cores can run a USDPAA IPv4 Forwarding application thread in USDPAA SDK v1.1.

IPFwd packet-processing:

· Receives ethernet frames on all USDPAA ethernet interfaces.

· Discards (and deallocates buffers for) all non-IPv4 frames.

· For IPv4 frames processing takes place as defined in section Overview of packet flow: on page 1700

9.9.6.3.1 USDPAA IPv4 forwarding application flow

The IPFwd application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAA PPAC User Guide for more details). The configuration phase also includes PPAM (i.e. IPFwd) related initialization. Once the configuration phase is completed the IPFwd application moves to the packet processing phase. This application provides a command-line interface to enable users to add and remove routing table and ARP cache entries at any given time. For each user input, the appropriate information is communicated to the IPFwd application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPfwd application. Note that the IPFwd application does not dynamically resolve ARP – missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking “IRQ mode” whenever IPFwd has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to “USDPAA PPAC User Guide”. In polling mode – the application constantly looks for data to process on its dedicated QMan portal . Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

9.9.6.3.1.1 Overview of packet flow:

1. Route table entries are populated using the IPFwd configuration commands at any given time.
2. Packet is received by the FMan, which uses 2-tuple (Src IP & Dest IP) to hash the packet to a Rx frame queue.
3. The classified packet is presented to the USDPAA IPFwd application thread running on one of the cores - the distribution is based on the portal and FQ setup done by the application during its portal and frame queue initialization. The packet is subjected to IPv4 Forwarding route lookup.
4. Based on the packet destination, the application transmits the packet by enqueueing it on a frame queue destined for the appropriate Tx interface.



Figure 261. Packet flow for USDPAA IPv4 Forwarding

9.9.6.4 Overview of PPAC

The source code to IPFwd has been reorganized into two parts; the “PPAC” (Packet-Processing Application Core) and a “PPAM” (Packet-Processing Application Module). The PPAM portion implements the IPFWD application specific logic of processing the packet to forward it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document “USDPAA PPAC User Guide”.

9.9.6.5 IPFwd related PPAC Details

9.9.6.5.1 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired.

9.9.6.5.1.1 Order Preservation in IPFWD

This section describes how user can enable Order Preservation in IPFWD application. By default Order Preservation is disabled in IPFWD application and in order to enable it the user will have to re-compile the binary by making following changes to the source code.

In file, usdpaa/apps/include/ppac.h you can find these two lines.

/* Application options */

#undef PPAC_2FWD_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_2FWD_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

Change the above to

#define PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */

#define PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

And then compile usdpaa once again. Now run the IPFWD application with Order Preservation.

9.9.6.5.1.2 Order Restoration in IPFWD

Oder restoration is the functionality of QMan software portal interface which restores the relative temporal order of a flow of frames (sequence of frames) to that observed before transmitting to the destination Frame Queue and Order Definition Point (ODP) takes note of the correct order of packets before start processing by using the sequence number. Use of "HOLDACTIVE" is mutually exclusive with another QMan option "AVOIDBLOCK", which is selected by default in PPAC. To enable order-restoration, the user will have to re-compile the binary by making following changes to the source code.

/* Application options */

#undef PPAC_2FWD_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_2FWD_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

#undef PPAC_2FWD_ORDER_RESTORATION /* Use ORP */

#define PPAC_2FWD_AVOIDBLOCK /* No full-DQRR blocking of FQs */

Change the above to

#undef PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

#define PPAC_ORDER_RESTORATION /* Use ORP */

#define PPAC_AVOIDBLOCK /* No full-DQRR blocking of FQs */

And then compile usdpaa once again. Now run the IPFWD application with Order Restoration. Implementation note: Order restoration has been implemented such that each PCD Frame queue has a corresponding ORP (order restoration point) frame queue associated with it. Each ORP is configured with default window settings as seen below

#define PPAC_ORP_WINDOW_SIZE 7 /* 0->32, 1->64, 2->128, ... 7->4096 */

#define PPAC_ORP_AUTO_ADVANCE 1 /* boolean */

#define PPAC_ORP_ACCEPT_LATE 3 /* 0->no, 3->yes (for 1 & 2->see RM) */

Here the ORP window size is set to be 4K, auto advance window size as 4K and accept late arrival window size as 8K. This ensures that no traffic is getting dropped but are always accepted below and at Zero loss throughput. Beyond zero loss throughput, as usual packets would be dropped and thus you can see mis-ordering.

ORP FQ descriptor attributes settings:

- Prefer in cache
- No " HOLDACTIVE "
- No " AVOIDBLOC K"
- ORP enabled

Assumption: To see the effect of Order Restoration in IPFwd application the user must use separate streamblocks as a source of traffic. If not done so, mis-ordering would be seen.

Key observation: It has been observed in IPFwd application that use of "HOLDACTIVE" with traffic generated using separate streamblocks, all the packets are IN sequence. Therefore, it is recommended that if user wants to see the real effect of Order

restoration in IPFwd application he should use “AVOIDBLOCK” with “RESTORATION” and not “HOLDACTIVE” with “RESTORATION”

9.9.6.5.1.3 Monitoring Rx/Tx fill-levels and flow-control via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. This simply allows the user to monitor the overall fill-level of frame queues in the system, in particular to determine whether build-up is occuring before or after the software-processing phase. Refer to section “ Monitoring Rx/Tx fill-levels via CGR ” of USDPAA PPAC User Guide for more details. To enable this feature, in ppac.h change;

#undef PPAC_CGR

to;

#define PPAC_CGR

The CGRs can also be configured to perform flow-control using Congestion state tail drop by setting CSTD_EN bits . Each congestion group record can be configured to track either byte counts or frame counts in all frame queues in the Congestion Group. When the threshold set for each CGR is exceeded, the CS bit is set in the CGR, and the congestion group is said to have entered congestion. At this point the incoming frames are marked for discard and QMAN will generate enqueue rejections to the producer. When the group’s I_BCNT returns below the threshold (minus approximately 1/8 of the threshold to provide hysteresis), the CS bit is cleared, and the congestion group’s state exits congestion. To enable tail drop, in ppac.h change;

#undef PPAC_CGR /* Track rx and tx fill-levels via CGR */

#undef PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

to

#define PPAC_CGR /* Track rx and tx fill-levels via CGR */

#define PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

And then compile usdpaa once again. Now run the IPFWD application with CGR tail drop enabled. To test this feature PPAC CLI provides a command “cgr” which will query and display all the fields of both CGRs. On pumping the traffic to IPFWD application at full line rate, t he instantaneous group byte count value I_BCNT(Instantaneous frame/byte count) must be maintained lesser than the CGR threshold set for each congestion group. Here is one such cgr command output:

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0e1e

a_bcnt: 0x00_0000_0e1e

Tx CGR ID: 11, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_0002

a_bcnt: 0x00_0000_0004

On the other hand if this feature of Congestion Group tail drop is disabled in IPFWD application I_BCNT is never maintained below CGR threshold value with traffic at full line-rate. This can be checked by compiling the IPFWD application with;

#define PPAC_CGR /* Track rx and tx fill-levels via CGR */

#undef PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

Now on pumping traffic at full line-rate, atleast one of the CGRs must go into congestion state and its I_BCNT should be above CGR threshold value. Here is the sample output:

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 1

cscn_targ: 0x00800000

cstd_en: 0

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0006

a_bcnt: 0x00_0000_0004

Tx CGR ID: 11, selected fields;

cscn_en: 1

cscn_targ: 0x00800000

cstd_en: 0

cs: 1

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_5fb7

a_bcnt: 0x00_0000_5fb8

Thus, the above test showcases the flow control achieved by enabling congestion group tail drop in IPFWD application.

9.9.6.6 PPAM related compile time configuration

9.9.6.6.1 One million route support

By default IPfwd application can work only with 1K routes. The user may want to run it for higher number of routes, so there is an option available in the header located at apps/ipfwd/include/app_common.h. To enable one million route support, the user will have to re-compile the binary by making following changes to the source code.

#undef ONE_MILLION_ROUTE_SUPPORT

Change the above line to

#define ONE_MILLION_ROUTE_SUPPORT

And then compile usdpaa once again. Now run the IPFWD application for one million routes. There is a sample shell script available at /usr/bin/ipfwd_20G_1Mroutes.sh in the rootfs. It creates one million routes using the 2 x 10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. The following table summarizes the setting done by this script.

Port ID	Source IP Address	Destination IP Address (for each src IP addr)	Default Gateway IP Address
FM1:TGEC1	192.168.24.2 to 192.168.24.255	192.168.29.2 .. 255	192.168.29.2
	192.168.25.2 to 192.168.25.255	192.168.29.2 .. 255	192.168.29.2
	192.168.26.2 to 192.168.26.255	192.168.29.2 .. 255	192.168.29.2
	192.168.27.2 to 192.168.27.255	192.168.29.2 .. 255	192.168.29.2
	192.168.28.2 to 192.168.28.255	192.168.29.2 .. 255	192.168.29.2
	192.168.1.2 to 192.168.1.255	192.168.29.2 .. 255	192.168.29.2
	192.168.18.2 to 192.168.18.255	192.168.29.2 .. 255	192.168.29.2
	192.168.30.2 to 192.168.30.255	192.168.29.2 .. 255	192.168.29.2
	192.168.31.2 to 192.168.31.91	192.168.29.2 .. 91	192.168.29.2
FM2:TGEC2	192.168.29.2 to 192.168.29.255	192.168.24.2 .. 255	192.168.24.2
	192.168.2.2 to 192.168.2.255	192.168.24.2 .. 255	192.168.24.2
	192.168.3.2 to 192.168.3.255	192.168.24.2 .. 255	192.168.24.2
	192.168.4.2 to 192.168.4.255	192.168.24.2 .. 255	192.168.24.2
	192.168.5.2 to 192.168.5.255	192.168.24.2 .. 255	192.168.24.2
	192.168.6.2 to 192.168.6.255	192.168.24.2 .. 255	192.168.24.2
	192.168.17.2 to 192.168.17.255	192.168.24.2 .. 255	192.168.24.2
	192.168.19.2 to 192.168.19.255	192.168.24.2 .. 255	192.168.24.2

Table continues on the next page...

Table continued from the previous page...

	192.168.20.2 to 192.168.20.91	192.168.24.2 .. 91	192.168.24.2

9.9.6.7 IPFWD Application Suite



The figure above shows the structure of the IPFWD USDPAA application suite. Its purpose is to forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using ipfwd_config command as mentioned in section Assign IP address to interfaces on page 1722 IP address can be assigned to all these interfaces. Each interface has a fixed netmask shown in the figure. The notation "/24" refers to a netmask of 255.255.255.0. The MAC addresses of these interfaces are determined by u-boot environment variables ethaddr, eth1addr, eth2addr, etc.

The ipfwd application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An ipfwd_config command should be used to set these MAC addresses.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "NXP DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAA applications. See the USDPAA User Guide for more information.
3. Finally, the fmc configuration file passed by command line argument to ipfwd_app determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by ipfwd_app are:

- FMan1-TGEC1
- FMan2-DTSEC3
- FMan2-DTSEC4
- FMan2-TGEC1

Running the ipfwd application suite involves four steps:

1. Run the fmc application to configure the FMan hardware instances.
2. Run ipfwd_app
3. Run ipfwd_config repeatedly to add routes to ipfwd's route cache.
4. Run "ipfwd_config -O" to start application processing.

Specific examples showing these steps are provided in other sections of this document.

9.9.6.8 Possible configuration scenario for IPFWD

IPfwd application can run in different configuration scenario. The table below shows the configuration files and corresponding sample shell script existing in the repository.

xml file	Sample shell script	Number of routes (as per sample shell script)
usdpaa_config_p4_serdes_0xe.xml (2x10G+2x1G)	ipfwd_20G.sh	1012 routes (2x10G)
	ipfwd_22G.sh	1022 routes (2x10G+2x1G)
usdpaa_config_p3_p5_serdes_0x36.xml (5x1G+10G)	ipfwd_15G.sh	1000 routes (5x1G+10G)

Figure 1: usdpaa_config_p4_serdes_0xe.xml

The figure above shows the ethernet interfaces available as per configuration file " usdpaa_config_p4_serdes_0xe.xml". It contains the 1 RGMII port (FMAN1, DTSEC 2), 2 SGMII ports (FMan2, DTSECs 3- 4) and 2 XAUI ports (FMAN1, TGEC1 and FMAN2-TGEC2). It is the user's wish to use any combination of ports available with the configuration file. The above table shows the sample shell scripts that user can use for this configuration. If user uses 2x10G, following is the flow configuration.

Test Center <-> P4080



Figure 2 : Flow Configuration for 2x10G on P4080DS



```
Port 1 to Port 2 (462 flows):
Source: 192.168.60.2 – 192.168.60.22
Destination: 192.168.160.2 – 192.168.160.23
Port 2 to Port 1 (462 flows) :
Source: 192.168.160.2 – 192.168.160.22
Destination: 192.168.60.2 – 192.168.60.23
Port 3 to Port 4 (46 flows):
Source: 192.168.130.2 – 192.168.130.22
Destination: 192.168.140.2 – 192.168.140.23
Port 2 to Port 1 (46 flows) :
Source: 192.168.140.2 – 192.168.140.22
Destination: 192.168.130.2 – 192.168.130.23
```

Figure 3 : configuration for 2x10G and 2x1G on P4080

For running IPFwd on P3041DS/P5020DS, the user can use " usdpaa_config_p3_p5_serdes_0x36.xml". Following is the flow configuration.



Left: 10G port (port6: fm1-10g)
Right: top-to-bottom: 1G port (port1-port6)

Figure 4 : Flow Configuration for 10G on P5020DS and P3041DS

```
Port 1 to Port 6 (100 flows):
                Src: 192.168.10.2 - 192.168.10.11
                Dst: 192.168.60.2 - 192.168.60.11

Port 2 to Port 6 (100 flows):
                Src: 192.168.20.2 - 192.168.20.11
                Dst: 192.168.60.2 - 192.168.60.11

Port 3 to Port 6 (100 flows):
                Src: 192.168.30.2 - 192.168.30.11
                Dst: 192.168.60.2 - 192.168.60.11

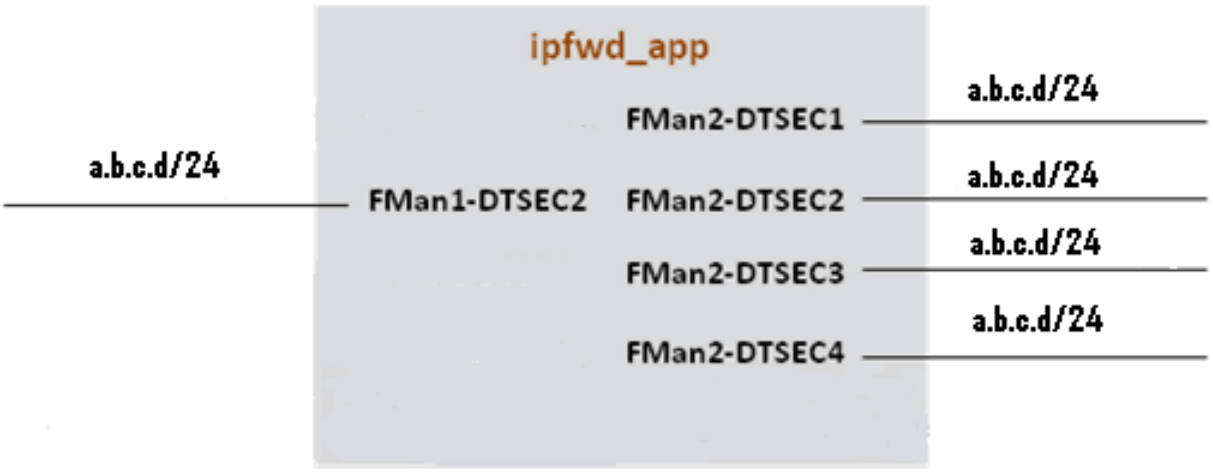
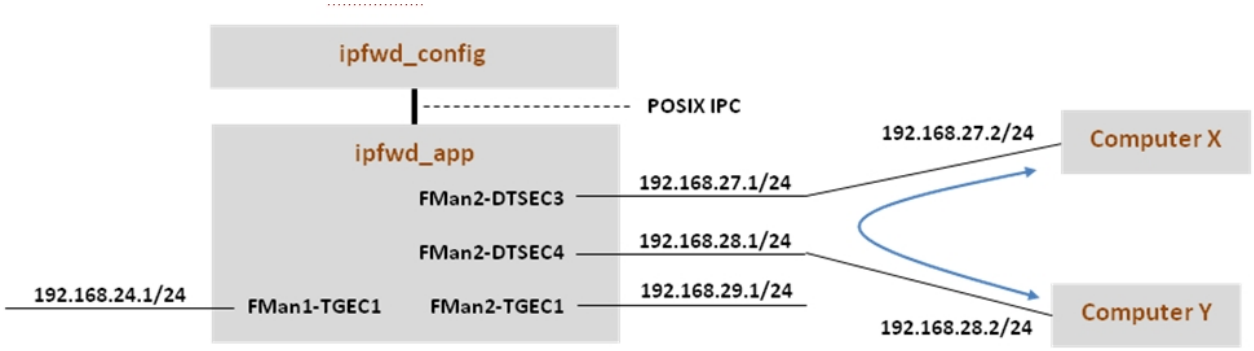
Port 4 to Port 6 (100 flows):
                Src: 192.168.40.2 - 192.168.40.11
                Dst: 192.168.60.2 - 192.168.60.11

Port 5 to Port 6 (100 flows):
                Src: 192.168.50.2 - 192.168.50.11
                Dst: 192.168.60.2 - 192.168.60.11
```

```
Port 6 to Port 1 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.10.2 - 192.168.10.11

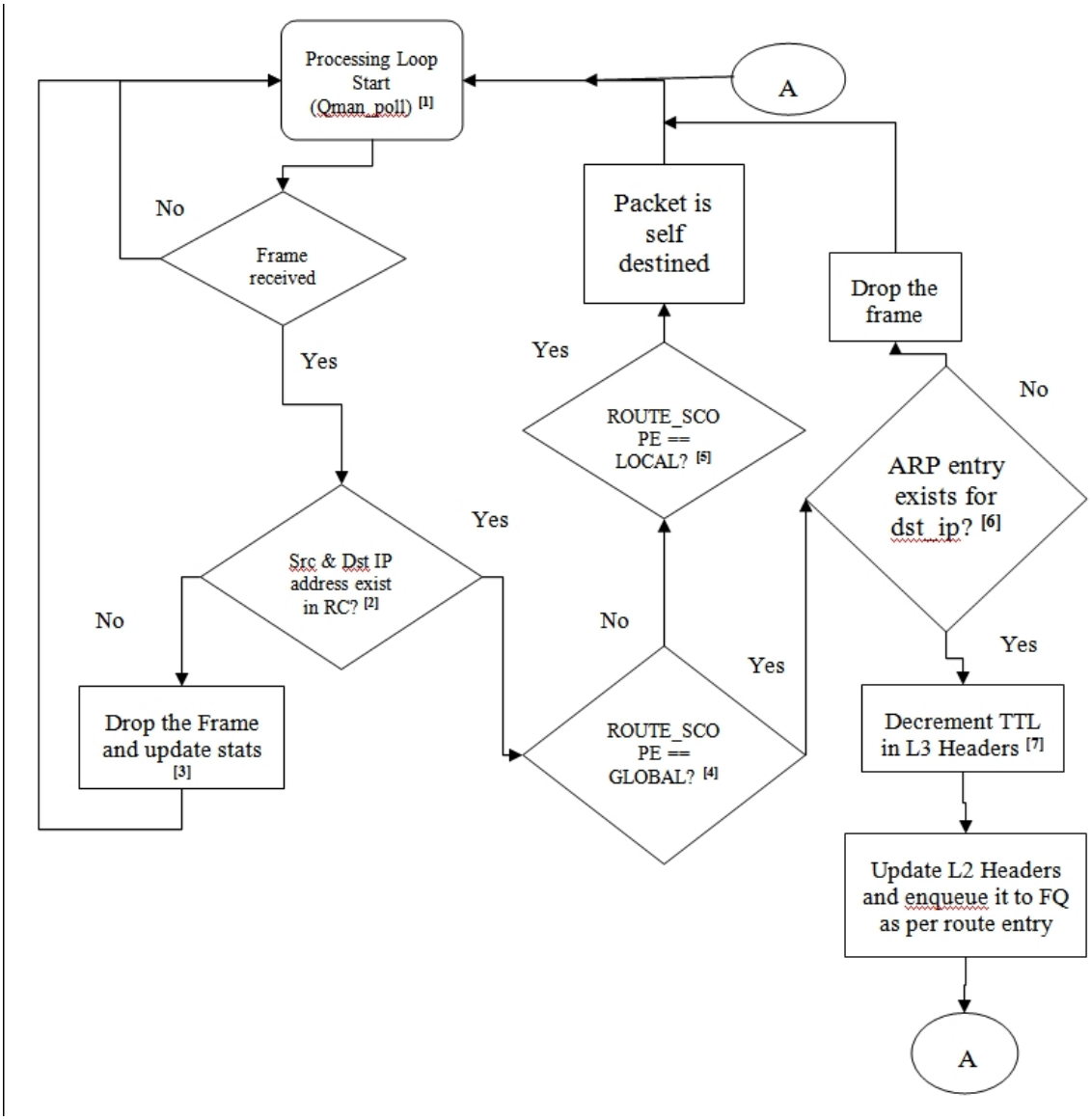
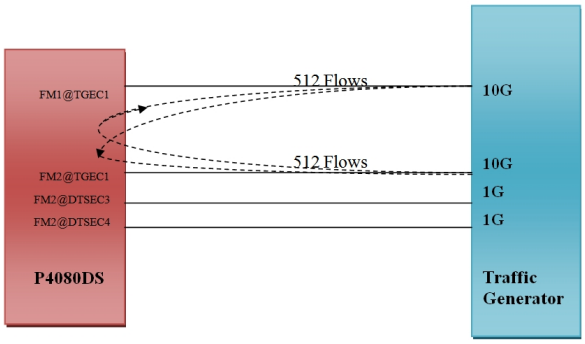
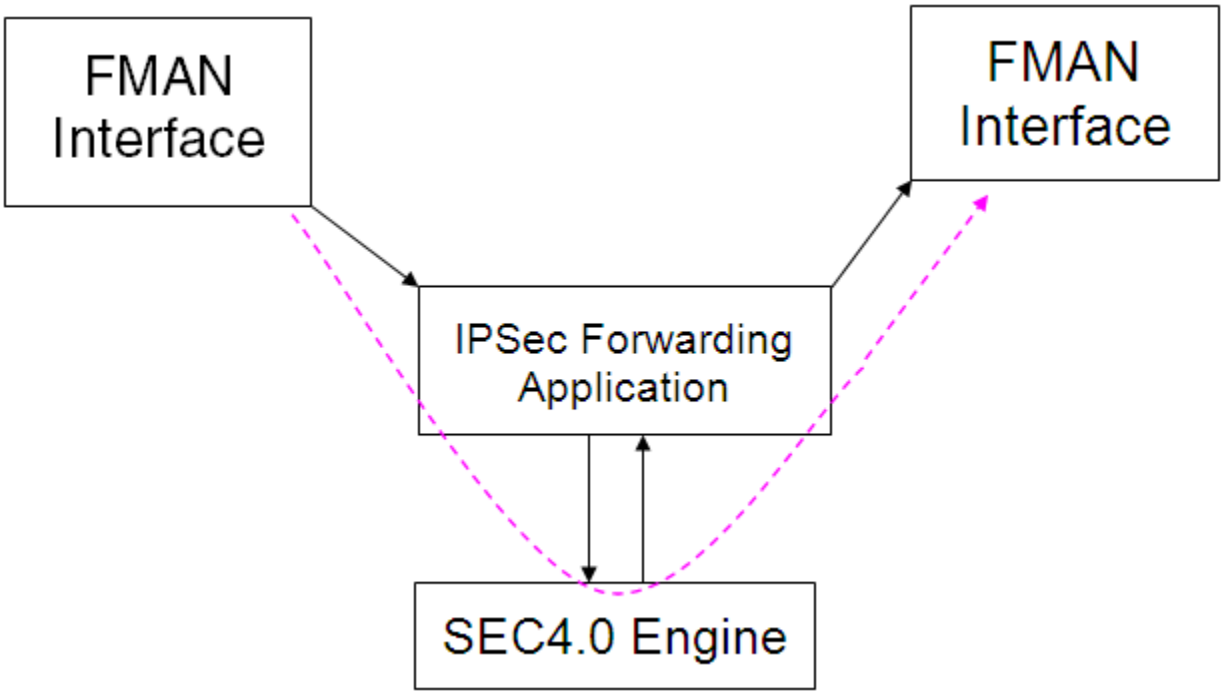
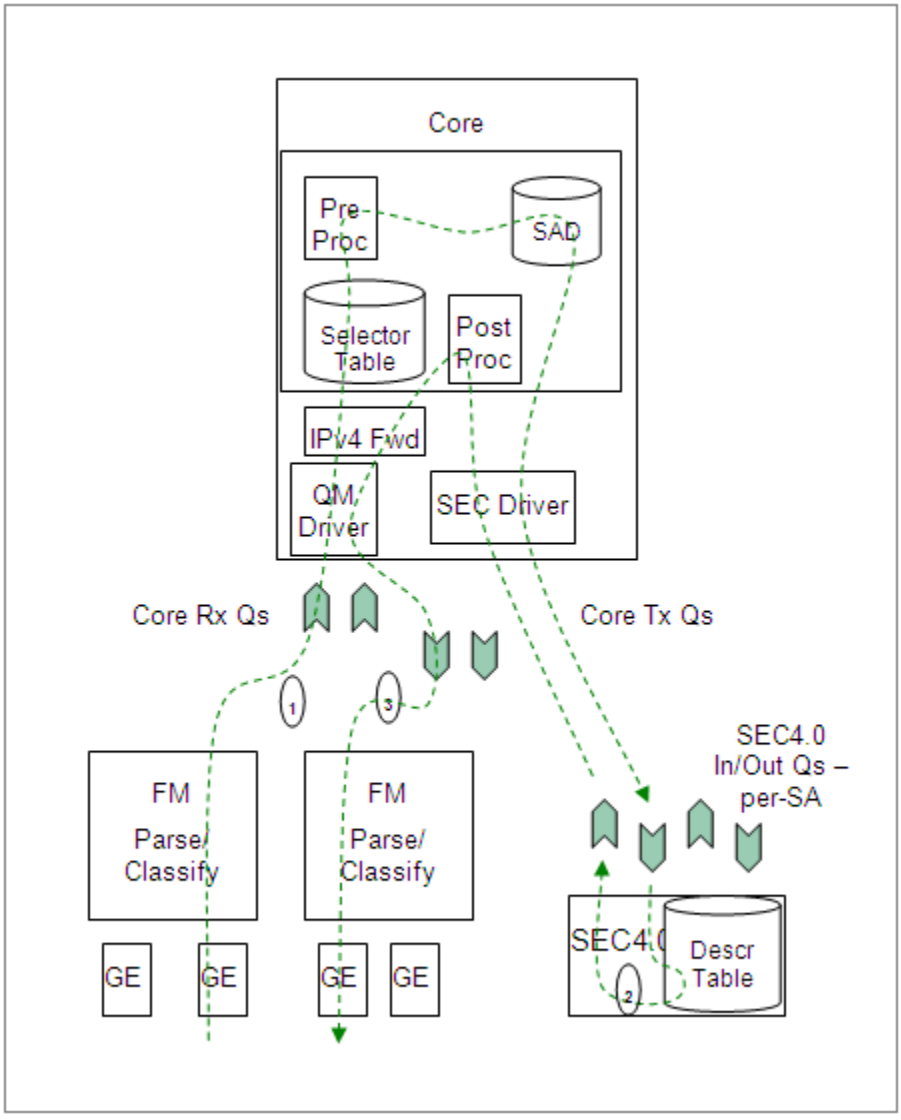
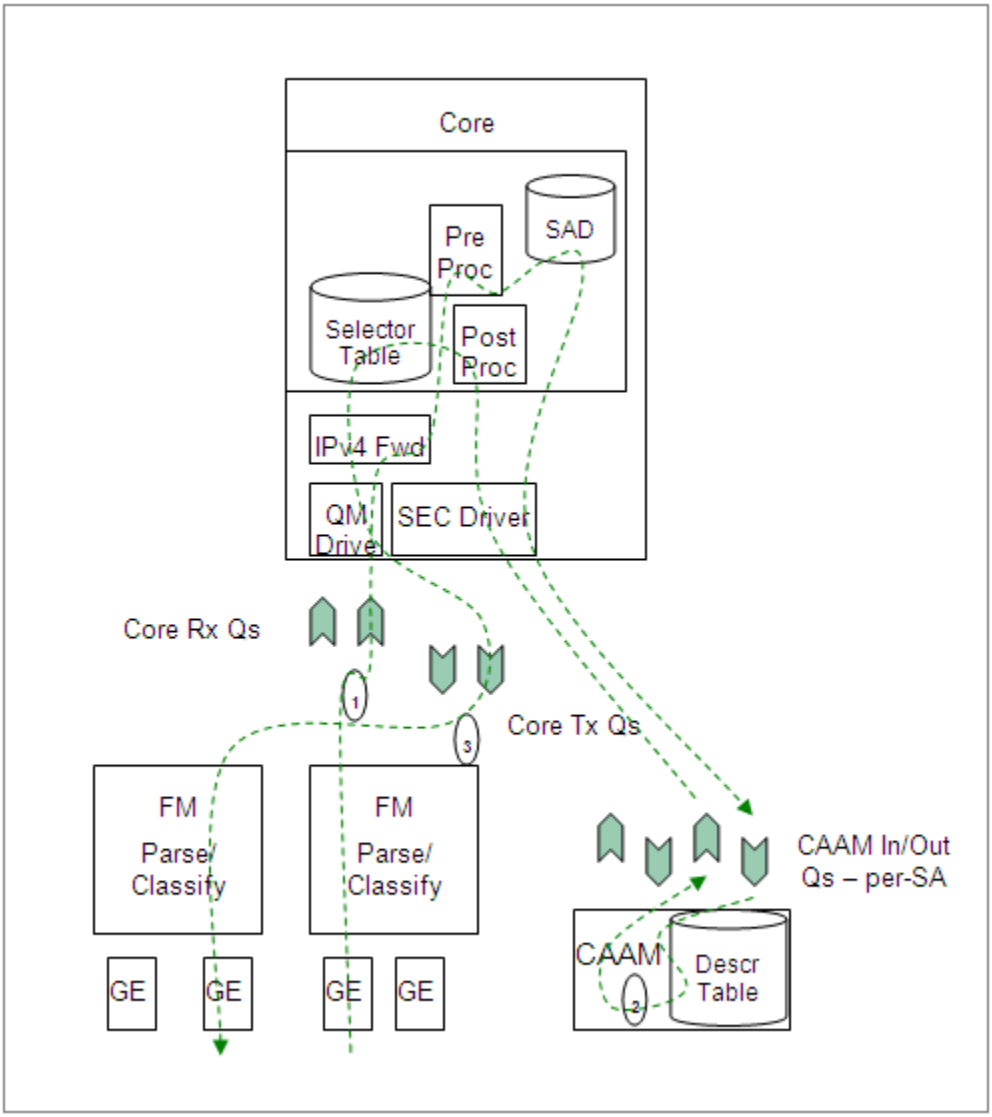
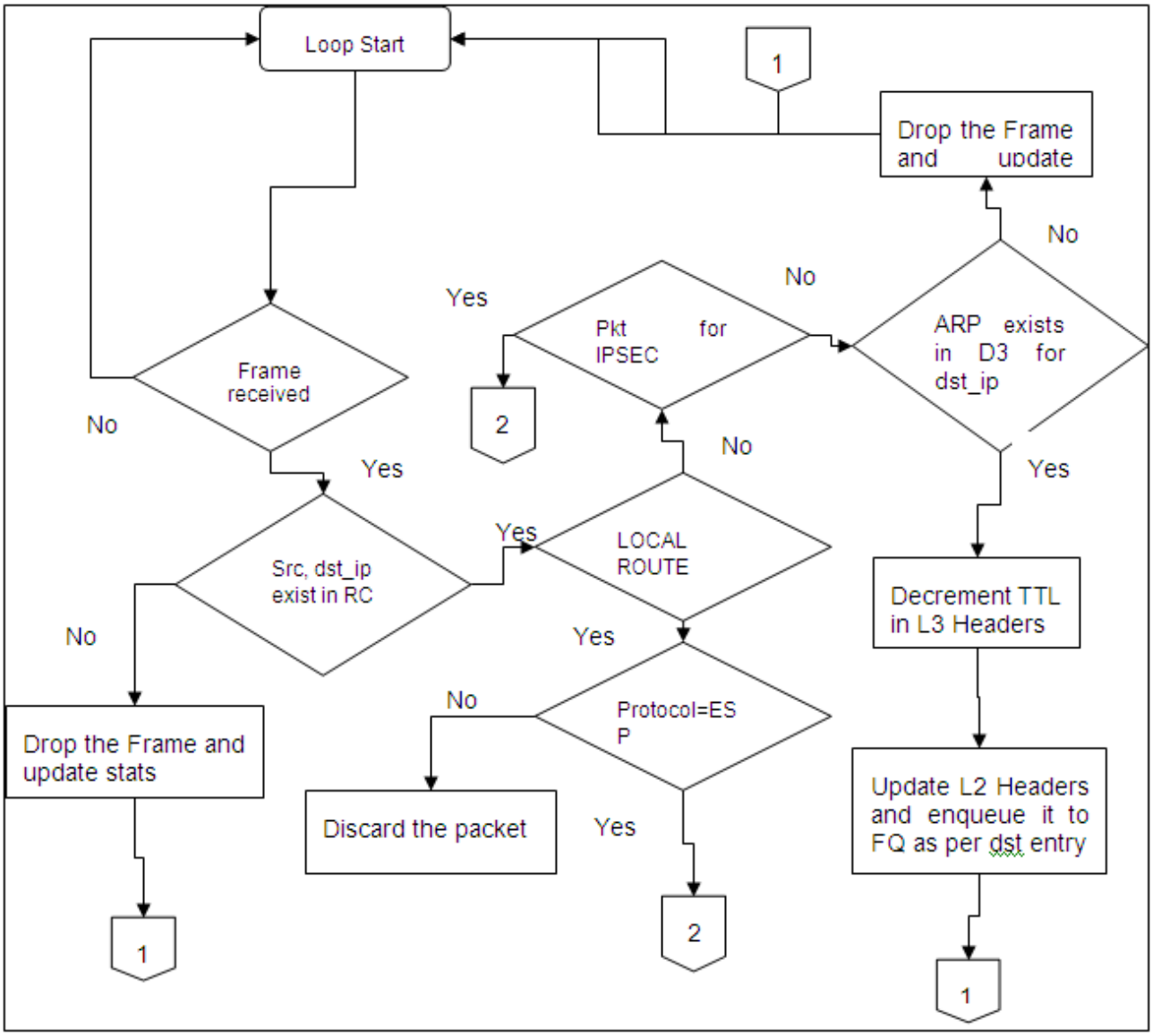
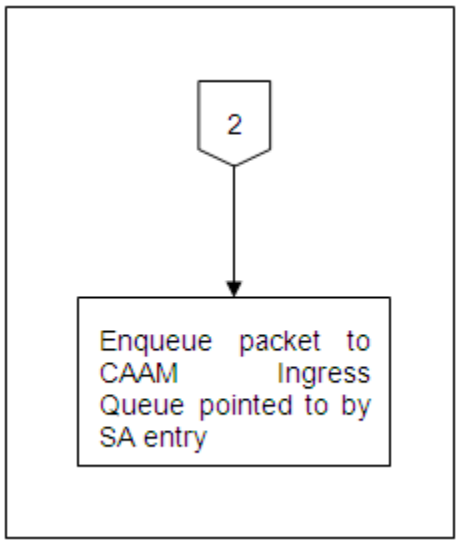
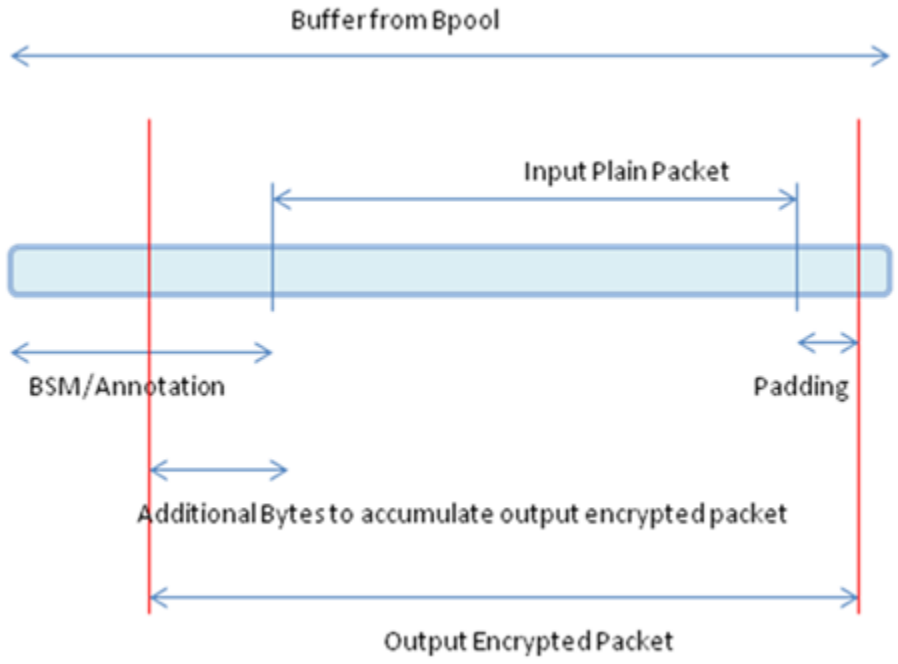
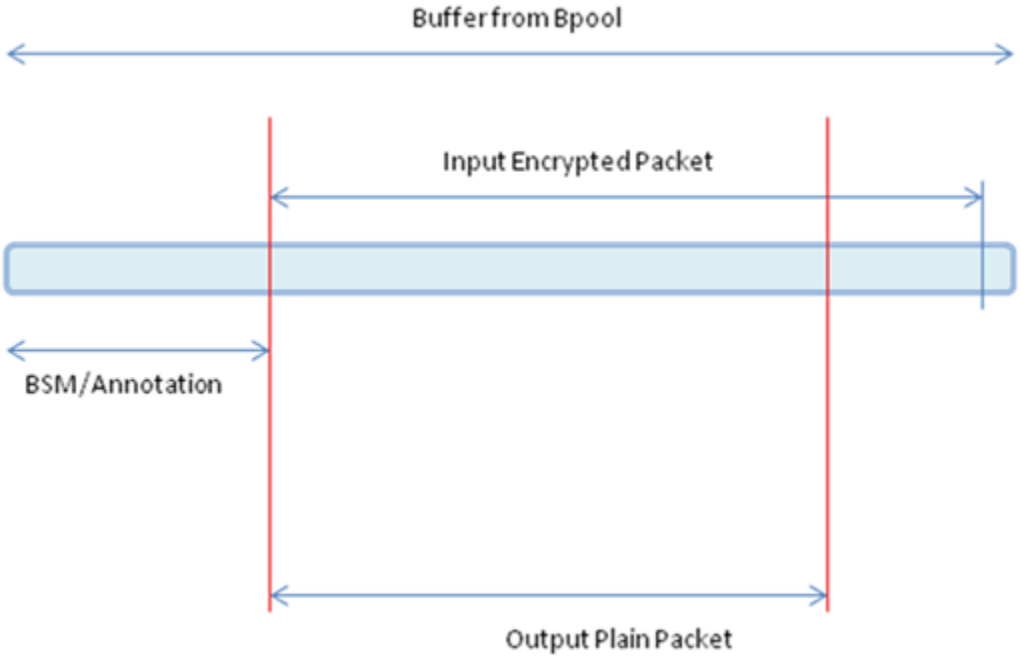
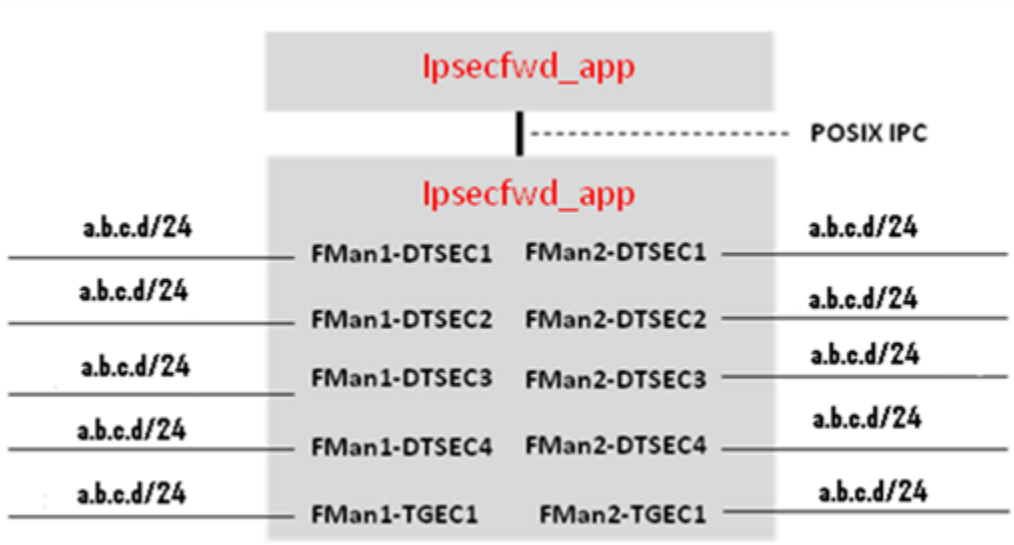
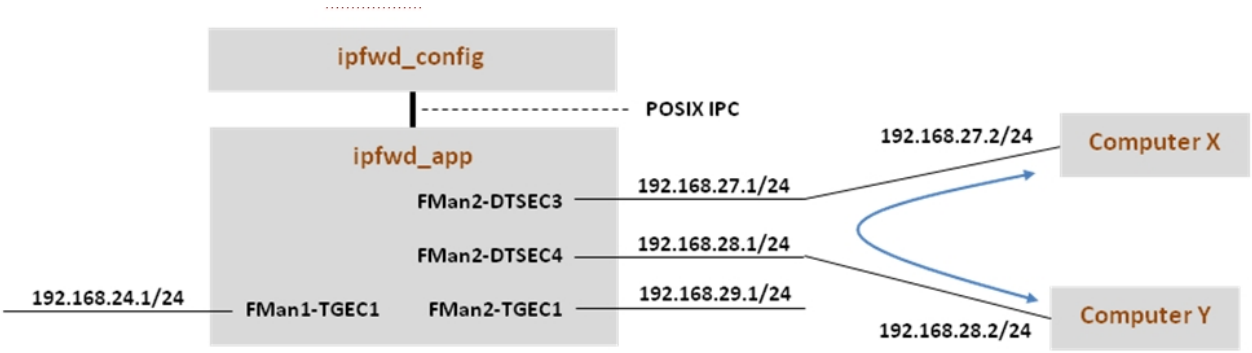
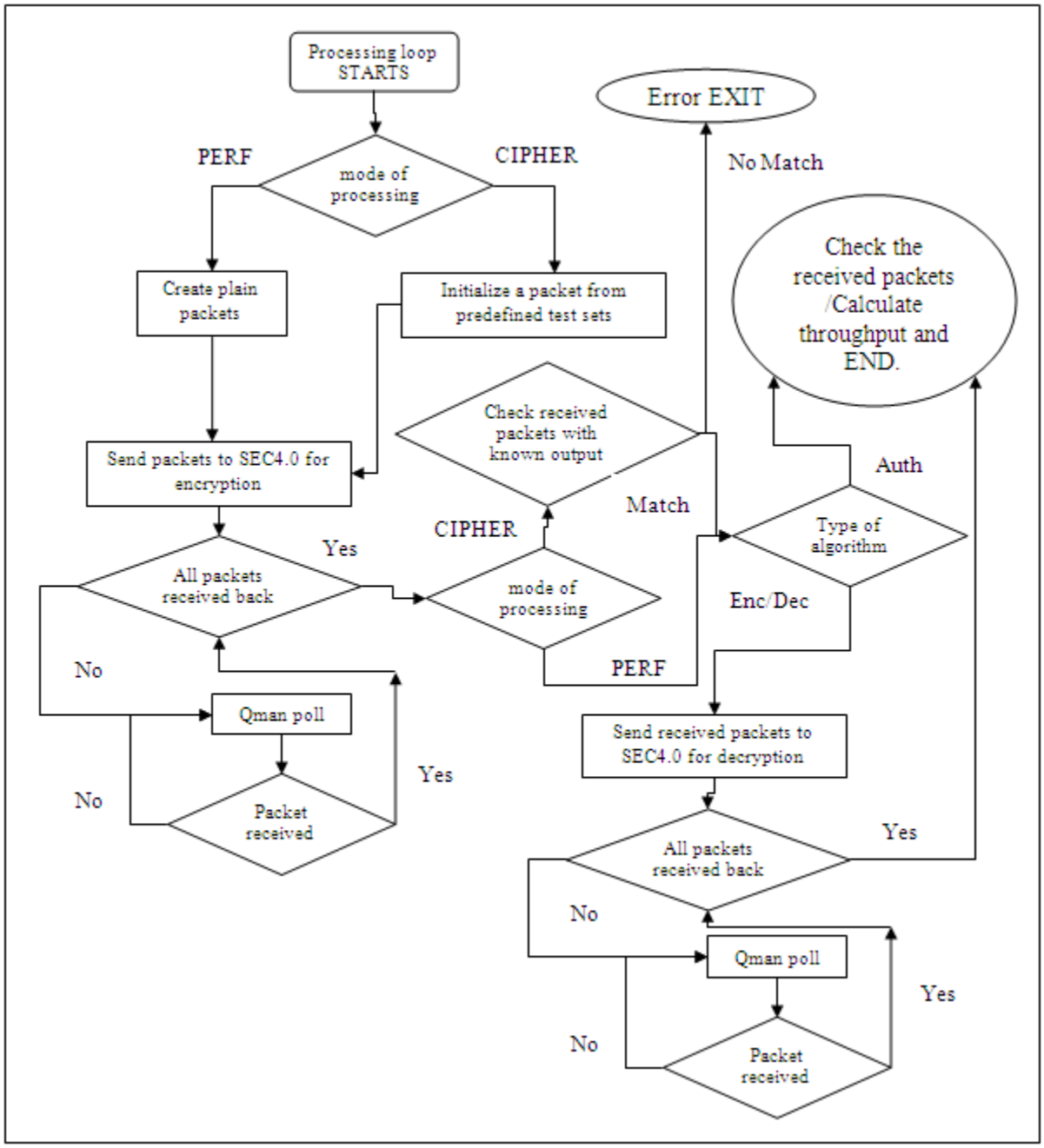
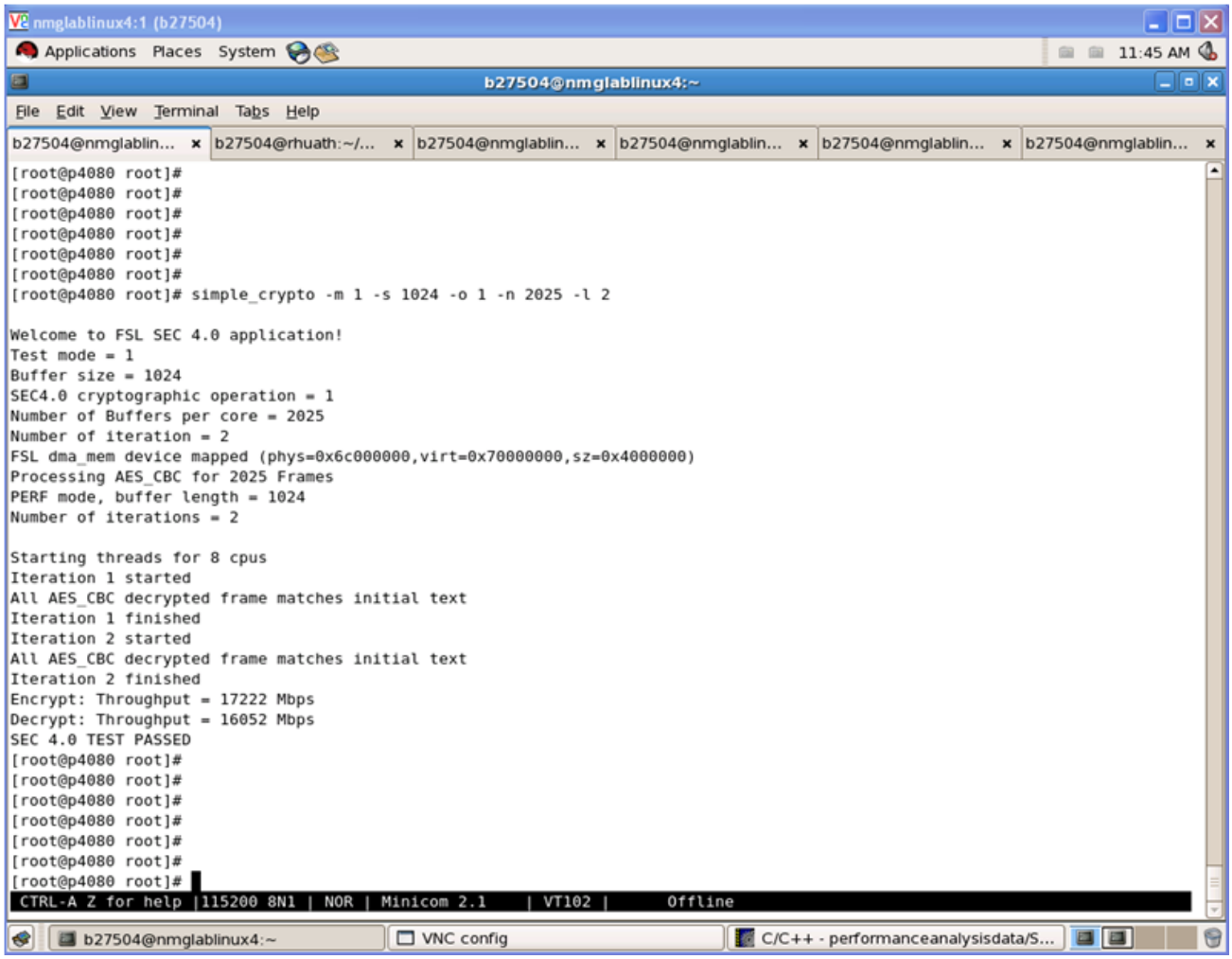
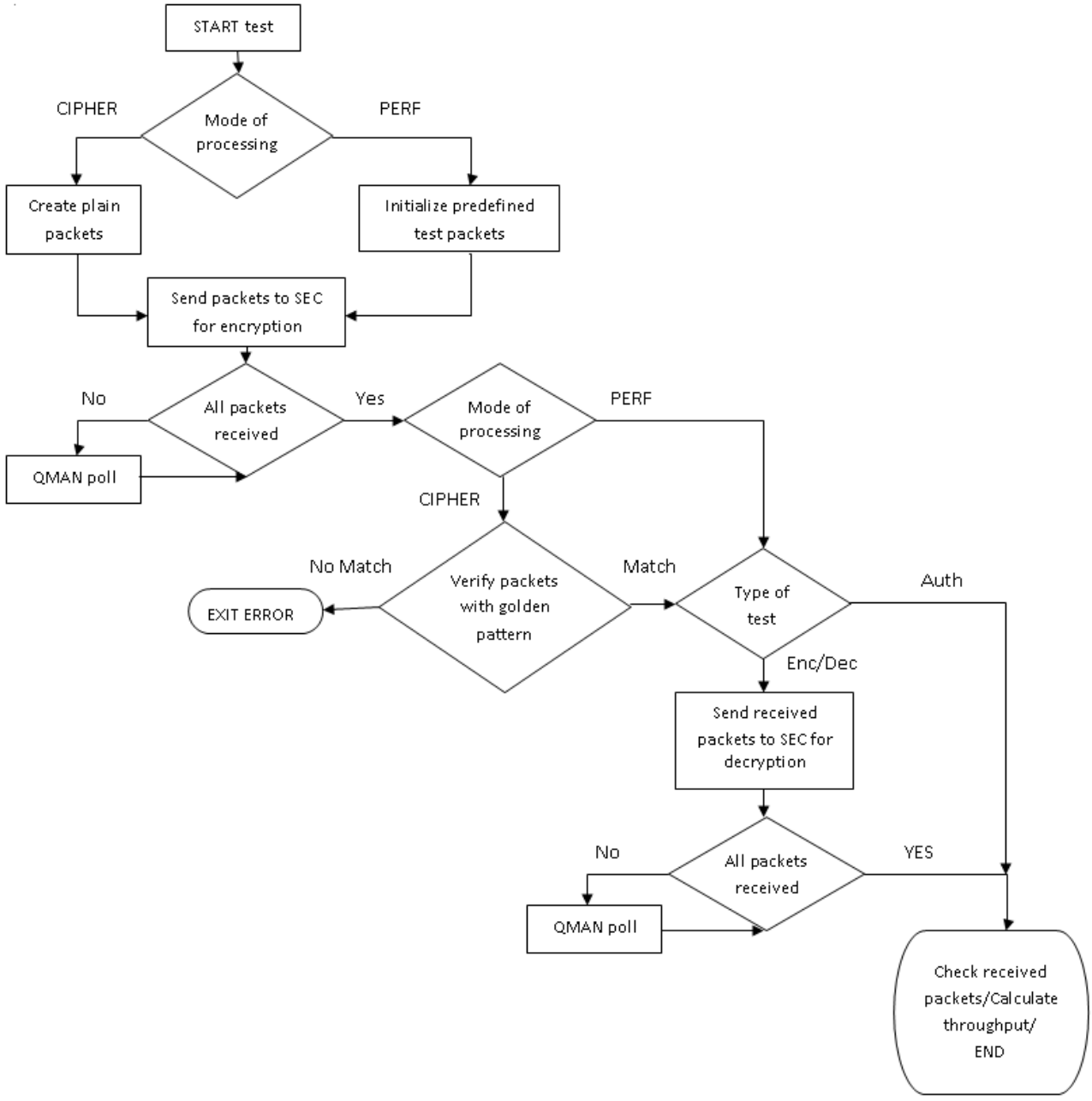
Port 6 to Port 2 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.20.2 - 192.168.20.11

Port 6 to Port 3 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.30.2 - 192.168.30.11

Port 6 to Port 4 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.40.2 - 192.168.40.11

Port 6 to Port 5 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.50.2 - 192.168.50.11
```

How to run if user has no XAUI but only SGMII card?

Assume user does not have a XAUI card and wants to run IPFwd using only the SGMII ports as shown below.



This configuration contains the 1 RGMII port (FMAN1, DTSEC 2), 4 SGMII ports (FMan2, DTSECs 1- 4). The user can modify the configuration file and sample shell scripts as per requirement. The configuration file would contain the following

```
<cfgdata>
<config>
<engine name="fm1">
<port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
<port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
<port type="1G" number="2" policy="hash_ipsec_src_dst_spi_policy8"/>
<port type="1G" number="3" policy="hash_ipsec_src_dst_spi_policy9"/>
</engine>
</config>
</cfgdata>
```

User can create a new shell script which would make routes between the 4x1G. Here is the content of the script.

```
net_pair_routes()
{
for net in $1 $2
      do
ipfwd_config -P $pid -B -s 192.168.$net.2 -c $3 \
-d 192.168.$(expr $1 + $2 - $net).2 -n $4 -g \
192.168.$(expr $1 + $2 - $net).2
done
}
ipfwd_config -P $pid -F -a 192.168.110.1 -i 6
```

```
ipfwd_config -P $pid -F -a 192.168.120.1 -i 7
ipfwd_config -P $pid -F -a 192.168.130.1 -i 8
ipfwd_config -P $pid -F -a 192.168.140.1 -i 9
ipfwd_config -P $pid -G -s 192.168.110.2  -m 02:00:c0:a8:6e:02 -r true
ipfwd_config -P $pid -G -s 192.168.120.2 -m 02:00:c0:a8:78:02 -r true
ipfwd_config -P $pid -G -s 192.168.130.2 -m 02:00:c0:a8:82:02 -r true
ipfwd_config -P $pid -G -s 192.168.140.2 -m 02:00:c0:a8:8c:02 -r true
# 1024
net_pair_routes 110 120  16  16   # 2 *  16 *  16 =  512
net_pair_routes 130 140 16 16   # 2 * 16 * 16 = 512
echo IPFwd Route Creation completed
```

Now traffic can be run as per the routes created.

9.9.6.9 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.



Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that ipfwd_app has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

```
ifconfig eth0 192.168.27.2 up
```

Set the default gateway for subnet 192.168.27.1

```
route add default gw 192.168.27.1 eth0
```

2. Add arp for gw

```
arp –s 192.168.27.1 “MAC address for 192.168.27.1 on P4080”
```

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

```
ifconfig eth0 192.168.28.2 up
```

Set the default gateway for subnet 192.168.28.1

route add default gw 192.168.28.1 eth0

2. Add arp for gw

arp –s 192.168.28.1 “MAC address for 192.168.28.1 on P4080”

The commands to perform on P4080 are:

Boot the P4080 and login as root.

Assign IP address to fm1-gb1

ifconfig fm1-gb1 <IPADD> up

cd /usr/etc

fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a

Assume use of cores 1 - 7 ipfwd_app 1..7

Now ssh to P4080 linux on other terminal

ssh root@<IPADD>

give IP address as assigned to fm1-gb1 in the beginning

Now assign ip address to the interfaces

First run command to check what all enabled interfaces are available

Note: check pid from application print "Message queue to send: /mq_snd_2536"

ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces

Assign IP address to the interfaces.Use the same interface number displayed as an ouput on giving the above command

```
ipfwd_config -P 2536 -F -a 192.168.24.1 -i 5
ipfwd_config -P 2536 -F -a 192.168.28.1 -i 9
ipfwd_config -P 2536 -F -a 192.168.27.1 -i 8
ipfwd_config -P 2536 -F -a 192.168.29.1 -i 11

# Now enter routes and MAC addresses. Format of a MAC address is
# aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

ipfwd_config -P 2536 -B -s 192.168.27.2 -d 192.168.28.2 -g 192.168.28.2
ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true
ipfwd_config -P 2536 -B -s 192.168.28.2 -d 192.168.27.2 -g 192.168.27.2
ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true
# Computer X and Computer Y need to be told to route via the P4080.
# On Computer X (assuming it runs Linux), enter this command as root:
route add -net 192.168.28.0 netmask 255.255.255.0 gw 192.168.27.1
# On Computer Y (assuming it runs Linux), enter this command as root:
route add -net 192.168.27.0 netmask 255.255.255.0 gw 192.168.28.1
# Now, traffic can pass between Computer X and Computer Y. For example, on Computer X
# enter:
ping 192.168.28.2
```

9.9.6.10 Flowchart for packet processing

This topic provides a flowchart for packet processing.



9.9.6.10.1 Description of Flow chart

1. All of the threads enter the processing loop where they call Qman_poll till a frame is received.
2. If a frame is received, the application checks whether the source and destination IP address exist in the Route cache. For this it calls rc_entry_fast_lookup(), which does the fast route look up by using the hash value provided by the FMan. The hash value is indexed into the route table. The IPFWD application does not change its route cache in response to seeing the first packet in a flow. Instead the route cache is set only by commands, as mentioned in section IPv4 forward application Configuration command on page 1720.
3. If the route entry for this frame is not present in the Route cache, the frame is dropped and statistics are updated. It then continues with Qman_poll.
4. If the route entry for this frame exists in the Route cache, it checks for the scope of the frame. If the scope is GLOBAL, the frame is sent for forwarding.
5. If the scope is not GLOBAL but LOCAL, the frame is self-destined. It then continues with Qman_poll.

6. If the frame is to be forwarded, check if ARP entry exists in ARP table for destination IP address. IPFWD application will not dynamically resolve the ARP. So if sending packets to IPFWD using a regular computer, the user will have to create static ARP entries.
7. If ARP entry exists, TTL is decremented in L3 header.
8. Finally, the L2 header is updated, which includes changing the dst MAC address in L2 header. The frame is then enqueued to the TX FQ.

9.9.6.10.2 Running IPv4 forwarding on P4080DS board

The instructions below describe how to run IPFWD. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipfwd_config application is completed.

- On linux prompt, assign IP address to fm1-gb1

$ ifconfig fm1-gb1 <IPADD> up

- Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a

To setup the FMan to distribute traffic to 1024 ingress frame queues per port:

$ fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_1024_fq.xml -a

- Run IPFWD application

The main IPFwd application binary is called **ipfwd_app** *. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:*

$ ipfwd_app <m..n>

By default ipfwd_app uses us_config_serdes_0xe.xml and us_policy_hash_ipv4_src_dst_32_fq.xml files.

To run for scenario 1024 ingress frame queues per port

$ ipfwd_app <m..n> -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_1024_fq.xml

IPFWD application command syntax:

[root@p4080 etc]# ipfwd_app --usage

Usage: ipfwd_app [-n?V] [-c FILE] [-d SIZE] [-i FILE] [-p FILE] [--fm-config=FILE]

[--non-interactive] [--fm-pcd=FILE] [--cpu-range] [--help]

[--usage] [--version] [cpu-range]

IPFWD application run command:

[root@p4080 root]# cd /usr/etc

<_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a

If the user wants to use all interfaces :

[root@p4080 etc]# ipfwd_app 1..7

If the user wants to use ONLY selective interfaces :

[root@p4080 etc]# ipfwd_app 1..7 -i fm1-10g,fm2-gb2

[1] 5363

[root@p4080 etc]# Found /fsl,dpaa/ethernet@9

Found /fsl,dpaa/ethernet@8

Found /fsl,dpaa/ethernet@7

Found /fsl,dpaa/ethernet@4

Qman: FQID allocator includes range 512:128

Bman: BPID allocator includes range 56:8

Configuring for 4 network interfaces and 4 pool channels

FSL dma_mem device mapped (phys=0xf8000000,virt=0x70000000,sz=0x4000000)

Thread uid:0 alive (on cpu 1)

Release 16384 bufs to BPID 7

Release 4096 bufs to BPID 8

Release 4096 bufs to BPID 9

ipfwd_app starting

Message queue to send: /mq_snd_2536

Message queue to receive: /mq_rcv_2536

Thread uid:1 alive (on cpu 2)

Thread uid:2 alive (on cpu 3)

Thread uid:3 alive (on cpu 4)

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

If, in the run application command, cpu-range is given i.e. "ipfwd_app <*m..n*>" IPFWD application starts threads on cpu-range m..n. The main thread (by default on CPU 1) then does global initialization needed by the application, including starting other application threads.

If, on the other hand, run application command is given without any cpu-range i.e. "ipfwd_app" IPFWD application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren't blocked):

> list

Thread alive on cpu 1

Thread alive on cpu 2

Thread alive on cpu 3

Thread alive on cpu 4

Thread alive on cpu 5

Thread alive on cpu 6

To enable all interfaces

> macs on

To disable all interfaces

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

- Once the application starts, it can receive the configuration commands. Run application configuration script.

For creating route entries, the binary *ipfwd_config* is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the IPFwd application.

The shell script mentioned below contains sample commands to add route entries. Detailed description of all ipfwd_config commands is provided in section IPv4 forward application Configuration command on page 1720.

SSH to p4080 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipfwd_20G.sh "pid"

$ ipfwd_20G.sh 2536

There are two example shell scripts available and those are "ipfwd_20G.sh" and "ipfwd_22G.sh". ipfwd_20G.sh creates routes for only the 2 x 10G interfaces and ipfwd_22G.sh creates routes for 2 x 10G and 2 x 1G interfaces. They can assign IP addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. The following table summarizes the settings done by these scripts. Check sectionPossible configuration scenario for IPFWD on page 1706 for more details.

For the IPFwd application to forward traffic successfully, traffic destined for the P4080DS ports must have the appropriate source and destination addresses.

Console messages are printed for each entry added to the routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "Application Started successfully" is printed on the console. At this point, traffic can be sent to the IPFWD application, which would do its processing on the cpu-range specified by the user on the application command-line.

9.9.6.10.3 Running IPv4 forwarding on P3041/P5020 board

The instructions below describe how to run IPFWD on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the ipfwd_config application is completed. · On linux prompt, assign IP address to eth0

$ ifconfig eth0 <IPADD> up

Running IPFWD

· Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -a

· Run IPFWD application

If the user wants to use all interfaces :

$ ipfwd_app <m..n> -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml

If the user wants to use ONLY selective interfaces :

$ipfwd_app m..n -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -i fm1-10g,fm1-gb4

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to p4080 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipfwd_15G.sh "pid"

$ ipfwd_15G.sh 2536

There is an example shell script available named as ipfwd_15G.sh creates routes for only the 5 x 1G and 1x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Check section Possible configuration scenario for IPFWD on page 1706 for more details. Now traffic can be run as per the routes created.

9.9.6.10.4 Running IPv4 forwarding on T4240 board

The instructions below describe how to run IPFWD on T4240. Traffic should only be directed to T4240 once the application is running and configuration via the ipfwd_config application is completed. · On linux prompt, assign IP address to eth0

$ ifconfig eth0 <IPADD> up

Running IPFWD

· Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -a

· Run IPFWD application

$ipfwd_app m..n -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000

For T4240, m..n can be 0..23.

SSH to t4240 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipfwd_20G.sh "pid"

$ ipfwd_20G.sh 2536

There is an example shell script available named as ipfwd_20G.sh creates routes for only the 2x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Now traffic can be run as per the routes created.

9.9.6.10.5 Running IPv4 forwarding on B4860 board

The instructions below describe how to run IPFWD on B4860. Traffic should only be directed to B4860 once the application is running and configuration via the ipfwd_config application is completed. · On linux prompt, assign IP address to eth0

$ ifconfig eth0 <IPADD> up

Running IPFWD

· Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml -a

· Run IPFWD application

$ipfwd_app m..n -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml

For B4860, m..n can be 0..7.

SSH to b4860 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipfwd_20G.sh "pid"

$ ipfwd_20G.sh 2536

There is an example shell script available named as ipfwd_20G.sh creates routes for only the 2x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Now traffic can be run as per the routes created.

9.9.6.10.6 USDPAA IP Fwd performance gap between 6 core and 8 core

USDPAA IPfwd performance for 8 core is less than 6 core on e6500 series. USDPAA IP Fwd application need to run using "-s" option to bridge the gap between two configuration.

9.9.6.10.7 PPAC (and IPFwd) CLI commands

The following commands are illustrated in the context of IPFwd, but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren't blocked):

> list

Thread uid:0 alive (on cpu 1)

Thread uid:1 alive (on cpu 2)

Thread uid:2 alive (on cpu 3)

Thread uid:3 alive (on cpu 4)

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

To remove a thread by its UID:

> rm uid:2

Thread uid:2 killed (cpu 3)

To remove a thread running on a given CPU:

> rm 5 Thread uid:4 killed (cpu 5)

To enable all interfaces:

> macs on

To disable all interfaces:

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

To query cgr

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0e1e

a_bcnt: 0x00_0000_0e1e

Tx CGR ID: 11, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_0002

a_bcnt: 0x00_0000_0004

9.9.6.11 IPv4 forward application Configuration command

9.9.6.11.1 Syntax

The syntax is as follows:

$ [root@p4080 bin]# ipfwd_config --help

Usage: ipfwd_config [OPTION...]

-B, --routeadd=TYPE adding a route

-C, --routedel=TYPE deleting a route

-E, --showintf=TYPE show interfaces

-F, --intfconf=TYPE change intf config

-G, --arpadd=TYPE adding a arp entry

-H, --arpdel=TYPE deleting a arp entry

-O, --Start/ Go=TYPE Start the processing of packets

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

9.9.6.11.1.1 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

Ipfwd_config -P pid -E -a true

Table 272. Field Description (show all enabled interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	Show all the interfaces	Yes	true

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# Ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces [root@p4080 bin]#

9.9.6.11.1.2 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

ipfwd_config -E -help

Help for show all enabled interfaces

[root@p4080 etc]# ipfwd_config -E --help

Usage: -E [OPTION...]

-a, --a=ALL All interfaces

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.6.11.1.3 Assign IP address to interfaces

The command to assign IP address while running IPv4 forward is as follows:

ipfwd_config -P pid -F -a 192.168.60.1 -i <Interface number>

NOTE

Note: The interface number to be used here must be one of the numbers that got displayed as the output of "show all enabled interfaces command" in section Command to show all enabled interfaces and their interface numbers on page 1721.

Table 273. Field description (assign IP address to interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	IP Address	Yes	a.b.c.d
-i	Interface number	Yes	0-11 (Choose this number from "show all enabled interfaces" command output)

Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

ipfwd_config -P 2536 -F -a 192.168.60.1 -i 5

IPADDR assigned = 0xc0a83c01 to interface num 5

Intf Configuration Changed successfully

9.9.6.11.1.4 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

ipfwd_config -F --help

Help for assign IP address to interfaces

[root@p4080 etc]# ipfwd_config -F --help

Usage: -F [OPTION...]

-a, --a=IPADDR IP Address

-i, --i=IFNAME If Name

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.6.11.1.5 Adding a Route Entry

The command to add a route while running IPv4 forward is as follows:

ipfwd_config -P pid -B -s a.b.c.d -d b.c.d.e -g c.d.e.f

Table 274. Field Description (Adding a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d
-g	Gateway IP Address	Yes	a.b.c.d

Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# ipfwd_config -P 2536 -B -s 192.168.29.2 -d 192.168.24.2 -g 192.168.24.2

Route Entry Added successfully

[root@p4080 bin]#

9.9.6.11.1.6 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

ipfwd_config -B --help

Help for Adding a Route Entry

[root@p4080 bin]# ipfwd_config -B --help

Usage: -B [OPTION]

-d, --d=DESTIP Destination IP

-f, --f=FLOWID Flow ID - (0 - 1024) {Default: 0}

-g, --g=GWIP Gateway IP

-s, --s=SRCIP Source IP

-t, --t=TOS Type of Service - (0 - 256) {Default: 0}

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.6.11.1.7 Deleting a Route Entry

The command to delete a route while running IPv4 forward is as follows:

ipfwd_config -P pid -C -s a.b.c.d -d b.c.d.e

Table 275. Field Description (Deleting a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d

Deleting a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# ipfwd_config -p 2536 -C -s 192.168.29.2 -d 192.168.24.2 Route Entry Deleted successfully

9.9.6.11.1.8 Help for Deleting a Route Entry

The command to obtain help for route entry deletion is as follows:

ipfwd_config -C --help

Help for Deleting a Route Entry

[root@p4080 bin]# ipfwd_config -C --help

Usage: -C [OPTION...]

-d, --d=DESTIP Destination IP

-s, --s=SRCIP Source IP

-t, --t=TOS Type of Service - (0 - 256) {Default: 0}

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.6.11.1.9 Starting the Application Processing

The command to start the application processing phase is as follows:

ipfwd_config -O

Starting the Application Processing

[root@p4080 bin]# ipfwd_config -O

Application Started successfully

9.9.6.11.1.10 Adding an ARP Entry

The command to add an ARP entry while running IPv4 forward is as follows:

ipfwd_config -P pid -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]

Table 276. Field Description (Adding an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d
-m	Mac Address	Yes	aa:bb:cc:dd:ee
-r	Replace existing entry	No	true/ false {Default: false}

Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# ipfwd_config -P 2536 -G -s 192.168.24.2 -m 02:00:c0:a8:33:fd -r true

ARP Entry Added successfully

9.9.6.11.1.11 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

ipfwd_config -G --help

Help for Adding an ARP Entry

[root@p4080 etc]# ipfwd_config -G --help

Usage: -G [OPTION...]

-m, --m=MACADDR MAC Address

-r, --r=Replace Replace Exiting Entry - true/ false {Default:

false}

-s, --s=IPADDR IP Address

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.6.11.1.12 Deleting an ARP Entry

The command to delete an ARP while running the IPv4 forward is as follows:

ipfwd_config -P pid -H -s a.b.c.d

Table 277. Field Description (Deleting an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d

Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# ipfwd_config -P 2536 -H -s 192.168.24.2

Arp Entry Deleted successfully

[root@p4080 bin]#

9.9.6.11.1.13 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

ipfwd_config -H --help

Help for Deleting an ARP Entry

[root@p4080 etc]# ipfwd_config -H --help

Usage: -H [OPTION...]

-s, --s=IPADDR IP Address

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.6.12 Traffic Generation



9.9.6.13 References

1. USDPAA PPAC User Guide
2. QMan/BMan API Guide

9.9.7 NXP USDPAA IPSecfwd User Manual

9.9.7.1 Introduction

The User Space Data Path Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA IPSecfwd application.
- Execution steps for the IPSecfwd application.

9.9.7.1.1 Purpose

This document describes the USDPAA IPSec forwarding application. This application documents the USDPAA IPSecfwd demonstration applications forwarding flow.

9.9.7.1.2 Change History

Table 278. Change History

Version	Updates
1.1	Creation of IPSecfwd UG for phase v1.0 release. - PPAC/PPAM overview - Basic application flow - Commands to use - Testing IPSecfwd application
1.2	Addition of t4/b4 sections

9.9.7.2 USDPAA IPSecfwd application

The USDPAA IPSec forwarding (IPSecfwd) application is a multi-threaded application that routes IPv4 packets from one Ethernet interface to another after performing encryption/decryption if required on P4080/P3041/P5020/T4240/B4860 systems. The configuration done for the system and the type of the packet is used to determine the type of operation to be performed on the packet. The routing is done based on the source IP address and destination IP address in the frame.

Any combination of the 8 cores on the P4080 can run a USDPAA IPSec Forwarding application thread in USDPAA SDK v1.0.

9.9.7.2.1 Application Overview

The IPSecfwd application can route IPv4 traffic directly over interfaces as well as over the IPSec tunnel between two network nodes connected over different subnets with the assistance of the IPSec protocol security provided by SEC4.0 block. The application works as an extension of the IPv4 forwarding application with encryption/decryption in addition to route lookup and packet forwarding. The IPSec processing tunnel table contains information on frame queues toward SEC4.0 for the IPSec protocol processing of the packet before it is sent out on the egress interface. IPsec can use an extended sequence number (ESN) optionally that is authenticated but not transmitted. If an ESN is found in the PDB, it is given to the authentication CHA in the last. The ESN is incremented whenever the Seq Num rolls over.

The IPSecfwd application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAA PPAC User Guide for more details). The configuration phase also includes PPAM (i.e. IPSecfwd) related initialization. Once the configuration phase is completed the IPSecfwd application moves to the packet processing phase.

The application provides a command-line interface to enable users to add and remove routing table, ARP cache entries and SA entries at any given time. For each user input, the appropriate information is communicated to the IPSecfwd application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPSecfwd application. Note that the IPSecfwd application does not dynamically resolve ARP - missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking ìIRQ modeî whenever IPSecfwd has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to ìUSDPAA PPAC User Guideî. In polling mode - the application constantly looks for data to process on its dedicated QMan portal. Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

9.9.7.2.2 Packet Flow

The IPSecfwd application is capable of routing packets with the security processing support from the P4080's SEC4.0 engine. After all the initialization and configurations, each core enters the polling loop and polls for packets from FMan(PPAC), or SEC4.0 (PPAM). The loop dequeues packets from the frame queue associated to pool channels shared by the application cores.

Once a frame is dequeued, it is sent for processing based on callback handler in contextB of the frame queue response ring entry. When the frame is ingressed from FMAN, then the callback handler is called which performs the route lookup and tunnel lookup; and then sends the packet to SEC4.0 block/FMan. For the frame returning from SEC4.0, the encap callback handler is called for a frame coming after encryption or decap callback handler is called for a frame coming after decryption.

The steps for the packet flow for ingress from FMan are as follows:

1. Once the packet is dequeued from FMan, DQRR callback handler associated with Fman is called.
2. After a basic sanity check of the packet (packet header is correct, packet checksum is correct etc.), the packet is checked for the ESP protocol.
    a. If the packet is ESP, a direct tunnel lookup happens based on FMan's PCD hash value and the frame is sent to SEC4.0 engine

b. If the packet is non-ESP, a route lookup based on FMan's PCD happens. If there is an SA entry, it is sent to SEC4.0 block after updating annotations in frame with next level route destination.

c. If the packet is non-ESP and the route is normal, the packet is sent to FMan interface.

d. If no route is found, the packet is discarded.

3. When the packet is dequeued from SEC4.0 Engine, the DQRR callback handler associated with SEC4.0 frame queues is called. Encap callback handler or Decap callback handler will be called from based on the callback handler in contextB of DQRR ring entry.

4. The packet is now taken directly to the FMan interface as the destination field was already filled while sending the packet to the SEC4.0 Engine.

A typical packet flow in a simulated P4080 environment is shown below.



Figure 262. IPSecfwd Application basic packet flow

9.9.7.2.3 Overview of IPSecfwd packet processing

IPSecfwd Steady state processing or Forwarding consists of two parts

- Outbound processing: when un-encrypted packets are received by the system and tunneled using IPSecfwd application.
- Inbound processing: when encrypted packets are received by the systems and are decrypted by the system.

9.9.7.2.3.1 Outbound processing:

1. *Pre-processing:*

   a. Packet is received by the FM which uses 2-tuple (Src IP, Dest IP) to hash the packets to different Core Rx Queues.

   b. This packet is picked up by the core and first subjected to IPv4 Forwarding lookup.

   c. If the action is to do IPSec processing, then it looks up for a corresponding entry in the SADB, which has the SA information and Queue-ids for the SA used by SEC4.0.

2. *Crypto-processing:*

   a. SEC4.0 dequeues the job from egress Queue toward SEC4.0.

b. The SEC4.0 encrypts, authenticates and adds the ESP header and outer IP header to the packet. It then enqueues the processed packet to the Ingress Queue from SEC4.0.

3. *Post-processing*:

a. The core now dequeues the processed packet from the SEC4.0 on Out FQ and sends it to the IPv4 Forwarding module (since there is a new Outer-IP header).

b. IPv4 Forwarding module does a lookup and transmits the packet from the egress interface.

c. If the SEC4.0 result indicates an error in the Crypto processing of the packet, it is discarded and statistics (packet, octets, and errors - per SA / Interface) are updated.



Figure 263. IPSecfwd Outbound processing

9.9.7.2.3.2 Inbound Processing:

1. Pre-processing:

a. The IPSec tunneled packet is received by the FM, which uses a 3-tuple (Dest-IP, Src-IP, SPI) to hash the packet to different Core Rx Queues.

b. The packet is picked up and undergoes IPv4 Forwarding lookup. If packet is self-destined, and protocol is ESP, it is sent to IPSec Forwarding module for processing.

c. The SA info in SADB yields a SEC4.0-In Queue for the SA. The packet is sent to that SEC4.0 on its ingress FQ.

2. Crypto-processing:

a. The SEC4.0 decrypts, authenticates and enqueues the processed packet to the SEC4.0-Out Queue found in its input FQs contextB.

3. Post-processing:

a. The core de-queues the processed packet from the SEC4.0-Out Queue, and subjects the packet to a Selector-table lookup.

b. The packet is handed over to IPv4 Forwarding module, which does a route cache lookup and transmits the packet from the egress interface.

c. If the SEC4.0 result indicates an error in the Crypto processing of the packet, it is discarded and statistics are updated.



Figure 264. IPSecfwd Inbound processing

9.9.7.2.4 Flow chart for IpSecfwd packet processing



Figure 265. Flow chart for IpSecfwd packet processing

sep text



Figure 266. IPSecfwd packet processing

9.9.7.3 Overview of PPAC

The source code to IPSecfwd has been reorganized into two parts; the "PPAC" (Packet-Processing Application Core) and a "PPAM" (Packet-Processing Application Module). The PPAM portion implements the IPSecfwd application specific logic of processing the packet and forwarding it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document "USDPAA PPAC User Guide".

9.9.7.4 IPSecfwd related PPAM Details

9.9.7.4.1 In-Place Encryption/Decryption

IPSecfwd application uses in-place Encryption/Decryption when packet is sent to SEC4.0 block. In-place Encryption/ Decryption is supported in IPSecfwd application by using the same output buffer as the input buffer.

In case of Encryption the size of the output packet is more than the size of input packet due to addition of tunnel header, padding of extra bytes etc. The FMam is configured to acquire the buffer from BMan which is large enough to accommodate the output packet after encryption.



Figure 267. IPSecfwd In-place Encryption

In case of Decryption the output packet size is smaller than the Input packet size. So the output plain packet in case of Decryption can easily be accommodated in the buffer acquired by FMan for storing the Input packet.



Figure 6: IPSecfwd In-place Decryption

Figure 268. IPSecfwd In-place Decryption

9.9.7.5 Secfwd application suite

The figure below shows the structure of the IPSecfwd USDPAA application suite. Its purpose is to encrypt/decrypt and forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using ipsecfwd_config command as mentioned in section 5.3.1.3 IP address can be assigned to all these interfaces. Each interface has a fixed netmask shown in the figure. The notation "/24" refers to a netmask of 255.255.255.0. The MAC addresses of these interfaces are determined by u-boot environment variables ethaddr, eth1addr, eth2addr, etc.



Figure 269. IPSecfwd application suite

The ipsecfwd application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An ipsecfwd_config command should be used to set these MAC addresses.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "NXP DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAA applications. See the USDPAA User Guide for more information.
3. Finally, the fmc configuration file passed by command line argument to ipsecfwd_app determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by ipsecfwd_app are:

- FMan1-TGEC1
- FMan2-DTSEC3
- FMan2-DTSEC4
- FMan2-TGEC1

Running the ipsecfwd application suite involves four steps:

1. Run the fmc application to configure the FMan hardware instances.
2. Run ipsecfwd_app
3. Run ipsecfwd_config repeatedly to add SA Entries/routes.

Specific examples showing these steps are provided in other sections of this document.

9.9.7.5.1 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.



Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that ipfwd_app has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.27.2 up

Set the default gateway for subnet 192.168.27.1

route add default gw 192.168.27.1 eth0

2. Add arp for gw

arp –s 192.168.27.1 “MAC address for 192.168.27.1 on P4080”

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.28.2 up

Set the default gateway for subnet 192.168.28.1

route add default gw 192.168.28.1 eth0

2. Add arp for gw

arp –s 192.168.28.1 “MAC address for 192.168.28.1 on P4080”

The commands to perform on P4080 are:

Boot the P4080 and login as root.

Assign IP address to fm1-gb1

ifconfig fm1-gb1 <IPADD> up

cd /usr/etc

fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a

Assume use of cores 1 - 7 ipfwd_app 1..7

Now ssh to P4080 linux on other terminal

ssh root@<IPADD>

give IP address as assigned to fm1-gb1 in the beginning

Now assign ip address to the interfaces

First run command to check what all enabled interfaces are available

Note: check pid from application print "Message queue to send: /mq_snd_2536"

ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces

Assign IP address to the interfaces.Use the same interface number displayed as an ouput on giving the above command

ipfwd_config -P 2536 -F -a 192.168.24.1 -i 5

ipfwd_config -P 2536 -F -a 192.168.28.1 -i 9

ipfwd_config -P 2536 -F -a 192.168.27.1 -i 8

ipfwd_config -P 2536 -F -a 192.168.29.1 -i 11

Now enter routes and MAC addresses. Format of a MAC address is

aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

ipfwd_config -P 2536 -B -s 192.168.27.2 -d 192.168.28.2 -g 192.168.28.2

ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true

ipfwd_config -P 2536 -B -s 192.168.28.2 -d 192.168.27.2 -g 192.168.27.2

ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true

Computer X and Computer Y need to be told to route via the P4080.

On Computer X (assuming it runs Linux), enter this command as root:

route add -net 192.168.28.0 netmask 255.255.255.0 gw 192.168.27.1

On Computer Y (assuming it runs Linux), enter this command as root:

route add -net 192.168.27.0 netmask 255.255.255.0 gw 192.168.28.1

Now, traffic can pass between Computer X and Computer Y. For example, on Computer X

enter:

ping 192.168.28.2

9.9.7.5.2 Running IPSecfwd on P4080DS board

The instructions below describe how to run IPSecfwd. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipfsecwd_config application is completed.

•On linux prompt, assign IP address to fm1-gb1

```
$ ifconfig fm1-gb1 <IPADD> up
```

•Configure FMan PCD using fmc with the XML files in /usr/etc:

```
    $ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
    $ fmc -c usdpaa_config_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPSecfwd application

The main IPSecfwd application binary is called **ipsecfwd_app**. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:

```
$ ipsecfwd_app <m..n>
```

By default ipsecfwd_app uses usdpaa_config_serdes_0xe.xml and usdpaa_policy_hash_ipv4.xml files.

IPSECFWD application command syntax:

```
[root@p4080 etc]# ipsecfwd_app --usage
Usage: ipsecfwd_app [-n?V] [-c FILE] [-p FILE] [--fm-config=FILE]
            [--non-interactive] [--fm-pcd=FILE] [--cpu-range] [--help]
            [--usage] [--version] [cpu-range]
```

IPSECFWD application run command:

```
[root@p4080 root]# cd /usr/etc
<_config_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
[root@p4080 etc]# ipsecfwd_app 1..7
[1] 5363
ipsecfwd_app starting
Message queue to send: /mq_snd_2536
Message queue to receive: /mq_rcv_2536
```

If in the run application command, cpu-range is given i.e. ìipsecfwd_app <m..n>î IPSecfwd application starts threads on cpu-range m..n. The main thread (by default on CPU1), then does global initialization needed by the application, including starting other application threads.

If on the other hand run application command is given without any cpu-range i.e. ìipsecfwd_appî IPSecfwd application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

•Once the application starts it can receive the configuration commands. Run application configuration script

For creating SA entries, the binary **ipsecfwd_config** is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the IPSecfwd application.

The shell script mentioned below contains sample commands to add SA entries.

SSH to p4080 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipsecfwd_20G.sh "pid"

```
$ ipsecfwd_enc_20G.sh 2536 or
$ ipsecfwd_dec_20G.sh 2536
```

One of the example shell scripts available is ìipsecfwd_enc_20G.shî which creates SA Entries for encryption for only the 2 x 10G interfaces. They can assign IP addresses to the interfaces, add SA and ARP entries and assumes the netmask to be 255.255.255.0. The following table summarizes the settings done by this script.

Table 279. ipsecfwd_enc_20G.sh

Port ID	Source IP Address	Destination IP Address (for each src IP addr)	Source tunnel address	Destination tunnel address	Default Gateway IP Address
4	192.168.60.2 to 192.168.60.24	192.168.160.2 .. 24	192.168.60.2	192.168.60.1	192.168.160.2
9	192.168.160.2 to 192.168.160.24	192.168.60.2 .. 24	192.168.160.2	192.168.160.1	192.168.60.2

For the IPSecfwd application to send out traffic successfully, traffic destined for the P4080DS ports must have the appropriate source and destination addresses.

Console messages are printed for each entry added to the SA/routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "Application Started successfully" is printed on the console. At this point, traffic can be sent to the IPSecfwd application which would do its processing on the cpu-range specified by the user on the application command-line.

9.9.7.5.3 Running IPSec forwarding on P3041/P5020 board

The instructions below describe how to run *IPSecfwd* on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the *ipsecfwd_config* application is completed.

•On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

•Configure FMan PCD *using fmc* with the XML files in /usr/etc:

```
    $ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
    $ fmc -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPSecfwd application

```
    $ ipsecfwd_app <m..n> -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml
```

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to p3041/p5020 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipsecfwd_mix_15G.sh "pid"

```
    $  ipsecfwd_mix_15G.sh 2536
```

This is an example shell script available which creates SA Entries for the 5 x 1G and 1x10G interfaces for back to back configuration.

9.9.7.5.4 Running IPSec forwarding on T4240 board

The instructions below describe how to run *IPSecfwd* on T4240. Traffic should only be directed to T4240 once the application is running and configuration via the *ipsecfwd_config* application is completed.

•On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

•Configure FMan PCD *using fmc* with the XML files in /usr/etc:

```
    $ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
    $ fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPSecfwd application

```
    $ ipsecfwd_app <m..n> -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -
d 0x10000000 -b 0:0:1024
```

For T4240, m..n can be 0..23.

SSH to t4240 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipsecfwd_mix_20G.sh "pid"

```
$  ipsecfwd_mix_20G.sh 2536
```

This is an example shell script available which creates SA Entries for the 2x10G interfaces for back to back configuration.

9.9.7.5.5 Running IPSec forwarding on B4860 board

The instructions below describe how to run *IPSecfwd* on B4860. Traffic should only be directed to B4860 once the application is running and configuration via the *ipsecfwd_config* application is completed.

•On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

•Configure FMan PCD *using fmc* with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPSecfwd application

```
$ ipsecfwd_app <m..n> -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml
```

For B4860, m..n can be 0..7.

SSH to b4860 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipsecfwd_mix_20G.sh "pid"

```
$  ipsecfwd_mix_20G.sh 2536
```

This is an example shell script available which creates SA Entries for the 2x10G interfaces for back to back configuration.

9.9.7.5.6 PPAC (and IPSecfwd) CLI commands

The following commands are illustrated in the context of IPSecfwd, but the commands are common to all PPAC-based applications.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren't blocked):

> list

Thread alive on cpu 1

Thread alive on cpu 2

Thread alive on cpu 3

Thread alive on cpu 4

Thread alive on cpu 5

Thread alive on cpu 6

To enable all interfaces

> macs on

To disable all interfaces

> macs off

To perform a controlled shutdown of ipsecfwd (this includes disabling the network ports):

> quit

9.9.7.5.7 IPSecfwd application Configuration command

9.9.7.5.7.1 Syntax

The syntax is as follows:

```
$ [root@p4080 bin]# ipsecfwd_config --help
Usage: ipsecfwd_config [OPTION...]
  -A  --SAadd=TYPE           adding an SA entry
  -B, --routeadd=TYPE        adding a route
  -C, --routedel=TYPE        deleting a route
  -D  --SAdel=TYPE           deleting an SA entry
  -E, --showintf=TYPE        show interfaces
  -F, --intfconf=TYPE        change intf config
  -G, --arpadd=TYPE          adding a arp entry
  -H, --arpdel=TYPE          deleting a arp entry
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

9.9.7.5.7.1.1 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

Ipfwd_config -P pid -E -a true

Table 280. Field Description (show all enabled interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	Show all the interfaces	Yes	true

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# Ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces [root@p4080 bin]#

9.9.7.5.7.1.2 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

```
ipsecfwd_config -E -help
Example 2. Help for show all enabled interfaces
 [root@p4080 etc]# ipsecfwd_config -E --help
Usage: -E [OPTION...]
  -a, --a=ALL               All interfaces
  -?, --help                Give this help list
      --usage               Give a short usage message
  -V, --version             Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.7.5.7.1.3 Assign IP address to interfaces

The command to assign IP address while running IPSecfwd is as follows:

```
ipsecfwd_config -P pid  -F -a 192.168.60.1 -i <Interface number>
```

Note: The interface number to be used here must be one of the numbers that got displayed as the output of "show all enabled interfaces command" in section 2.8.1.1.

Table 281. Field description (assign IP address to interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	IP Address	Yes	a.b.c.d

Table continues on the next page...

Table 281. Field description (assign IP address to interfaces) (continued)

-i	Interface number	Yes	0-11 (Choose this number from "show all enabled interfaces" command output)

Example 3. Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
ipsecfwd_config -P 2536 -F -a 192.168.60.1 -i 5
IPADDR assigned = 0xc0a83c01 to interface num 5
Intf Configuration Changed successfully
```

9.9.7.5.7.1.4 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

```
ipsecfwd_config -F --help
```

Example 4. Help for assign IP address to interfaces

```
[root@p4080 etc]# ipsecfwd_config -F --help
Usage: -F [OPTION...]
  -a, --a=IPADDR             IP Address
  -i, --i=IFNAME             If Name
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.7.5.7.1.5 Adding an SA Entry

The command to add a SA while running IPSecfwd application is as follows:

```
ipsecfwd_config  -P pid  -A -a "AH SA configuration!" -e "encryption key" -s a.b.c.d -d b.c.d.e -
g
c.d.e.f -G a.d.d.a -i 0 -r dir
```

Example 5. Adding an SA Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config  -P 2536  -A -a "AH SA configuration!" -e "This is 128 bits" -s
192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i 0 -r in
```

SA Entry Added successfully

```
[root@p4080 bin]#
```

For the purpose of using ESN (Extended Sequence Number) feature, user is provided two optional parameters. One is -x, which is intended to tell if user wants to use ESN option or not. Other is -v, which the user can configure with some starting sequence number for the packets. For example :

```
[root@p4080 bin]# ipsecfwd_config  -P 2536  -A -a "AH SA configuration!" -e "This is 128 bits" -s
192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i 0 -r in -x 1 -v 4294967294
```

Table 282. Field Description (Adding an SA Entry)

Parameter	Description	Mandatory	Format/Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d
-g	Left tunnel IP Address	Yes	a.b.c.d
-G	Right Tunnel IP Address	Yes	a.b.c.d
-D	Default Gateway	No	a.b.c.d
-a	Authentication Key for HMAC-SHA1	No (Default key is taken if not supplied)	"string" of length 20 bytes (default = 0x2122_2324_2526_2728_292a_2b2c_2d2e_2f30_3132_3334;
-e	Encryption Key for AES-CBC	No (Default key is taken if not supplied)	"string" of length 16 bytes. (default = 0x0102_0304_0506_0708_090a_0b0c_0d0e_0f10
-i	SPI	Yes	Unsigned int
-r	Direction (encryption/ decryption)	Yes	in (decryption) or out (encryption)
-x	ESN option	No	Unsigned int
-v	Sequence number	No	Unsigned int

9.9.7.5.7.1.6 Help for SA Entry Addition

The command to help add a SA while running IPSecfwd application is as follows:

```
ipsecfwd_config -A --help
```

Example 6. Help for Adding an SA Entry

```
[root@p4080 bin]#]# ipsecfwd_config -A --help
Usage: -A [OPTION...]
  -a, --a=AKEY               Authentication Key
  -d, --d=DESTIP             Destination IP
  -D, --dg=DEFGW             Default Gateway
  -e, --e=EKEY               Encryption Key
  -g, --ss=SRCGW             Source Gateway IP
  -G, --sd=SRCGW             Destination Gateway IP
  -i, --spi=SPI              SPI - 32 bit unsigned int
  -p, --p=PROTO              IPsec Proto type - ESP(0) {Default: 0}
  -r, --dir=DIR              DIR- in/ out
  -s, --s=SRCIP              Source IP
  -t, --t=ETYPE              Encryption Type - AES-CBC(0), 3DES-CBC(1)
                             {Default: 0}
  -v, --seq_num=SEQNUM       Sequence Number
  -x, --is_esn=ESN           Extended Sequence Number support
  -y, --y=ATYPE              Authentication Type - HMAC-SHA1(0) {Default: 0}
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

```
[root@p4080 bin]#
```

9.9.7.5.7.1.7 Deleting an SA Entry

The command to delete an SA while running IPSecfwd is as follows:

```
ipsecfwd_config  -P pid  -D -s 192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i
0
```

Example 7. Deleting an SA Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
root@p4080 bin]# ipsecfwd_config -P 2536 -D -s 192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G
192.168.60.99 -i 0
```

SA Entry Deleted successfully

```
[root@p4080 bin]#
```

Table 283. Field Description (Deleting an SA Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d
-g	Left tunnel IP Address	Yes	a.b.c.d

Table continues on the next page...

Table 283. Field Description (Deleting an SA Entry) (continued)

-G	Right Tunnel IP Address	Yes	a.b.c.d
-i	SPI	Yes	Unsigned int

9.9.7.5.7.1.8 1.4.1.8 Help for Deleting an SA Entry

The command to obtain help for SA entry deletion is as follows:

```
ipsecfwd_config -D --help
```

Example 8. Help for Deleting an SA Entry

```
[root@p4080 bin]# ipsecfwd_config -D --help
Usage: -D [OPTION...]
  -d, --d=DESTIP             Destination IP
  -g, --ss=SRCGW             Source gateway IP
  -G, --sd=DESTGW            Destination gateway IP
  -i, --spi=SPI              SPI
  -s, --s=SRCIP              Source IP
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

```
[root@p4080 bin]#
```

9.9.7.5.7.1.9 Adding a Route Entry

The command to add a route while running IPSecfwd is as follows:

```
ipsecfwd_config -P pid -B -s a.b.c.d -d b.c.d.e -g c.d.e.f
```

Table 284. Field Description (Adding a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d
-g	Gateway IP Address	Yes	a.b.c.d

Example 9. Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config  -P 2536 -B -s  192.168.29.2 -d  192.168.24.2 -g  192.168.24.2
```

Route Entry Added successfully

```
[root@p4080 bin]#
```

9.9.7.5.7.1.10 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

```
ipsecfwd_config -B --help
```

Example 10. Help for Adding a Route Entry

```
[root@p4080 bin]# ipsecfwd_config -B --help
Usage: -B [OPTION]
  -d, --d=DESTIP             Destination IP
  -f, --f=FLOWID             Flow ID - (0 - 1024) {Default: 0}
  -g, --g=GWIP               Gateway IP
  -s, --s=SRCIP              Source IP
  -t, --t=TOS                Type of Service - (0 - 256) {Default: 0}
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.7.5.7.1.11 Deleting a Route Entry

The command to delete a route while running IPSecfwd is as follows:

```
ipsecfwd_config  -P pid  -C -s a.b.c.d -d b.c.d.e
```

Table 285. Field Description (Deleting a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d

Example 11. Deleting a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -C -s  192.168.29.2 -d  192.168.24.2 Route Entry
Deleted
successfully
```

9.9.7.5.7.1.12 Help for Deleting a Route Entry

The command to obtain help for route entry deletion is as follows:

```
ipsecfwd_config -C --help
```

Example 12. Help for Deleting a Route Entry

```
[root@p4080 bin]# ipsecfwd_config -C --help
Usage: -C [OPTION...]
  -d, --d=DESTIP             Destination IP
  -s, --s=SRCIP              Source IP
  -t, --t=TOS                Type of Service - (0 - 256) {Default: 0}
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.7.5.7.1.13 Adding an ARP Entry

The command to add an ARP entry while running IPSecfwd is as follows:

```
ipsecfwd_config  -P pid  -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]
```

Table 286. Field Description (Adding an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d
-m	Mac Address	Yes	aa:bb:cc:dd:ee
-r	Replace existing entry	No	true/ false {Default: false}

Example 14. Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536  -G -s  192.168.24.2 -m 02:00:c0:a8:33:fd -r true
```

ARP Entry Added successfully

9.9.7.5.7.1.14 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

```
ipsecfwd_config -G --help
```

Example 15. Help for Adding an ARP Entry

```
[root@p4080 etc]# ipsecfwd_config -G --help
Usage: -G [OPTION...]
  -m, --m=MACADDR            MAC Address
  -r, --r=Replace            Replace Exiting Entry - true/ false {Default:
                             false}
  -s, --s=IPADDR             IP Address
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.7.5.7.1.15 Deleting an ARP Entry

The command to delete an ARP while running the IPSecfwd is as follows:

```
ipsecfwd_config   -P pid   -H -s a.b.c.d
```

Table 287. Field Description (Deleting an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d

Example 16. Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -H -s  192.168.24.2
Arp Entry Deleted successfully
[root@p4080 bin]#
```

9.9.7.5.7.1.16 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

```
ipsecfwd_config -H --help
```

Example 17. Help for Deleting an ARP Entry

```
[root@p4080 etc]# ipsecfwd_config -H --help
Usage: -H [OPTION...]
-s, --s=IPADDR             IP Address
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.7.5.7.1.17 Adding a high bandwidth tunnel

A high bandwidth tunnel is same as normal tunnel except that it has capability to carry high volume of traffic. The purpose of this feature is to provide high throughput when only single IPSec tunnel is created. For non-high bandwidth tunnel only a single core can process a tunnel's packet at any point of time. High bandwidth tunnel option allows multiple cores to process in parallel the packets of a single tunnel.

To create a high bandwidth tunnel "-b 1" should be appended to command for creating a new security association as shown below:

```
ipsecfwd_config  -P pid  -b 1
```

Table 288. Field Description (Create a high bandwidth tunnel)

Parameter	Description	Mandatory	Format/ Value
-b	High bandwidth tunnel enable	No	1/0 [enable/ disable (default: false)]

When a tunnel is in high bandwidth mode, it should show a higher throughput

9.9.7.6 References

1. USDPAA PPAC User Guide
2. QMan/BMan API Guide

9.9.7.7 Revision History

Document revision history.

Table 289. Revision history

Version	Author	Description
1.0	Nipun Gupta	Initial Draft

9.9.8 NXP Simple Crypto User Manual

9.9.8.1 Revision History Archive

Table 290. Revision History

Version	Author	Description
0.1	Nipun Gupta	Initial draft release
0.2	Mustafa Hamid	Miscellaneous typographical changes
1.0	Nipun Gupta	Updates for SDK V1.0 release
1.1	Nipun Gupta	Updates for SDK V1.1 release

9.9.8.2 Introduction

About this Document

The User Space Data Path Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA "simple crypto" application.
- Execution steps for the "simple crypto" application.

Conventions

This document uses the following conventions:

`Courier` is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

9.9.8.3 USDPAA Simple Crypto Application

9.9.8.3.1 Overview

USDPAA Crypto Application is a multi-threaded Linux User space process. This application exercises the interface to the SEC4.0 engine for raw mode encryption, decryption, and authentication of data. Using the P4080 DPAA, the application generates traffic for the SEC4.0 engine and also processes the output from the SEC4.0 engine. The application also generates performance data for its SEC4.0 interactions.

The applications' threads are created using the standard pthreads library. In Linux User Space, any of the 8 cores can be configured to run a USDPAA Crypto application thread. A dedicated QMan software portal is assigned to a USDPAA application thread and each thread is affined to a core. Each core has its own dedicated Frame Queues to interact with the SEC4.0 block. Traffic is directed to the SEC4.0 via frame descriptor enqueues onto QMan frame queues, which are configured to deliver the frames to SEC4.0 engine for processing.

The SEC4.0 engine processes packets on the basis of commands passed in the form of a shared descriptor. A pointer to the shared descriptor is passed to the SEC4.0 in the ingress (towards the security engine) frame queue descriptors' *contextA* field. The egress FQID is used by SEC4.0 to return output to the application - this is passed in the *contextB* field of the ingress frame queue descriptor. Different SEC4.0 shared descriptors are created for different operation like encryption and decryption.

9.9.8.3.2 Parameters to the application

The crypto application supports various runtime parameters. These configurable parameters are passed to the crypto application from the command line while running the application.

1. Mode: Mode can be PERF or CIPHER
   a. **PERF** Mode: In PERF or Performance mode the application calculates the throughput of SEC4.0 processing of data (including enqueue and dequeue operations to and from SEC4.0 block).
   b. **CIPHER** mode: CIPHER mode allows a test run to demonstrate the SEC4.0 throughput and also compares ciphertext generated by SEC4.0 with ciphertext of a standard test vector.
2. Algorithm: This argument specifies the Algorithm to perform on the data. It is passed to SEC4.0 block using a Shared Descriptor. The algorithm choices are documented in Section Simple Crypto command syntax on page 1754.
3. Number of Iterations: This parameter specifies the number of iterations of data to be looped through the SEC4.0 engine in a test run.
4. Number of buffers: It specifies the total number of buffers to send to SEC4.0 block from each core in one encryption/ decryption or authentication iteration. These buffers are distributed among each core.
5. Size of the buffer: This argument is used only with PERF mode. It specifies the size of the each input buffers sent to SEC4.0.
6. Test set number: This argument is used only with CIPHER mode. It specifies the predefined test set to be used as the data set to send to SEC4.0. There are various test sets (with varying size and data) hardcoded in the application. These are documented in the Section Simple Crypto command syntax on page 1754.
7. Number of cores: This is an optional parameter. By default the application uses all the active cores for the processing of the data sets. By specifying the number of cores, the user can limit the application threads to a specific number of cores.

9.9.8.3.3 Packet Flow



Figure 270. Packet Flow

After initializing the dedicated ingress and egress frame queues for a SEC4.0 operation, the application creates compound frame descriptors (FD's) and fills plain text or pre-defined "cipher" text in input buffers for SEC processing on the basis of the specified mode of operation. For PERF mode, this is plain text; and in the case of CIPHER mode it is pre-defined "cipher" text. The number of FD's equals to the total number of buffers. The FD's, which contain input and output buffer pointers, are first distributed amon the cores and then enqueued to the ingress FQ's. The application thread then polls for output packets from the SEC4.0 egress frame queue until all the packets are received back after being encrypted. As the application uses dedicated Frame Queues between cores and SEC4.0, each core receives the packets which it has enqueued.

In case of CIPHER mode the application checks the received encrypted packets with the already known result.

Lastly, the application checks for the algorithm type. If the algorithm is authentication, then it calculates the throughput in millions of bits per second (Mbps) and exits. If the algorithm type is encryption/decryption, the application sends the same packet (which it received from SEC4.0 after encryption) back to SEC4.0 block for decryption. It does this by changing the pointers of output buffers to point to input buffers and vice-versa in FD's and enqueues these to the SEC4.0's ingress FQ's. It then polls the packet from the SEC4.0 egress frame queue until all the encrypted packets are received back after decryption. The application checks if the packet received after decryption is same as the original plain/cipher text (as a packet after encryption followed by decryption should be the same as the original packet). It then calculates the throughput in Mbps and exits.

9.9.8.3.4 Throughput calculation

The application measures the CPU cycles just before enqueuing the first packet on the FQ and just after receiving the last packet after processing from SEC4.0 for each iteration. The difference between these is the 'delta_cycles' which is accumulated over all the iterations.

Throughput of the application is reported in millions of bits per second (Mbps).

Throughput calculation involves the following parameters.

- 'l' is the number of iterations the application runs for in a test run
- 'n' is the total number of buffers
- 's' is the size of buffer
- 'cpu_freq' is the CPU frequency in MHz

The cycles per frame equals:

cycles_per_frame = (delta_cycles) /(l*n);

Throughput in Mbps equals:

Throughput = (cpu_freq * bits_per_byte *s) / (cycles_per_frame);

= (cpu_freq * 8 *s) / (cycles_per_frame);

9.9.8.3.5 Running Simple Crypto Application on board

1. On the Linux prompt on USDPAA, run the application by typing the following command:

   ```
   simple_crypto -m <mode> -s <size> -n <num_buffer> -o <algo> -l <num_iterations> -t <test_set> [-c
   <num_cores>]
   ```

   Refer to section Simple Crypto command syntax on page 1754 for command syntax.

2. Upon successful completion, the application shows the following message on the USDPAA boot core's console. In case of a failure, a failure message is displayed

   ```
   INFO: SEC4.0 test PASSED
   ```

   Also upon successful completion, the application reports SEC4.0 raw algorithm's throughput on boot core's console.

9.9.8.3.6 Simple Crypto command syntax

The command syntax is as follows:

[root@p4080 root]# simple_crypto --help

Usage: simple_crypto [OPTION...]

-c, --ncpus=CPUS

OPTIONAL PARAMETER

Number of cpus to work for the Application (1-8)(OPTIONAL)

-l, --itrnum=ITERATIONS

Number of iteration to repeat

-m, --mode=TEST MODE

test mode: specify one of the following

1 for perf

2 for cipher

Only the following two combinations are valid. All options are mandatory:

-m 1 -s <buf_size> -n <buf_num_per_core>

-o <algo> -l <itr_num>

or

-m 2 -t <test_set> -n <buf_num_per_core>

-o <algo> -l <itr_num>

-n, --bufnum=TOTAL BUFFERS

Number of buffers per core (1-6400)

-o, --algo=ALGORITHM

Cryptographic operation to be performed by SEC4.0

Specify one of the following:

1 for AES_CBC

2 for TDES_CBC

3 for SNOW_F8

4 for SNOW_F9

5 for KASUMI_F8

6 for KASUMI_F9

7 for CRC

8 for HMAC_SHA1

9 for SNOW_F8_F9(only with PERF mode)

-s, --bufsize=BUFFER SIZE

OPTION IS VALID ONLY IN PERF MODE

Buffer size (64, 128 ...upto 6400)

-t, --testset=TEST SET

OPTION IS VALID ONLY IN CIPHER MODE

provide following test set number:

AES_CBC: 1-4

TDES_CBC: 1-2

SNOW_F8: 1-5

SNOW_F9: 1-5

KASUMI_F8: 1-5

KASUMI_F9: 1-5

CRC: 1-5

HMAC_SHA1: 1-2

SNOW_F8_F9: 1

-?, --help Give this help list

--usage Give a short usage message

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

Note: For For p3041 the ncpus(-c) varies from 1-4 and for p5020 it varies from 1-2.

9.9.8.3.7 Snapshot of Simple Crypto output

The figure below shows a snapshot of simple crypto application output.



Figure 271. Snapshots of Simple Crypto Application

9.9.9 NXP Simple Proto User Manual

9.9.9.1 Revision History Archive

Table 291. Revision History

Version	Author	Description
7	Tudor Ambarus	Added RSA, TLS, IPsec protocol processing
6	Alex Porosanu	Added MBMS protocol processing
0.1	Carmen Iorga	Initial draft release

9.9.9.2 Introduction

About this Document

The USDPAA Simple Proto application demonstrates the usage of security coprocessor's capabilities in handling traffic in security protocols context

This document provides the following:

- A summary of the USDPAA "simple proto" application.
- Execution steps for running "simple proto" application.

Conventions

This document uses the following conventions:

`Courier` is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

9.9.9.3 USDPAA Simple Proto Application

9.9.9.4 Overview

USDPAA Simple Proto application demonstrates usage of SEC engine protocols' integrity and confidentiality algorithms. Using the DPAA framework, the application generates traffic and enqueues it to SEC engine, process the ouput and generate performance data for its SEC interactions.

This is a multi-threaded Linux User Space application. Threads are created using pthreads library, each application thread has an assigned QMan software portal and is affined to a core. Each core has its own dedicated Frame Queues to interact with SEC. Traffic is injected to SEC via frame descriptors enqueued onto QMan frame queues.

The SEC engine processes packets on the basis of commands passed in the form of a shared descriptor. A pointer to the shared descriptor is passed to the SEC in the ingress frame queue descriptors'*contextA* field. The egress FQID is used by SEC to return output to the application - this is passed in the *contextB* field of the ingress frame queue descriptor. Different SEC shared descriptors are created for different protocols' operation.

9.9.9.5 Parameters to the application

The simple_proto application supports various runtime parameters. These configurable parameters are passed to the application from the command line when running the application.

1. Mode: Mode can be PERF or CIPHER
   a. **PERF** Mode: In PERF or Performance mode the application calculates the throughput of SEC processing of data (including enqueue and dequeue operations to and from SEC block). Also, output frames from decapsulation are compared against input frames to encapsulation.

   b. **CIPHER** mode: CIPHER mode allows a test run to demonstrate the SEC throughput and also compares ciphertext generated by SEC with ciphertext of a standard test vector.

2. Protocol: This argument specifies the protocol to be tested. It is passed to SEC block using a Shared Descriptor. The protocol choices are documented in Section Simple Proto command syntax on page 1760. Each protocol has its own set of mandatory and/or optional parameters, which are explained in MACSec protocol options on page 1762, WiMAX protocol options on page 1762, PDCP protocol options on page 1762 and MBMS Protocol Options on page 1765
3. Number of Iterations: This parameter specifies the number of iterations of data to be looped through the SEC engine in a test run.
4. Number of buffers: This specifies the total number of buffers to send to SEC block from each core in one encryption/ decryption or authentication iteration. These buffers are distributed among each core.
5. Size of the buffer: This argument is used only with PERF mode. It specifies the size of the each input buffers sent to SEC.
6. Test set number: This argument is used only with CIPHER mode. It specifies the predefined test set to be used as the data set to send to SEC. There are various test sets (with varying size and data) hardcoded in the application. These are documented in the Section Simple Proto command syntax on page 1760.
7. Number of cores: This is an optional parameter. By default the application uses all the active cores for the processing of the data sets. By specifying the number of cores, the user can limit the application threads to a specific number of cores.
8. SEC Era: This is an optional parameter. This specifies the SEC era hardware block revision for which SEC descriptors will be generated. **By default, the application runs with default era set to value 2**. By specifying the SEC era, the user can set the right value for the targeted platform to test. For example, SEC era on the following platforms is:
   - 2 for P4080 TO2
   - 3 for P3041, P5020
   - 4 for P4080 TO3
   - 5 for P5040, B4860
   - 6 for T4240, T2080 and T1040
   - 7 for LS1021

9.9.9.6 Packet Flow



Figure 272. Packet Flow

After initializing the dedicated ingress and egress frame queues for a SEC operation, the application creates compound frame descriptors (FD's) and prepare input buffers for SEC processing based on specified mode of operation; for PERF mode, buffers are filled in with incrementing pattern plain text (for WiMAX encapsulation, SEC expects GMH aware input frames: GMH Header Type bit set to zero, the CRC indicatr bit set to one if CRC is included in the PDU, and the GMH LEN field updated accordingly to the input plain data length), and in the case of CIPHER mode, with golden pattern data. The number of FD's equals to the total number of buffers and are first distributed among the cores and then enqueued to the ingress FQ's. The application thread then polls for output packets from the SEC egress frame queue until all the packets are received back after being encapsulated. As the application uses dedicated Frame Queues between cores and SEC, each core receives the

packets which it has enqueued. In CIPHER test mode, the application checks the received encapsulated packets against a golden pattern.

The next step is the application to verify the type of test. If the test set is unidirectional , then it calculates the throughput in millions of bits per second (Mbps) and exits. If the test verifies both encapsulation/decapsulation, the application sends the packet which it received from SEC after encapsulation back to SEC block for decapsulation. It does this by interchanging the pointers to output and input buffers in FD's and enqueues these to the SEC's ingress FQ's. It then polls the packet from the SEC egress frame queue until all the encapsulated packets are received back after decapsulation. The application checks if the packet received after decapsulation is the same as the original plain/cipher text (as a packet after encapsulation followed by decapsulation should be the same as the original packet). It then calculates the throughput in Mbps and exits.

9.9.9.7 Throughput calculation

The application measures the CPU cycles just before enqueuing the first packet on the FQ and just after receiving the last packet after processing from SEC for each iteration. The difference between these is the 'delta_cycles' which is accumulated over all the iterations.

Throughput of the application is reported in millions of bits per second (Mbps).

Throughput calculation involves the following parameters.

- 'l' is the number of iterations the application runs for in a test run
- 'n' is the total number of buffers
- 's' is the size of buffer
- 'cpu_freq' is the CPU frequency in MHz

The cycles per frame equals:

cycles_per_frame = (delta_cycles) /(l*n);

Throughput in Mbps equals:

Throughput = (cpu_freq * bits_per_byte *s) / (cycles_per_frame);

= (cpu_freq * 8 *s) / (cycles_per_frame);

9.9.9.8 Running Simple Proto Application on board

1. On the Linux prompt on USDPAA, run the application by typing the following command:

   ```
   simple_proto -m <mode> -s <size> -n <num_buffer> -p <protocol> -l <num_iterations> -t <test_set> [-c <num_cores> -e <sec_era>]
   ```

   Refer to section Simple Proto command syntax on page 1760 for command syntax.

2. Upon successful completion, the application shows the following message on the USDPAA boot core's console:

   ```
   INFO: SEC4.0 test PASSED
   ```

   Also upon successful completion, the application reports SEC4.0 raw algorithm's throughput on boot core's console.

   In case of failure, a failure message is displayed on console.

9.9.9.9 Simple Proto command syntax

The command syntax is as follows:

```
root@p4080ds:~# simple_proto --help
Usage: simple_proto [OPTION...]

  -c, --ncpus=CPUS           OPTIONAL PARAMETER
```

```
                              Number of cpus to work for the application(1-8)

  -e, --sec_era=ERA           OPTIONAL PARAMETER

                              SEC Era version on the targeted platform(2-5)

  -l, --itrnum=ITERATIONS     Number of iterations to repeat

  -m, --mode=TEST MODE        Test mode:
                                 1 for perf
                                 2 for cipher

                              Following two combinations are valid only and all
                              options are mandatory:
                              -m 1 -s <buf_size> -n <buf_num_per_core> -p
                              <proto> -l <itr_num>
                              -m 2 -t <test_set> -n <buf_num_per_core> -p
                              <proto> -l <itr_num>

  -n, --bufnum=TOTAL BUFFERS Total number of buffers (1-6400). Both of Buffer
                              size and buffer number cannot be greater than 3200
                              at the same time.

  -p, --proto=PROTOCOL        Cryptographic operation to perform by SEC:
                               1 for MACsec
                               2 for WiMAX
                               3 for PDCP
                               4 for SRTP
                               5 for WiFi
                               6 for RSA
                               7 for TLS
                               8 for IPsec
                               9 for MBMS


  -s, --bufsize=BUFSIZE       OPTION IS VALID ONLY IN PERF MODE

                              Buffer size (64, 128 ... up to 6400). Note: Both
                              of Buffer size and buffer number cannot be greater
                              than 3200 at the same time.
                              The WiMAX frame size, including the FCS if
                              present, must be shorter than 2048 bytes.

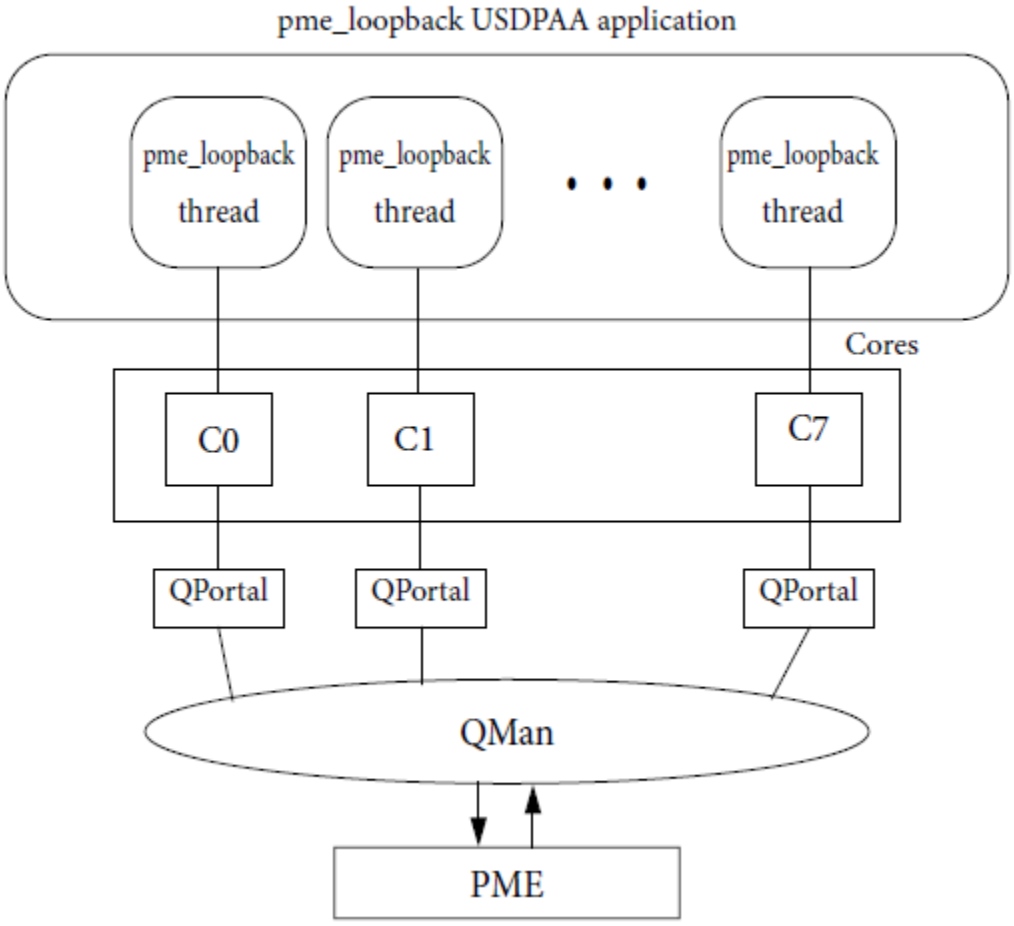
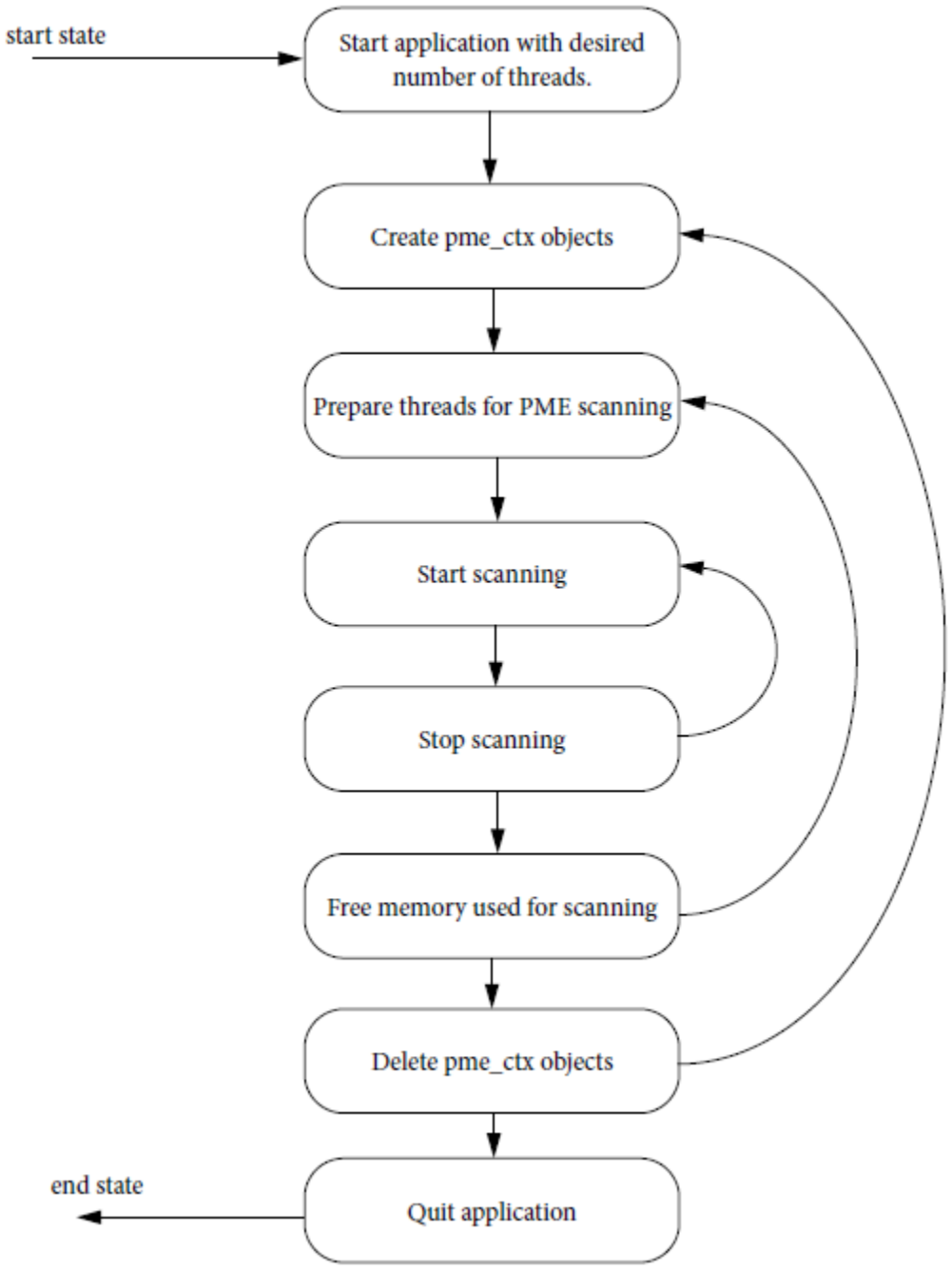
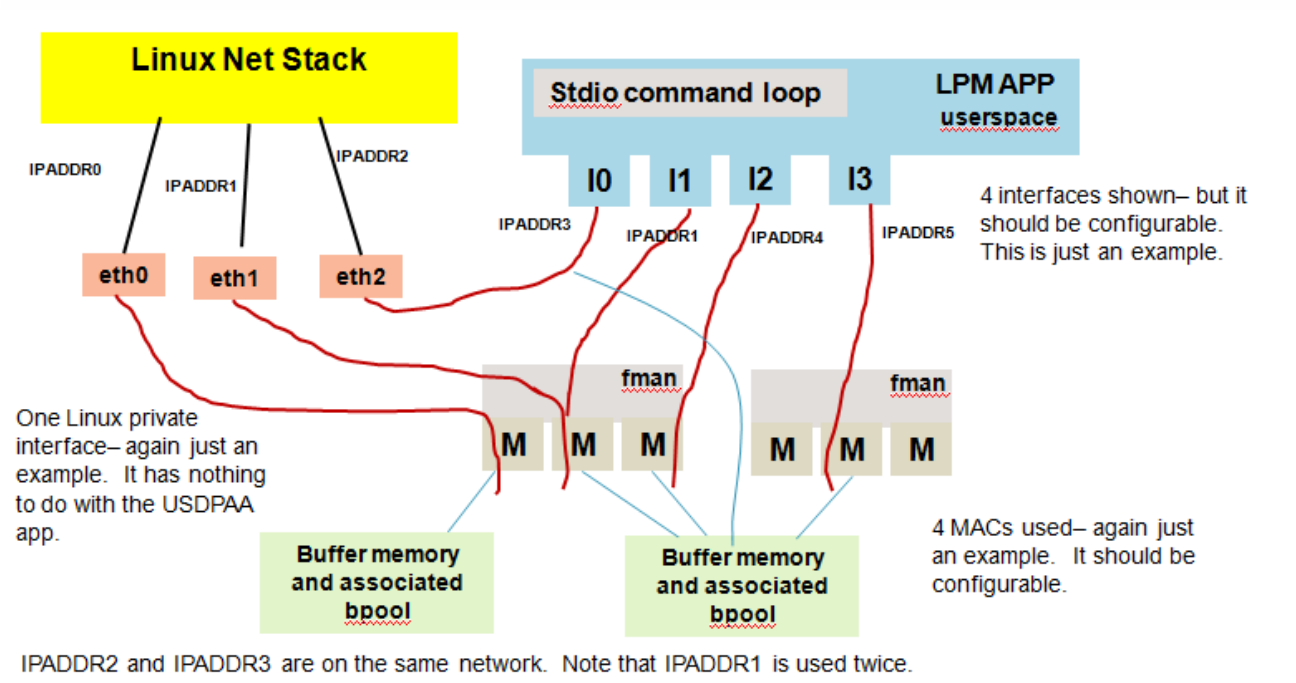
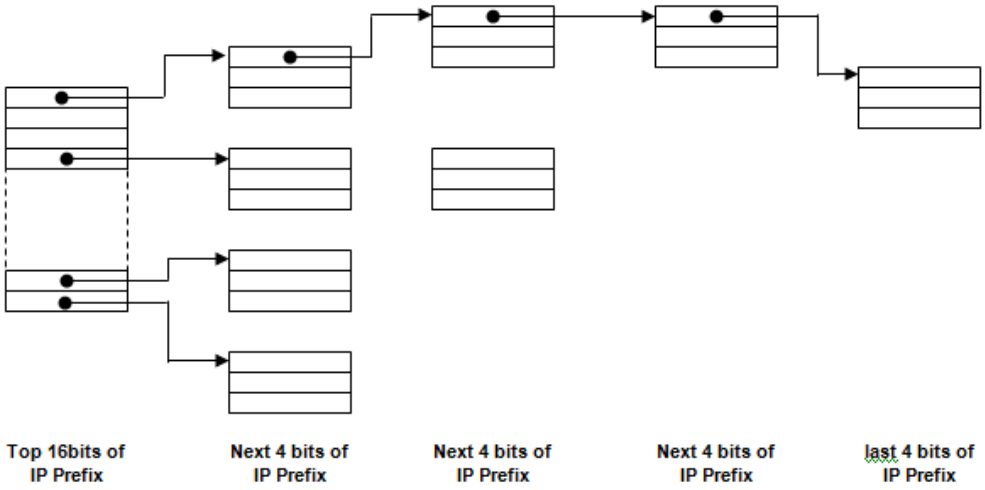
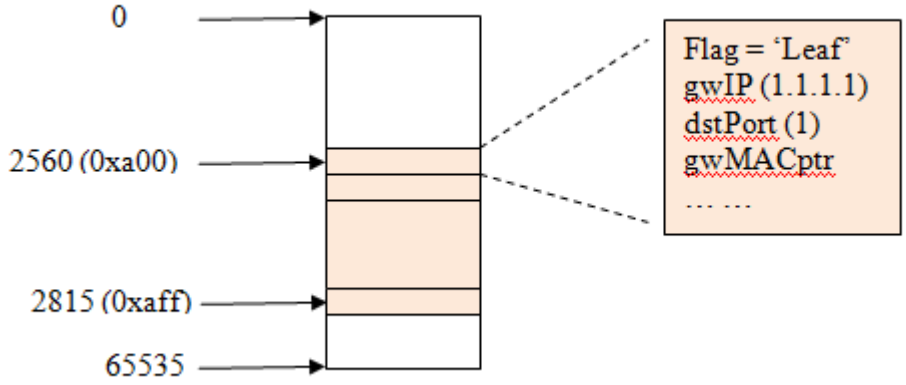
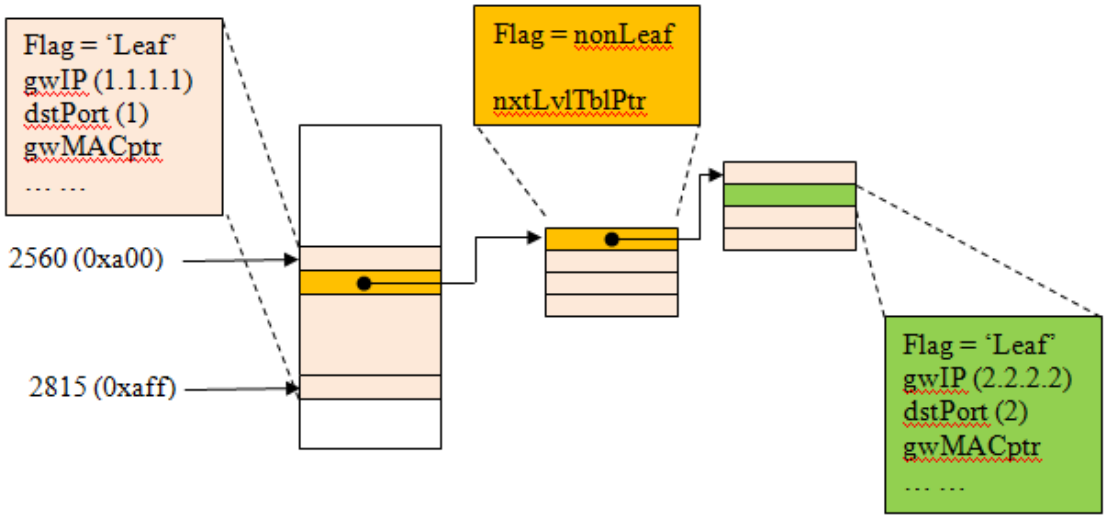
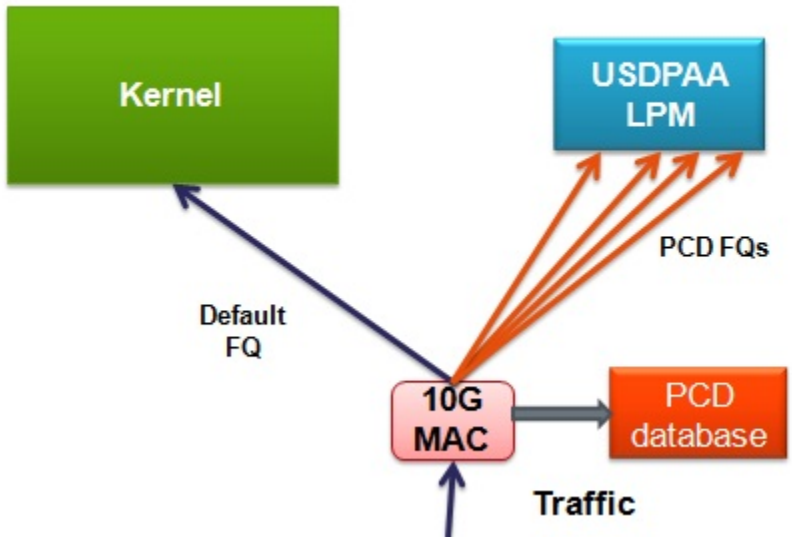
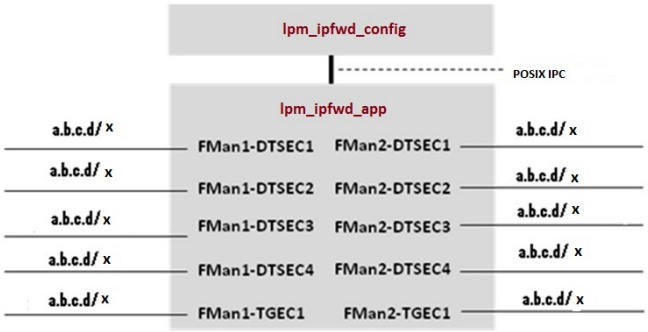
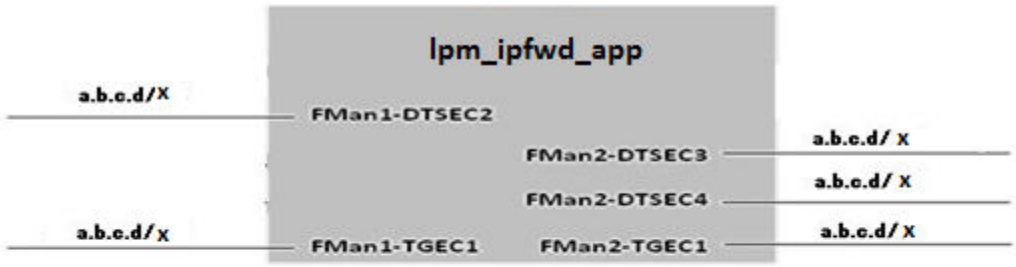
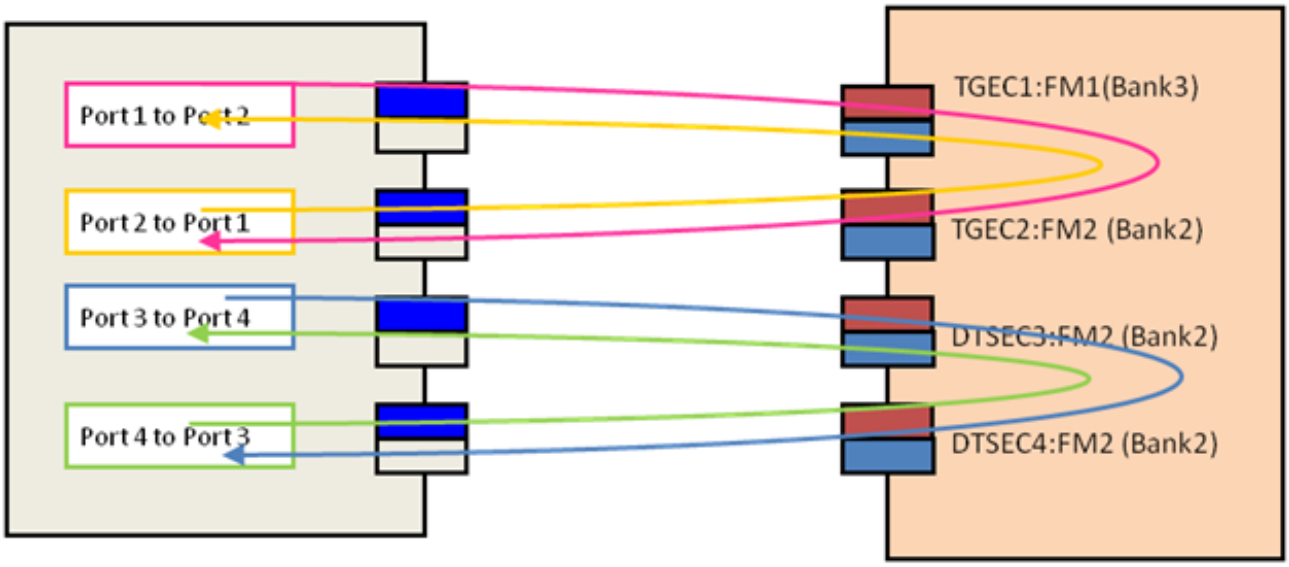
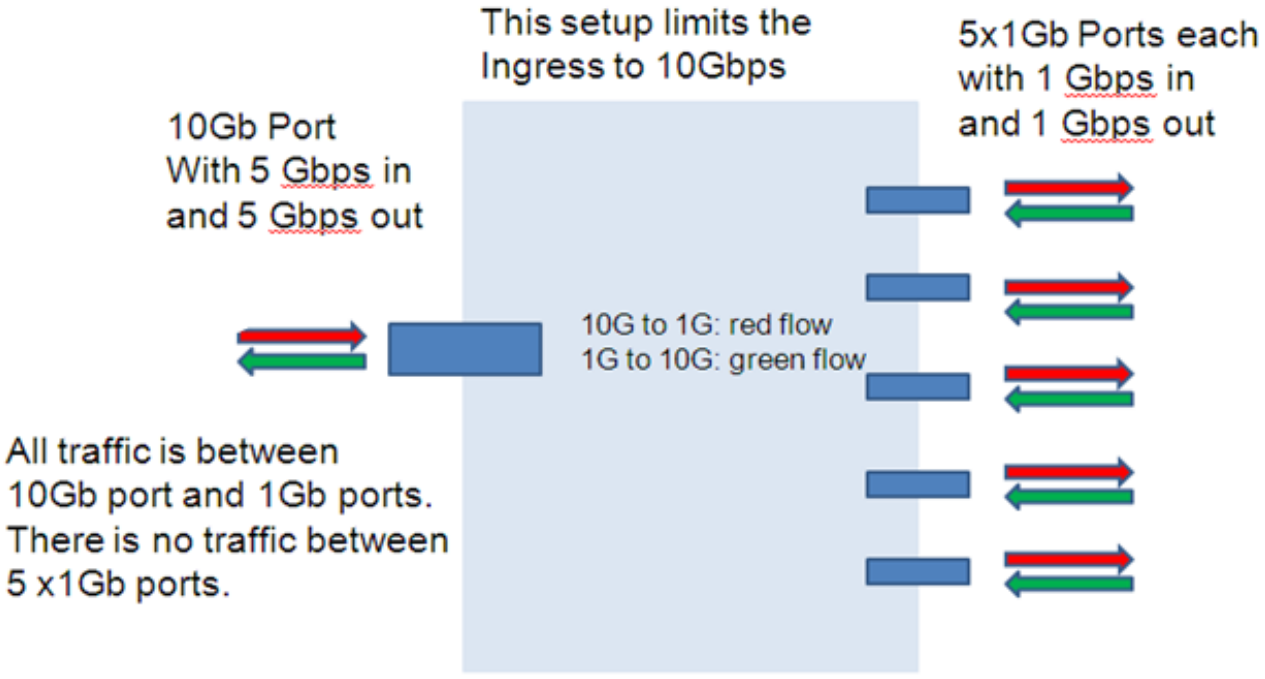
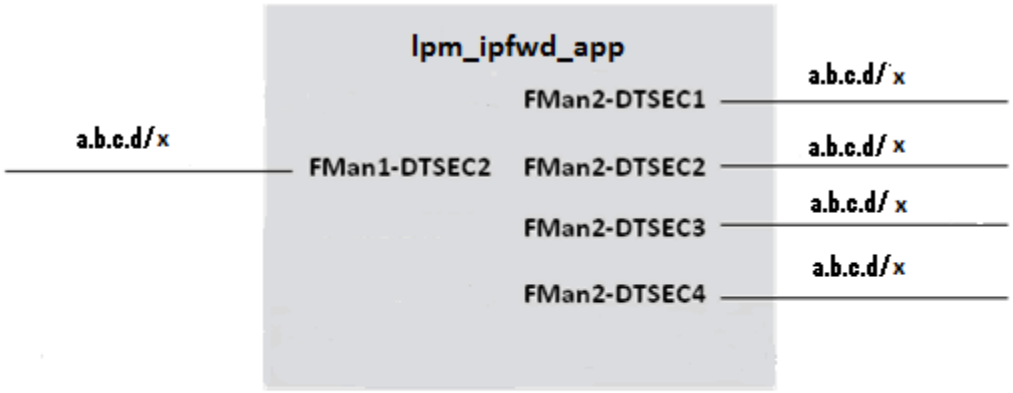
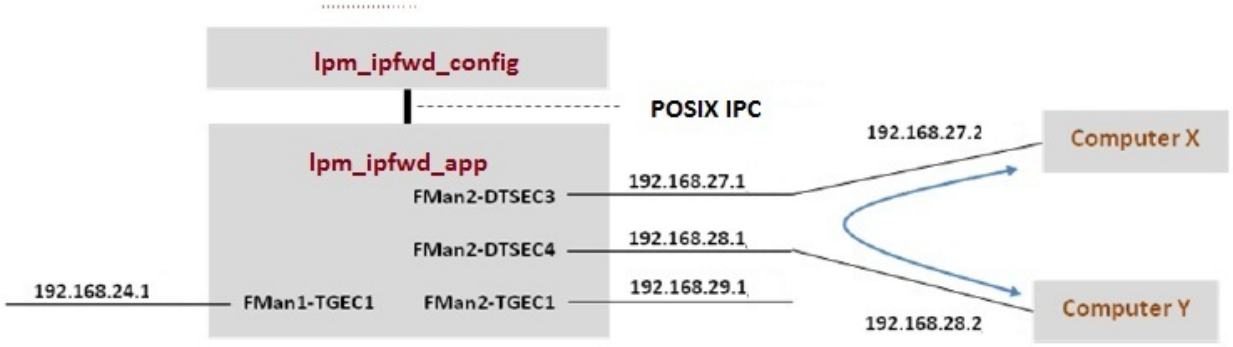
  -t, --testset=TEST SET      OPTION IS VALID ONLY IN CIPHER MODE

  -?, --help                  Give this help list
      --usage                 Give a short usage message

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

NOTE

Depending on the hardware platform, the ncpus(-c) varies as follows: for p3041 between 1-4, for p5020 between 1-2, for B4860 between 1-8 and for T4240 between 1-24.

NOTE

The valid test set numbers are the following, per each protocol:

1. MACSec - 1 .. 5
2. WiMAX - 1 .. 4
3. PDCP - 1
4. SRTP - 1
5. WiFi - 1 .. 2
6. RSA - 1 .. 2
7. TLS - 1
8. IPsec - 1
9. MBMS:
    a. MBMS PDU Type 0 - 1 .. 2
    b. MBMS PDU Type 1 - 1 .. 3
    c. MBMS PDU Type 3 - 1 .. 3

9.9.9.10 MACSec protocol options

For MACSec processing, the simple_proto application understands the following parameters (apart from the mandatory & optional ones specified in Simple Proto command syntax on page 1760):

- -o, --algo : this is an optional parameter that, when set, allows the user to choose the cipher type. The cipher type can be GCM or GMAC; by default, the MACSec protocol will use GCM processing.

9.9.9.11 WiMAX protocol options

WiMAX processing is available only if SEC Era if equal or grater than 4.

For WiMAX processing, the simple_proto application understands the following parameters (apart from the mandatory & optional ones specified in Simple Proto command syntax on page 1760):

- -a, --ofdma : this is an optional parameter that, when set, it enables OFDMA processing for WiMAX. By default, the WiMAX protocol offload does OFDM processing;
- -f, --fcs : this is an optional parameter that, when set, instructs the WiMAX protocol offload to compute the FCS over the input frame, making it longer by 4 bytes;
- -w, --ar_len=ARWIN : another optional parameter that enables the anti-replay mechanism in WiMAX protocol processing. This parameter also sets the anti-replay window length, which cannot exceed 64 frames.

9.9.9.12 PDCP protocol options

Packet Data Convergence Protocol (abbrev. PDCP) is one of the layers of the Radio Traffic Stack in UMTS and performs IP header compression and decompression, transfer of user data and maintenance of sequence numbers for Radio Bearers which are configured for lossless serving radio network subsystem (SRNS) relocation.

In simple_proto application, the following protocol sub-sets are tested & supported:

1. PDCP Control Plane;
2. PDCP User Plane;
3. PDCP Short MAC.

The PDCP ciphering & integrity algorithm combinations supported by simple_proto application are the following:

1. PDCP Control Plane:
   a. NULL encryption & NULL integrity (EEA0/EIA0)
   b. NULL encryption & SNOW f9 integrity (EEA0/EIA1)
   c. NULL encryption & AES CMAC integrity (EEA0/EIA2)
   d. NULL encryption & ZUC integrity (EEA0/EIA3)*
   e. SNOW f8 encryption & NULL integrity (EEA1/EIA0)
   f. SNOW f8 encryption & SNOW f9 integrity (EEA1/EIA1)
   g. SNOW f8 encryption & AES CMAC integrity (EEA1/EIA2)
   h. SNOW f8 encryption & ZUC integrity (EEA1/EIA3)*
   i. AES CTR encryption & NULL integrity (EEA2/EIA0)
   j. AES CTR encryption & SNOW f9 integrity (EEA2/EIA1)
   k. AES CTR encryption & AES CMAC integrity (EEA2/EIA2)
   l. AES CTR encryption & ZUC integrity (EEA2/EIA3)*
   m. ZUC encryption & NULL integrity (EEA3/EIA0)
   n. ZUC encryption & SNOW f9 integrity (EEA3/EIA1)
   o. ZUC encryption & AES CMAC integrity (EEA3/EIA2)
   p. ZUC encryption & ZUC integrity (EEA3/EIA3)*
2. PDCP User Plane:
   a. NULL encryption (EEA0)
   b. SNOW f8 encryption (EEA1)
   c. AES CTR encryption (EEA2)
   d. ZUC encryption (EEA3)*
3. PDCP Short MAC:
   a. NULL integrity (EIA0)
   b. SNOW f9 integrity (EIA1)
   c. AES CMAC integrity (EIA2)
   d. ZUC integrity (EIA3)*

NOTE

Starred combinations above are available only for platforms with SEC ERA greater than 4 (for instance P5040/B4860R1&R2/T4240/etc.). Attempting to run these combinations on platforms without the proper SEC ERA version will result in a SEC error.

NOTE

For the following combinations used for decapsulating PDCP PDUs, the SEC will return an error code similar to 0x3000XX0a if the last 4 bytes of the decapsulated frame (the ICV) are not set to the value of {0x00, 0x00, 0x00, 0x00}

1. EEA1/EIA0
2. EEA2/EIA0
3. EEA3/EIA0

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in Simple Proto command syntax on page 1760):

Parameter	Explanation	Valid for Control Plane?	Valid for User Plane?	Valid for Short MAC?
-d, --direction	Selects the downlink direction for the inserted PDU; by default, the direction of thePDU is assumed to be uplink.	Yes, optional	Yes, optional	No
-i, --integrity	Selects the integrity algorithm to be used for processing the PDU	Yes, mandatory	No	Yes, mandatory
-r, --cipher	Selects the ciphering algorithm to be used for processing the PDU	Yes, mandatory	Yes, mandatory	No
-v, --hfn_ov	Enables the HFN value used for processing the PDU to be specified by the user.	Yes, optional	Yes, optional	No
-x, --snlen	Select the User Plane PDUs sequence number length. Three values are permitted: 0 = 12 bit Sequence Number PDU 1 = 7 bit Sequence Number PDU 2 = 15 bit Sequence Number PDU	No	Yes, optional	No
-y, --type	Selects the way the input PDU is to be treated: 0 = Control Plane 1 = User Plane 2 = Short MAC	Yes, mandatory	Yes, mandatory	Yes, mandatory

9.9.9.13 RSA operations options

For RSA processing, the simple_proto application understands the following parameter (apart from the mandatory & optional ones specified in Simple Proto command syntax on page 1760):

- -b, --form : this is an optional parameter that, when set, allows the user to choose one of the three RSA Decrypt Private Key formats:

  1 = Form 1 (default)

  2 = Form 2

  3 = Form 3

9.9.9.14 TLS protocol options

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols designed to provide communication security over the Internet.

simple_proto application tests and supports the TLS10 security protocol.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in Simple Proto command syntax on page 1760):

Parameter	Explanation	Optional / Mandatory
-j, --cipher	Selects the ciphering algorithm to be used for processing the PDU: 0 = AES-CBC	mandatory
-k, --integrity	Selects the integrity algorithm to be used for processing the PDU: 0 = HMAC-SHA1	mandatory
-g, --version	Select the SSL protocol version to run: 0 = SSL30 (not supported) 1 = TLS10 2 = TLS11 (not supported) 3 = TLS12 (not supported) 4 = DTLS10 (not supported)	mandatory

9.9.9.15 IPsec protocol options

Internet Protocol Security (IPsec) is a protocol suite for securing Internet Protocol (IP) communications by authenticating and encrypting each IP packet of a communication session.

simple_proto application tests and supports 3des & hmac-md5-96 IPsec ESP Tunnel mode.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in Simple Proto command syntax on page 1760):

Parameter	Explanation	Optional / Mandatory
-h, --cipher	Selects the ciphering algorithm to be used for processing the PDU: 0 = 3DES	mandatory
-q, --integrity	Selects the integrity algorithm to be used for processing the PDU: 0 = HMAC_MD5_96	mandatory

9.9.9.16 MBMS Protocol Options

MBMS SYNC protocol is defined in 3GPP TS 25.446 - MBMS synchronisation protocol (SYNC)

The MBMS Synchronisation protocol (SYNC) is located in the User plane of the Radio Network layer over the Iu interface: the Iu UP protocol layer.

The SYNC protocol for UTRAN is used to convey userdata associated to MBMS Radio Access Bearers.

The simple_proto application supports checking for the CRC validity of the following MBMS PDU Types:

1. MBMS PDU Type 0
2. MBMS PDU Type 1
3. MBMS PDU Type 3

The following table summarizes the behavior of the MBMS SYNC processing:

MBMS SYNC PDU	Default action	Header CRC fail action	Payload CRC fail action(s)
Type 0	Copy PDU	Drop PDU	N/A
Type 1	Copy PDU	Drop PDU	1. Update Payload CRC in PDU's header 2. Copy Header only
Type 3	Copy PDU	Drop PDU	1. Update Payload CRC in PDU's header 2. Copy Header only

The following table lists SEC return codes used for signaling the different actions the SEC takes in order to process the MBMS SYNC PDUs:

Processing result	SEC status/command
PDU Header & Payload CRC OK	0x0000_0000
Wrong PDU Header CRC	0x3000_XXAA
Wrong PDU Payload CRC	0x3000_XXAB

NOTE

The "XX" in the above rows is an internal offset used by SEC and can be safely masked out when checking the SEC status.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in Simple Proto command syntax on page 1760

Parameter	Explanation	Valid values
-z, --type	Selects the MBMS PDU Type to be processed	• 0 - MBMS PDU Type 0 • 1 - MBMS PDU Type 1 • 3 - MBMS PDU Type 2

9.9.10 SEC Descriptor construction library (DCL)

9.9.10.1 SEC Descriptor Construction Library (DCL)

A description of the SEC descriptor construction library (DCL) as a library within USDPAA.

The following is a description of the SEC descriptor construction library (DCL) as a library within USDPAA:

- DCL Description on page 1767, provides a brief overview of the DCL
- DCL Packaging on page 1767, describes how the DCL is packaged
- DCL Files on page 1767, lists the supported files
- DCL Functional Description on page 1767, describes the three layers of the DCL:
  - Lower layers-Command Generator on page 1767, and Descriptor Disassembler on page 1767
  - Upper layer Upper-Tier DCL Functions - Descriptor Constructors on page 1781

9.9.10.2 DCL Description

The Descriptor Construction Library (DCL) provides a collection of simple functions capable of building SEC4.x descriptors for a wide range of purposes, either as a standalone library or integrated within a driver subsystem. Applications may use all, part, or none of the DCL's functions, or they may use DCL as a simple reference for their own construction functions.

The DCL package will evolve over time. Additional protocol support and descriptor utilities will be added in future software (SW) releases, as new applications for SEC 4.x are developed.

9.9.10.3 DCL Packaging

DCL is packaged with and used only by USDPAA sample applications.

Types in DCL are defined using POSIX conventions for the sake of external portability.

9.9.10.4 DCL Files

The following files are supported

- `cmdgen.c`-Descriptor command generator.
- `disasm.c`-Descriptor disassembler.
- `jobdesc.c`-Job descriptor constructors.
- `protoshared.c`-Shared/protocol descriptor constructors.
- `dcl.h`-Definitions for all published DCL functions.

9.9.10.5 DCL Functional Description

DCL consists of two layers, decomposed into three subsystems:

- A lower tier, comprising the following:
  - Command generator (described in Command Generator on page 1767), colloquially referred to as "cmdgen".
  - Descriptor disassembler (described in Descriptor Disassembler on page 1767)
- An upper tier, composed of descriptor constructors (described in Upper-Tier DCL Descriptor Constructors on page 1768). This is dependent on the command generator in the lower tier.

9.9.10.6 Command Generator

The Command Generator is the lowest level of DCL functionality. Each function within it is capable of generating a single command/instruction in a SEC4.x descriptor and increments a "next in" pointer to the next descriptor word following the generated command.

Applications may use the command generator independently of any other DCL functionality.

In general, creation of any application needing to generate a descriptor on an individual command basis like this starts by building the first commands (or PDB data) past the header, until the body of the descriptor is complete. At this point, the full size of the descriptor in known; the application can then fill in the header using this size.

9.9.10.7 Descriptor Disassembler

The Descriptor Disassembler is meant to be a simple debug tool that can display the content of a constructed descriptor for the user to see in a simple "disassembled" representation. It is intended for developers to use as a visualization aid during development, or as a debug tool, in which descriptor content can be displayed on-the-fly in a human-decipherable form. It does not perform consistency checking, or otherwise identify problem areas in poorly formed descriptors.

The disassembler is a simple C function, and can be packaged in the library with the balance of DCL functions so that it may be linked into a higher-level application.

An example of a disassembled IPSec CBC decapsulation shared descriptor:

```
shrdesc: stidx=8 len=20 share-always
     (pdb): [00] 0x00340001 0x00000000 0x00000000 0x00000000
     (pdb): [04] 0x00000000 0x00000000 0x00000000
       key: len=20 class2->keyreg inline
            [00] 0x000e0f00 0x0d0f0a00 0x0d0f0a00 0x0d0f0a00
            [04] 0x0d0f0a00
       key: len=16 class1->keyreg inline
            [00] 0x00e0f0a0 0x00d0f0a0 0x00e0f0a0 0x00d0f0a0
 operation: type=decap-pcl ipsec aes-cbc hmac-sha1-96
```

An example of a disassembled IPSec CBC encapsulation shared descriptor:

```
shrdesc: stidx=23 len=35 share-always
     (pdb): [00] 0x0000000d 0x00000000 0x00000000 0x00000000
     (pdb): [04] 0x00000000 0x00000000 0x00000000 0x00000000
     (pdb): [08] 0x00000034 0x34001045 0x00402512 0xd2860640
     (pdb): [12] 0x7746430a 0xc046430a 0x160022d0 0x5d891888
     (pdb): [16] 0x9cee1912 0xbc211080 0x00000898 0x0a080101
     (pdb): [20] 0x22759aa6 0xdb143f08
       key: len=20 class2->keyreg inline
            [00] 0x000e0f00 0x0d0f0a00 0x0d0f0a00 0x0d0f0a00
            [04] 0x0d0f0a00
       key: len=16 class1->keyreg inline
            [00] 0x00e0f0a0 0x00d0f0a0 0x00e0f0a0 0x00d0f0a0
 operation: type=encap-pcl ipsec aes-cbc hmac-sha1-96
```

9.9.10.8 Upper-Tier DCL Descriptor Constructors

A higher level of functionality is provided through complex descriptor constructors. These constructors are single-purpose functions capable of generating complete descriptors from user specifications. These constructors fit into two categories, one for job descriptors and another for shared descriptors generally targeted to protocol processing.

These constructors are by no means the "definitive" reference to all possible descriptor permutations, nor are they meant to work with any specific application. Instead, they are meant to be general-purpose examples of descriptor construction. It is expected that, over time, this library will grow to accommodate a wide range of examples.

All constructor functions are dependent on the underlying command generator.

9.9.10.9 API reference

9.9.10.9.1 API reference command generator

9.9.10.9.1.1 cmd_insert_shared_hdr()

`cmd_insert_shared_hdr()`: Insert a shared descriptor header into a descriptor

```
  u_int32_t *cmd_insert_shared_hdr(u_int32_t *descwd,
                                   u_int8_t startidx,
                                   u_int8_t desclen,
                                   enum ctxsave ctxsave,
                                   enum shrst share);
```

Inputs:

- descwd-pointer to target descriptor word to hold this command. Note that this should always be the first word of a descriptor.
- startidx-index to continuation of descriptor data, normally the first descriptor word past a PDB. This tells DECO what to skip over.
- desclen-length of descriptor in words, including header.
- ctxsave-Saved or erases context when a descriptor is self-shared
  - CTX_SAVE = context saved between iterations
  - CTX_ERASE = context is erased
- share-Share state of this descriptor:
  - SHR_NEVER = Never share. Fetching is repeated for each processing pass.
  - SHR_WAIT = Share once processing starts.
  - SHR_SERIAL = Share once completed.
  - SHR_ALWAYS = Always share (except keys)

Returns:

Pointer to next incremental descriptor word past the header just constructed. If an error occurred, returns 0.

NOTE

Headers should normally be constructed as the final operation in the descriptor construction, because the start index and overall descriptor length will likely not be known until construction is complete. For this reason, there is little use to the "incremental pointer" convention. The exception is probably in the construction of simple descriptors where the size is easily known early in the construction process.

9.9.10.9.1.2 cmd_insert_hdr()

cmd_insert_hdr(): Insert a standard descriptor header into a descriptor

```
u_int32_t *cmd_insert_hdr(u_int32_t *descwd,
                          u_int8_t startidx,
                          u_int8_t desclen,
                          enum shrst share,
                          enum shrnext sharenext,
                          enum execorder reverse,
                          enum mktrust mktrusted);
```

Inputs:

- descwd-pointer to target descriptor word to hold this command. Note that this should always be the first word of a descriptor.
- startidx-index to continuation of descriptor data, or if sharenext = SHRNXT_SHARED, then specifies the size of the associated shared descriptor referenced in the following instruction.
- desclen-length of descriptor in words, including header.
- share-Share state for this descriptor:
  - SHR_NEVER-Never share. Fetching is repeated for each processing pass.
  - SHR_WAIT-Share once processing starts.
  - SHR_SERIAL-Share once completed.

- SHR_ALWAYS-Always share (except keys)
- SHR_DEFER-Use the referenced sharedesc to determine sharing intent

- sharenext-Control state of shared descriptor processing
  - SHRNXT_SHARED-This is a job descriptor consisting of a header and a pointer to a shared descriptor only.
  - SHRNXT_LENGTH-This is a detailed job descriptor, thus desclen refers to the full length of this descriptor.
- reverse-Reverse execution order between this job descriptor, and an associated shared descriptor:
  - ORDER_REVERSE-execute this descriptor before the shared descriptor referenced.
  - ORDER_FORWARD-execute the shared descriptor, then this descriptor.
- mktrusted-DESC_SIGN-sign this descriptor prior to execution
  - DESC_STD -leave descriptor non-trusted

9.9.10.9.1.3 cmd_insert_key()

cmd_insert_key(): Insert a key command into a descriptor

```
u_int32_t *cmd_insert_key(u_int32_t           *descwd,
              u_int8_t                        *key,
              u_int32_t                       keylen,
              enum ref_type                   sgref,
              enum key_dest                   dest,
              enum key_cover                  cover,
              enum item_inline                imm,
              enum item_purpose purpose);
```

Inputs:

- descwd-pointer to target descriptor word to hold this command
- key-pointer to key data as an array of bytes.
- keylen-pointer to key size, expressed in bits.
- sgref-pointer is actual data, or a scatter-gather list representing the key:
  - PTR_DIRECT-points to data
  - PTR_SGLIST-points to SEC4.x-specific scatter gather table. Cannot use if imm = ITEM_INLINE.
- dest-target destination in SEC4.x to receive the key. This may be:
  - KEYDST_KEYREG-Key register in the CHA selected by an OPERATION command.
  - KEYDST_PK_E-The 'e' register in the public key block
  - KEYDST_MD_SPLIT-Message digest IPAD/OPAD direct load.
- cover-Key was encrypted, and must be decrypted during the load. If trusted descriptor, use TDEK, else use JDEK to decrypt.
  - KEY_CLEAR-key is cleartext, no decryption needed
  - KEY_COVERED-key is ciphertext, decrypt.
- imm-Key can either be referenced, or loaded into the descriptor immediately following the command for improved performance.
  - ITEM_REFERENCE-a pointer follows the command.

- ITEM_INLINE-key data follows the command, padded out to a descriptor word boundary.

- purpose-Sends the key to the class 1 or 2 CHA as selected by an OPERATION command. If dest is KEYDST_PK_E, this must be ITEM_CLASS1.

Returns:

If successful, returns a pointer to the target word incremented past the newly-inserted command (including item pointer or inlined data). Effectively, this becomes a pointer to the next word to receive a new command in this descriptor. If error, returns 0

9.9.10.9.1.4 cmd_insert_seq_key()

cmd_insert_key(): Insert a key command into a descriptor using a sequence

```
u_int32_t *cmd_insert_key(u_int32_t           *descwd,
                  u_int32_t                   keylen,
                  enum ref_type               sgref,
                  enum key_dest               dest,
                  enum key_cover              cover,
                  enum item_inline            imm,
                  enum item_purpose purpose);
```

Inputs:

- descwd-pointer to target descriptor word to hold this command

- keylen-pointer to key size, expressed in bits.

- sgref-pointer is actual data, or a scatter-gather list representing the key:

  - PTR_DIRECT-points to data

  - PTR_SGLIST-points to SEC4.x-specific scatter gather table. Cannot use if imm = ITEM_INLINE.

- dest-target destination in SEC4.x to receive the key. This may be:

  - KEYDST_KEYREG-Key register in the CHA selected by an OPERATION command.

  - KEYDST_PK_E-The 'e' register in the public key block

  - KEYDST_MD_SPLIT-Message digest IPAD/OPAD direct load.

- cover-Key was encrypted, and must be decrypted during the load. If trusted descriptor, use TDEK, else use JDEK to decrypt.

  - KEY_CLEAR-key is cleartext, no decryption needed

  - KEY_COVERED-key is ciphertext, decrypt.

- imm-Key can either be referenced, or loaded into the descriptor immediately following the command for improved performance.

  - ITEM_REFERENCE-a pointer follows the command.

  - ITEM_INLINE-key data follows the command, padded out to a descriptor word boundary.

- purpose-Sends the key to the class 1 or 2 CHA as selected by an OPERATION command. If dest is KEYDST_PK_E, this must be ITEM_CLASS1.

Returns:

If successful, returns a pointer to the target word incremented past the newly-inserted command (including item pointer or inlined data). Effectively, this becomes a pointer to the next word to receive a new command in this descriptor. If error, returns 0

9.9.10.9.1.5 cmd_insert_proto_op_ipsec()

`cmd_insert_proto_op_ipsec()`-Insert an IPSec protocol operation command into a descriptor.

```
u_int32_t *cmd_insert_proto_op_ipsec(u_int32_t     *descwd,
                           u_int8_t                cipheralg,
                           u_int8_t                authalg,
                           enum protdir            dir);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instrction. For an `OPERATION` instruction, this is normally the final word of a single descriptor.
- `cipheralg`-blockcipher selection for this protocol descriptor. This should be one of `CIPHER_TYPE_IPSEC_`.
- `authalg`-authentication selection for this protocol descriptor. This should be one of `AUTH_TYPE_IPSEC_`.
- `dir`-Select `DIR_ENCAP` for encapsulation, or `DIR_DECAP` for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.6 cmd_insert_proto_op_wimax()

`cmd_insert_proto_op_wimax()`-Insert an 802.16 WiMAX protocol OPERATION instruction into a descriptor. These can only operate as AES-CCM.

```
u_int32_t *cmd_insert_proto_op_wimax(u_int32_t     *descwd,
                           u_int8_t                mode,
                           enum protdir            dir);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction. For an `OPERATION` instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- `mode`-a nonzero value selects OFDMa, else assume OFDM operation.
- `dir`-Select `DIR_ENCAP` for encapsulation, or `DIR_DECAP` for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.7 cmd_insert_proto_op_wifi()

`cmd_insert_proto_op_wifi()`-Insert an 802.11 WiFi protocol OPERATION command into a descriptor.

```
u_int32_t *cmd_insert_proto_op_wifi(u_int32_t  *descwd,
                        enum protdir           dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.8 cmd_insert_proto_op_macsec()

cmd_insert_proto_op_macsec()-Insert an MacSec protocol OPERATION instruction into a descriptor.

```
u_int32_t *cmd_insert_proto_op_macsec(u_int32_t    *descwd,
                        enum protdir               dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.9 cmd_insert_proto_op_unidir()

cmd_insert_proto_op_unidir()-Insert a unidirectional protocol OPERATION instruction into a descriptor.

```
u_int32_t *cmd_insert_proto_op_unidir(u_int32_t *descwd, u_int32_t protid,
                                                  u_int32_t protinfo);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- protid -Select any PROTID field needed for a unidirectional protocol descriptor from OP_PCLID_.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.10 cmd_insert_alg_op()

cmd_insert_alg_op()-Insert a simple algorithm OPERATION instruction into a descriptor.

```
u_int32_t *cmd_insert_alg_op(u_int32_t *descwd, u_int32_t optype,
                                    u_int32_t algtype, u_int32_t algmode,
                                    enum mdstatesel mdstate, enum icvsel icv,
                                    enum algdir dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- optype-specify as class 1 or 2 using OP_TYPE_CLASSx_ALG.

- `algtype`-cipher selection, specify one of `ALG_TYPE_`.
- `algmode`-cipher mode selection, specify one of `ALG_MODE_`. Some combinations are ORable depending on application.
- `mdstate`-if a message digest is being processed, selects the processing state. May be one of `MDSTATE_UPDATE`, `MDSTATE_INIT`, `MDSTATE_FINAL`, or `MDSTATE_COMPLETE`.
- `icv`-if processing a message digest, or a cipher with an inclusing authentication function, then `ICV_CHECK_ON` selects an inline signature comparison on the computed result.
- `protid` -Select any PROTID field needed for a unidirectional protocol descriptor from `OP_PCLID_`.
- `dir`-Select `DIR_ENCAP` for encapsulation, or `DIR_DECAP` for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.11 cmd_insert_pkha_op()

`cmd_insert_pkha_op()`-Insert a PKHA-algorithm OPERATION instruction into a descriptor.

```
u_int32_t *cmd_insert_pkha_op(u_int32_t *descwd, u_int32_t pkmode);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `pkmode`-mode selection bits, an OR of `OP_ALG_PKMODE_` from one of the 3 possible PKHA sets (clear memory, modular arithmetic, copy memory).

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.12 cmd_insert_seq_in_ptr()

`cmd_insert_seq_in_ptr()`: Insert an `SEQ IN PTR` command into a descriptor

```
int *cmd_insert_seq_in_ptr(u_int32_t  *descwd,
                           u_int32_t  *ptr,
                           u_int32_t  len,
                           enum ref_type sgref);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this command. For an `OPERATION` command, this is normally the final word of a single descriptor.
- `ptr`-bus address pointing to the input data buffer
- `len`-input length
- `sgref`-pointer is actual data, or a scatter-gather list representing the key:
  - `PTR_DIRECT`-points to data
  - `PTR_SGLIST`-points to SEC4.x-specific scatter gather table.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.13 cmd_insert_seq_out_ptr()

`cmd_insert_seq_out_ptr()`: Insert an `SEQ OUT PTR` command into a descriptor

```
int *cmd_insert_seq_out_ptr(u_int32_t *descwd,
                            u_int32_t *ptr,
                            u_int32_t len,
                            enum ref_type sgref);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this command. For an `OPERATION` command, this is normally the final word of a single descriptor.
- `ptr`-bus address pointing to the output data buffer
- `len`-output length
- `sgref-`pointer is actual data, or a scatter-gather list representing the key:
  - `PTR_DIRECT`-points to data
  - `PTR_SGLIST`-points to SEC4.x-specific scatter gather table.

Returns:

Pointer to next incremental descriptor word past the instruction just insertted. If an error occurred, returns 0.

9.9.10.9.1.14 cmd_insert_load()

`cmd_insert_load()`: Insert a `LOAD` instruction into a descriptor:

```
u_int32_t *cmd_insert_load(u_int32_t *descwd, void *data,
                                  u_int32_t class_access, u_int32_t sgflag,
                                  u_int32_t dest, u_int8_t offset,
                                  u_int8_t len, enum item_inline imm)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `data`-pointer to data to be loaded.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB` -access class 1 objects in CCB
  - `LDST_CLASS_2_CCB` -access class 2 objects in CCB
  - `LDST_CLASS_DECO` -access DECO objects
- `sgflag`-specify `LDST_SGF` if data reference points to a scatter/gather list representing the data.
- `dest` - internal destination for the LOAD. Should be one of `LDST_SRCDST_`.
- `offset` - starting point for writing in the destination.
- `len` - length of data in bytes.
- `imm` - if specified as `ITEM_INLINE`, data is inlined into the descriptor immediately following the LOAD instruction.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.1.15 cmd_insert_seq_load()

`cmd_insert_seq_load()`: Insert a `SEQ LOAD` instruction into a descriptor:

```
        int *cmd_insert_seq_load(u_int32_t *descwd,
 unsigned int class_access,
 int variable_len_flag,
 unsigned char dest,
 unsigned char offset,
 unsigned char len);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB` -access class 1 objects in CCB
  - `LDST_CLASS_2_CCB` -access class 2 objects in CCB
  - `LDST_CLASS_DECO` -access DECO objects
- `variable_len_flag`-use the variable input sequence length
  - `dest`-destination
  - `offset`-the start point for writing in the destination
  - `len`-length of data in bytes

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

9.9.10.9.1.16 cmd_insert_fifo_load()

`cmd_insert_fifo_load()`: Insert a FIFO LOAD instruction into a descriptor

```
u_int32_t *cmd_insert_fifo_load(u_int32_t *descwd, void *data, u_int32_t len,
                                u_int32_t class_access, u_int32_t sgflag,
                                u_int32_t imm, u_int32_t ext, u_int32_t type)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `data` - pointer to data to be loaded.
- `len` - length of load data in bytes.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB`-access class 1 objects in CCB
  - `LDST_CLASS_2_CCB`-access class 2 objects in CCB
  - `LDST_CLASS_DECO`-access DECO objects

- `sgflag`-data points to a scatter/gather list representing the data to be loaded.
- `imm` - specify `FIFOLDST_IMM` if fdata is to be included immediately following this instruction.
- `ext` - if length needs to be >16 bits, specify `FIFOLDST_EXT` to inlclude the extended length in a word following the instruction.
- `type`-FIFO input data type specified as `FIFOLD_TYPE_`.r

Returns:

Pointer to next incremental descriptor word past the instruction just constructed. If an error occurred, returns 0.

9.9.10.9.1.17 cmd_insert_seq_fifo_load()

`cmd_insert_seq_fifo_load()`: Insert a SEQ FIFO LOAD instruction into a descriptor

```
u_int32_t *cmd_insert_seq_fifo_load(u_int32_t *descwd, u_int32_t class_access,
                                                    u_int32_t variable_len_flag,
                                                    u_int32_t data_type, u_int32_t len)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB`-access class 1 objects in CCB
  - `LDST_CLASS_2_CCB`-access class 2 objects in CCB
  - `LDST_CLASS_DECO`-access DECO objects
- `variable_len_flag`-use the variable input sequence length
- `data_type`-FIFO input data type (`FIFOLD_TYPE_*` in `desc.h`)
- `len`-input data length

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.1.18 cmd_insert_store()

cmd_insert_store(): Insert a STORE instruction into a descriptor

```
u_int32_t *cmd_insert_store(u_int32_t *descwd, void *data,
                                    u_int32_t class_access, u_int32_t sg_flag,
                                    u_int32_t src, u_int8_t offset,
                                    u_int8_t len, enum item_inline imm)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- data - pointer to the data store location.
- class_access
- LDST_CLASS_IND_CCB-access class-independent objects in CCB
- LDST_CLASS_1_CCB-access class 1 objects in CCB
- LDST_CLASS_2_CCB-access class 2 objects in CCB

- LDST_CLASS_DECO-access DECO objects
- sgflag-if LDST_SGF, the data pointer references a scatter/gather list describing the buffer to receive the stored data.
- src - data souce specification, one of LDST_SRCDST_
- offset-offset into source to begin store operation.
- len-store length in bytes.
- imm - if LDST_IMM, then the data to be stored follows the instruction in the descriptor.

Returns:

1. Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.1.19 cmd_insert_seq_store()

`cmd_insert_seq_store()`: Insert a SEQ STORE instruction into a descriptor

```
u_int32_t *cmd_insert_seq_store(u_int32_t *descwd, u_int32_t class_access,
                                        u_int32_t variable_len_flag, u_int32_t src,
                                        u_int8_t offset, u_int8_t len);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB`-access class 1 objects in CCB
  - `LDST_CLASS_2_CCB`-access class 2 objects in CCB
  - `LDST_CLASS_DECO`-access DECO objects
- `variable_len_flag`-if `LDST_VLF`, uses the variable sequence output length.
- `src` - data souce specification, one of `LDST_SRCDST_`
- `offset`-offset into source to begin store operation.
- `len`-store length in bytes.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.1.20 cmd_insert_fifo_store()

`cmd_insert_fifo_store()`: Insert a FIFO STORE instruction into a descriptor

```
u_int32_t *cmd_insert_fifo_store(u_int32_t *descwd, void *data, u_int32_t len,
                                         u_int32_t class_access, u_int32_t sgflag,
                                         u_int32_t imm, u_int32_t ext, u_int32_t type)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `data` - pointer to data to be stored from FIFO.
- `len` - length of data to store.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB

  - LDST_CLASS_1_CCB-access class 1 objects in CCB
  - LDST_CLASS_2_CCB-access class 2 objects in CCB
  - LDST_CLASS_DECO-access DECO objects
- sgflag-if FIFOLDST_SGF, data points to a scatter/gather list describing the buffer to be used for the store.
- imm - if FIFOLDST_IMM, store data is to be inlined into the descriptor itself, immediately following the generated instructuion.
- ext-if FIFOLDST_EXT, length exceeds 16 bits, and therefore cannot be included in the instruction itself. Write the extended length out to a word following the instruction.
- type-FIFO input type, an OR combination of FIFOST_TYPE_ type and last/flush bits for class1 and 2.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.1.21 cmd_insert_seq_fifo_store()

cmd_insert_seq_fifo_store(): Insert a SEQ FIFO STORE instruction into a descriptor

```
u_int32_t *cmd_insert_seq_fifo_store(u_int32_t *descwd, u_int32_t class_access,
                                                u_int32_t variable_len_flag,
                                                u_int32_t out_type, u_int32_t len)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- len - length of data to store.
- class_access
  - LDST_CLASS_IND_CCB-access class-independent objects in CCB
  - LDST_CLASS_1_CCB-access class 1 objects in CCB
  - LDST_CLASS_2_CCB-access class 2 objects in CCB
  - LDST_CLASS_DECO-access DECO objects
- sgflag-if FIFOLDST_SGF, data points to a scatter/gather list describing the buffer to be used for the store.
- imm - if FIFOLDST_IMM, store data is to be inlined into the descriptor itself, immediately following the generated instructuion.
- ext-if FIFOLDST_EXT, length exceeds 16 bits, and therefore cannot be included in the instruction itself. Write the extended length out to a word following the instruction.
- type-FIFO input type, an OR combination of FIFOST_TYPE_ type and last/flush bits for class1 and 2.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.1.22 cmd_insert_jump()

cmd_insert_jump(): Insert a JUMP instruction into a descriptor

```
u_int32_t *cmd_insert_jump(u_int32_t *descwd, u_int32_t jtype,
                                    u_int32_t class, u_int32_t test, u_int32_t cond,
                                    int8_t offset, u_int32_t *jmpdesc)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- jtype - type of jump to perform, one of JUMP_TYPE_
- class
  - CLASS_NONE -jump is not a checkpoint
  - CLASS_1 -jump is a checkpoing for a class 1 operation
  - CLASS_1 -jump is a checkpoint for a class 2 operation
  - CLASS_BOTH -jump is a checkpoint for both classes
- test -selects how to assess the conditional test, one of JUMP_TEST_
- cond - OR combination of conditions to test, based on the test type selected in test. May be a combination of JUMP_COND_. Note that the JSL bit is factored into the definitions for JUMP_COND_, and therefore there are two possible combinational sets.
- offset -relative offset of descriptor words to jump to if JUMP_TYPE_LOCAL is selected. May be a positive or negative offset.
- jmpdesc -address of descriptor to jump to is JUMP_TYPE_NONLOCAL is selected.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.1.23 cmd_insert_math()

cmd_insert_math(): Insert a MATH instruction into a descriptor

```
u_int32_t *cmd_insert_math(u_int32_t *descwd, u_int32_t func,
                                  u_int32_t src0, u_int32_t src1,
                                  u_int32_t dest, u_int32_t len,
                                  u_int32_t flagupd, u_int32_t stall,
                                  u_int32_t immediate, u_int32_t *data)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- func - math functuon to perform, one of MATH_FUN_.
- src0 -first source operand, one of MATH_SRC0_.
- src1 - second source operand, one of MATH_SRC1_. Note differences between what can be selected between SRC0 and SRC1.
- dest - destination operand for the result, one of MATH_DEST_.
- flagupd - specify MATH_NFU if the flags should not be updated.
- stall -specify MATH_STL to cause the instruction to consume an extra clock cycle.
- immediate -specify MATH_IFB to use 4 bytes of immediate data when the length needs to remain as 8.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.1.24 cmd_insert_move()

`cmd_insert_move()`: Insert a MOVE instruction into a descriptor

```
u_int32_t *cmd_insert_move(u_int32_t *descwd, u_int32_t waitcomp,
                                u_int32_t src, u_int32_t dst, u_int8_t offset,
                                u_int8_t length)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `waitcomp` - specify `MOVE_WAITCOMP` if intending to stall execution until the move completes.
- `src` -data source, one of `MOVE_SRC_`.
- `dest` - destination, one of `MOVE_DEST_`.
- `offset` - offset into source for move.
- `length` -length of data to move.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

9.9.10.9.2 Upper-Tier DCL Functions - Descriptor Constructors

A higher level of functionality is provided through complex descriptor constructors. These constructors are single-purpose functions capable of generating complete descriptors from user specifications. These constructors fit into two categories, one for job descriptors and another for shared descriptors generally targeted to protocol processing.

These constructors are by no means the “definitive” reference to all possible descriptor permutations, nor are they meant to work with any specific application. Instead, they are meant to be general-purpose examples of descriptor construction. It is expected that, over time, this library will grow to accommodate a wide range of examples.

All constructor functions are dependent on the underlying command generator.

9.9.10.9.2.1 Job descriptor constructors

9.9.10.9.2.1.1 cnstr_seq_jobdesc()

`cnstr_seq_jobdesc()`: Construct simple sequence job descriptor

```
          int cnstr_seq_jobdesc(u_int32_t *jobdesc, unsigned short *jobdescsz,

       u_int32_t *shrdesc, unsigned short shrdescsize,

       unsigned char *inbuf, unsigned long insize,

       unsigned char *outbuf, unsigned long outsize);
```

Inputs:

- `jobdesc`-pointer to buffer in which to build descriptor in
- `jobdescsz`-max size of descriptor build buffer
- `shrdesc`-pointer to associated shared descriptor holding session contex
- `shrdescsz`-size of associated shared descriptor
- `inbuf`-pointer to input frame

- `insize`-size of input frame
- `outbuf`-pointer to output frame
- `outsize`-size of output frame

Constructs a simple job descriptor, emulating QI-level frame processing behavior useful at the job queue level. Besides a target descriptor output, this constructor depends on three references.

1. A pointer to a shared descriptor to do the work. This is normally assumed to be some sort of a protocol-level shared descriptor.
2. A pointer to a packet/frame for input data
3. A pointer to a packet/frame for output data

The constructed descriptor is a simple reverse-order-execution descriptor, and has no provisions for other content specifications.

9.9.10.9.2.1.2 cnstr_jobdesc_blkcipher_cbc()

Construct a job descriptor capable of performing a CBC blockcipher operation:

```
int cnstr_jobdesc_blkcipher_cbc(u_int32_t *descbuf, u_int16_t *bufsz,
                                        u_int8_t *data_in, u_int8_t *data_out,
                                        u_int32_t datasz,
                                        u_int8_t *key, u_int32_t keylen,
                                        u_int8_t *iv, u_int32_t ivlen,
                                        enum algdir dir, u_int32_t cipher,
                                        u_int8_t clear);
```

Inputs:

- `descbuf` - Pointer to DMA-able buffer for descriptor construction.
- `bufsz` - Size of constructed descriptor (as output)
- `data_in` - Pointer to input message
- `data_out` - Pointer to output message
- `datasz` - Size of input/output messages
- `key` - Pointer to cipher key
- `keylen` - Size of cipher key
- `iv` - Pointer to cipher IV
- `ivlen` - Size of cipher IV
- `dir` - Direction of cipher operation, select DIR_ENCRYPT or DIR_DECRYPT
- `cipher` - Blockcipher algorithmselection chosen from OP_ALG_ALGSEL_.
- `clear` - Clear descriptor buffer before construction

Returns: -1 on construction error, 0 if construction succeeded.

9.9.10.9.2.1.3 cnstr_jobdesc_hmac()

Construct a job descriptor capable of performing an HMAC operation:

```
int32_t cnstr_jobdesc_hmac(u_int32_t *descbuf, u_int16_t *bufsize,
                                    u_int8_t *msg, u_int32_t msgsz, u_int8_t *digest,
                                    u_int8_t *key, u_int32_t cipher, u_int8_t *icv,
                                    u_int8_t clear);
```

Inputs:

- `descbuf` - descriptor buffer
- `bufsize` - limit/returned descriptor buffer size
- `msg` - pointer to message being processed
- `msgsz` - size of message in bytes
- `digest` - output buffer for digest (size derived from cipher)
- `key` - key data (size derived from cipher)
- `cipher` - OP_ALG_ALGSEL_MD5/SHA1-512
- `icv` - HMAC comparison for ICV, NULL if no check desired
- `clear` - clear buffer before writing

Returns: -1 on construction error, 0 if construction succeeded.

9.9.10.9.2.1.4 cnstr_jobdesc_mdsplitkey()

Generate an MDHA "split key" from HMAC key content. A split key is a precomputed IPAD/OPAD pair; MDHA can save cycles during sequential packet processing on a flow by using the precomputed pair directly, and thus saving the pad generation step for each packet.

Generally, the split key is generated at flow setup time as a control-plane activity, thus, this step is performed as a job descriptor.

```
int cnstr_jobdesc_mdsplitkey(u_int32_t *descbuf, u_int16_t *bufsize,
                             u_int8_t *key, u_int32_t cipher,
                             u_int8_t *padbuf);
```

Inputs:

- `descbuf` - pointer to buffer to hold constructed descriptor
- `bufsize` - pointer to size of descriptor once constructed
- `key` - pointer to HMAC key to generate pad pair from. Key size is determined by cipher selection. Note that SHA224/384 pairs are not truncated to the digest size:

Table 292.

	Key size	Split key size	Buffer size
OP_ALG_ALGSEL_MD5	16	32	32
OP_ALG_ALGSEL_SHA1	20	40	48
OP_ALG_ALGSEL_SHA224	28	64	64
OP_ALG_ALGSEL_SHA256	32	64	64
OP_ALG_ALGSEL_SHA384	48	128	128
OP_ALG_ALGSEL_SHA512	64	128	128

- `cipher` - HMAC algorithm selection, one of OP_ALG_ALGSEL_
- `padbuf` - buffer to store generated ipad/opad. Should be 2x the untruncated HMAC keysize for the chosen cipher rounded up to the nearest 16-byte boundary (where 16 bytes = an AES blocksize). See table under "`key`" above.

Returns: -1 on construction error, 0 if construction succeeded.

9.9.10.9.2.1.5 cnstr_jobdesc_aes_gcm()

Construct a job descriptor capable of performing an AES-GCM operation:

```
int cnstr_jobdesc_aes_gcm(u_int32_t *descbuf, u_int16_t *bufsize,
                              u_int8_t *key, u_int32_t keylen, u_int8_t *ctx,
                              enum mdstatesel mdstate, enum icvsel icv, enum algdir dir,
                              u_int8_t *in, u_int8_t *out, u_int16_t size, u_int8_t *mac);
```

Inputs:

- `descbuf` - pointer to buffer that will hold constructed descriptor
- `bufsiz` - pointer to size of descriptor once constructed
- `key` - pointer to AES key
- `keylen` - AES key length
- `ctx` - points to GCM context block. This is a concatenation of: MAC (128 bits), Yi (128 bits), Y0 (128 bits), IV (64 bits), and text bitsize (64 bits). See the AESA section of the blockguide for more information.
- `mdstate` - select `MDSTATE_UPDATE`, `MDSTATE_INIT`, or `MDSTATE_FINAL` if a partial MAC operation is desired, else select `MDSTATE_COMPLETE`.
- `icv` - select `ICV_CHECK_ON` if a MAC compare is requested.
- `dir` - select `DIR_ENCRYPT` or `DIR_DECRYPT` as needed for cipher operation
- `in` - Pointer to input text buffer
- `out` - Pointer to output data text
- `size` - Size of data to be processed
- `mac` - Pointer to output MAC. This can point to the head of context if an updated MAC is required for subsequent operations.

Returns: -1 on construction error, 0 if construction succeeded.

9.9.10.9.2.1.6 cnstr_jobdesc_kasumi_f8()

Construct a job descriptor capable of performing a Kasumi f8 (confidentiality) operation:

```
int cnstr_jobdesc_kasumi_f8(u_int32_t *descbuf, u_int16_t *bufsz,
                              u_int8_t *key, u_int32_t keylen,
                              enum algdir dir, u_int32_t *ctx,
                              u_int8_t *in, u_int8_t *out, u_int16_t size);
```

Inputs:

- `descbuf` - pointer to buffer that will hold constructed descriptor
- `bufsiz` - pointer to size of descriptor once constructed
- `key` - pointer to KFHA cipher key
- `keylen` - cipher key length
- `dir` - select `DIR_ENCRYPT` or `DIR_DECRYPT` as needed
- `ctx` - points to preformatted f8 context block, containing the 32-bit count (word 0), bearer (word 1 bits 7:16), and cb (word 1 bits 17:31). Refer to the KFHA section of the block guide for more detail.
- `in` - Pointer to input data text

- `out` - Pointer to output data text
- `size` - Size of the data to be processed

Returns: -1 on construction error, 0 if construction succeeded.

9.9.10.9.2.1.7 cnstr_jobdesc_kasumi_f9()

Construct a job descriptor capable of performing a Kasumi f9 (authentication) operation:

```
int cnstr_jobdesc_kasumi_f9(u_int32_t *descbuf, u_int16_t *bufsz,
                            u_int8_t *key, u_int32_t keylen,
                            enum algdir dir, u_int32_t *ctx,
                            u_int8_t *in, u_int16_t size, u_int_t *mac);
```

Inputs:

- `descbuf` - pointer to buffer that will hold constructed descriptor
- `bufsiz` - pointer to size of descriptor once constructed
- `key` - pointer to cipher key
- `keylen` - size of cipher key
- `dir` - select `DIR_ENCRYPT` or `DIR_DECRYPT` as required
- `ctx` - points to preformatted f8 context block, containing 32-bit count (word 0), bearer (word 1 bits 0:5), direction (word 1 bit 6), ca (word 1 bits 7:16), cb (word 1 bits 17:31), fresh (word 2), and the ICV input (word 3). Refer to the KFHA section of the block guide for more detail
- `out` - pointer to input data
- `out_siz` - size of input data
- `mac` - pointer to output MAC

Returns: -1 on construction error, 0 if construction succeeded.

9.9.10.9.2.1.8 cnstr_jobdesc_pkha_rsaexp()

Construct a job descriptor capable of performing an RSA exponentiation operation:

```
int cnstr_jobdesc_pkha_rsaexp(u_int32_t *descbuf, u_int16_t *bufsz,
                                  struct pk_in_params *pkin,
                                  u_int8_t *out, u_int32_t out_siz,
                                  u_int8_t clear);
```

Inputs:

- `descbuf` - pointer to buffer to hold descriptor
- `bufsiz` - pointer to size of written descriptor
- `pkin` - Values of A, B, E, and N
- `out` - Encrypted output
- `out_siz` - size of buffer for encrypted output
- `clear` - nonzero if descriptor buffer space is to be cleared before construction

Returns: -1 on construction error, 0 if construction succeeded.

9.9.10.9.2.1.9 cnstr_jobdesc_dsaverify()

Construct a job descriptor capable of performing DSA signature verification:

```
int cnstr_jobdesc_dsaverify(u_int32_t *descbuf, u_int16_t *bufsz,
                                  struct dsa_pdb *dsadata, u_int8_t *msg,
                                  u_int32_t msg_sz, u_int8_t clear);
```

Inputs:

- `descbuf` - pointer to descriptor buffer for construction
- `bufsz` - pointer to size of descriptor constructed (output)
- `dsadata` - pointer to DSA parameters
- `msg` - pointer to input message for verfication
- `msg_sz` - size of message to verify
- `clear` - clear buffer before writing descriptor

Returns: -1 on construction error, 0 if construction succeeded.

9.9.10.9.2.2 Protocol/shared descriptor constructors

These constructors build a full protocol-level shared descriptor used for semi-autonomous processing of secured traffic through SEC4.x. Such descriptors function as single-pass processors (integrating cipher and authentication functions into a single logical step) with the added factor of performing protocol-level packet manipulation in the same step in the packet-handling process, by maintaining protocol-level connection state information within the descriptor itself.

9.9.10.9.2.2.1 cnstr_pcl_shdesc_ipsec_cbc_decap()

Note: this function is deprecated in 2.0, and will be removed in a future release. Use `cnstr_shdsc_ipsec_decap()` instead.

`cnstr_pcl_shdesc_ipsec_cbc_decap()`: Shared protocol-level descriptor for IPSec CBC decapsulation. This function can create a descriptor capable of either tunnel or transport mode processing.

```
        int32_t cnstr_pcl_shdsc_ipsec_cbc_decap(u_int32_t           *descbuf,
                                                u_int16_t           *bufsize,
                                                struct pdbcont      *pdb,
                                                struct cipherparams *cipherdata,
                                                struct authparams   *authdata,
                                                u_int8_t             clear);
```

Inputs:

- `descbuf`-Points to a buffer to construct the descriptor in. All SEC4.x descriptors are built of an array of up to sixty-three 32-bit words. If the caller wishes to construct a descriptor directly in the executable buffer, then that buffer must be hardware DMA-able, and physically contiguous.
- `bufsize`-Points to an unsigned 16-bit word with the maximum length of the buffer to hold the descriptor. This will be written back to with the actual size of the descriptor once constructed. (Note: bounds checking not yet implemented).
- `pdb`-Points to a block of data (struct pdbcont) used to describe the content if the Protocol Data Block to be maintained inside the descriptor. PDB content is protocol and mode specific:
  - `pdb.opthdrlen` = Size of inbound header to skip over.
  - `pdb.transmode` = `PDB_TUNNEL/PDB_TRANSPORT` for tunnel or transport handling for the next header.
  - `pdb.pclvers` = `PDB_IPV4/PDB_IPV6` as appropriate for this connection.
  - `pdb.seq.esn` = `PDB_NO_ESN` unless extended sequence numbers are to be supported, then `PDB_INCLUDE_ESN`.

- pdb.seq/antirplysz = PDB_ANTIRPLY_NONE if no antireplay window is to be maintained in the PDB. Otherwise may be PDB_ANTIRPLY_32 for a 32-entry window, or PDB_ANTIRPLY_64 for a 64-entry window.

- cipherdata-Points to a block of data used to describe the cipher information for encryption/decryption of packet content:
  - algtype-one of CIPHER_TYPE_IPSEC_xxx
  - key-pointer to the cipher key data
  - keydata-size of the key data in bits
- authdata-Points to a block of data used to describe the authentication information for validating the authenticity of the packet source.
  - algtype-one of AUTH_TYPE_IPSEC_xxx
  - key-pointer to the HMAC key data
  - keydata-size of the key data in bits
- clear-If nonzero, buffer is cleared before writing

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.2 cnstr_pcl_shdesc_ipsec_cbc_encap()

Note: this function is deprecated in 2.0, and will be removed in a future release. Use cnstr_shdsc_ipsec_encap() instead.

cnstr_pcl_shdesc_ipsec_cbc_encap(): Shared protocol-level descriptor for IPSec CBC encapsulation. This function can construct a descriptor for either transport or tunnel mode operation.

```
int32_t cnstr_pcl_shdsc_ipsec_cbc_encap(u_int32_t  *descbuf,
                                        u_int16_t            *bufsize,
                                        struct pdbcont       *pdb,
                                        struct cipherparams  *cipherdata,
                                        struct authparams    *authdata,
                                        u_int8_t clear);
```

Inputs:

- descbuf-Points to a buffer to construct the descriptor in. All SEC4.x descriptors are built of an array of up to sixty-three 32-bit words. If the caller wishes to construct a descriptor directly in the executable buffer, then that buffer must be hardware DMA-able, and physically contiguous.
- bufsize-Points to an unsigned 16-bit word with the maximum length of the buffer to hold the descriptor. This will be written back to with the actual size of the descriptor once constructed. (Note: bounds checking not yet implemented).
- pdb-Points to a block of data (struct pdbcont) used to describe the content if the Protocol Data Block to be maintained inside the descriptor. PDB content is protocol and mode specific:
  - pdbinfo.opthdrlen = Size of outbound IP header to be prepended to output.
  - pdbinfo.opthdr = Pointer to the IP header to be prepended to the output, of size opthdrlen.
  - pdbinfo.transmode = PDB_TUNNEL/PDB_TRANSPORT for tunnel/transport handling for the next header.
  - pdbinfo.pclvers = PDB_IPV4/PDB_IPV6 as appropriate for this connection.
  - pdbinfo.seq.esn = PDB_NO_ESN unless extended sequence numbers are to be supported, then PDB_INCLUDE_ESN.

- pdbinfo.ivsrc = PDB_IV_FROM_PDB if the IV is to be maintained in the PDB, else PDB_IV_FROM_RNG if the IV is to be generated internally by SEC4.x's random number generator.

- cipherdata-Points to a block of data used to describe the cipher information for encryption/decryption of packet content:
  - algtype-one of CIPHER_TYPE_IPSEC_xxx
  - key-pointer to the cipher key data
  - keydata-size of the key data in bits
- authdata-Points to a block of data used to describe the authentication information for validating the authenticity of the packet source.
  - algtype-one of AUTH_TYPE_IPSEC_xxx
  - key-pointer to the HMAC key data
  - keydata-size of the key data in bits
- clear-If nonzero, buffer is cleared before writing

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.3 cnstr_shdsc_ipsec_encap()

Construct a shared protocol-level descriptor capable of performing IPSec ESP packet encapsulation:

```
int32_t cnstr_shdsc_ipsec_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                                struct ipsec_encap_pdb *pdb, u_int8_t *opthdr,
                                struct cipherparams *cipherdata,
                                struct authparams *authdata);
```

Inputs:

- descbuf - Pointer to buffer used for descriptor construction
- bufsize - Pointer to size to be written back upon completion
- pdb - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the encapsulation PDB, and it's cipher-dependent unioned substructure.
- opthdr - Pointer to the optional header meant to be prepended to an encapsulated frame. Size of the optional header is defined in pdb.opt_hdr_len.
- cipherdata - Pointer to blockcipher transform definitions
- authdata - Pointer to authentication transform definitions. Note that an MDHA split key is to be used with this descriptor (potentially constructed using cnstr_jobdesc_mdsplitkey()), and so the size of the uncovered split key is to be specified here, not the size of the encrypted split key buffer. See the description of cnstr_jobdesc_mdsplitkey() for a detailed discussion of split key lengths versus buffer sizes

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.4 cnstr_shdsc_ipsec_decap()

Construct a shared protocol-level descriptor capable of performing IPSec ESP packet decapsulation:

```
int32_t cnstr_shdsc_ipsec_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                                struct ipsec_encap_pdb *pdb,
```

```
                                struct cipherparams *cipherdata,
                                struct authparams *authdata);
```

Inputs:

- `descbuf`

  - Pointer to buffer used for descriptor construction

- `bufsize`

  - Pointer to size to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the decapsulation PDB, and it's cipher-dependent unioned substructure.

- `cipherdata`

  - Pointer to blockcipher transform definitions

- `authdata`

  - Pointer to authentication transform definitions. Note that an MDHA split key is to be used with this descriptor (potentially constructed using

    `cnstr_jobdesc_mdsplitkey()`

    ), and so the size of the uncovered split key is to be specified here, not the size of the encrypted split key buffer. See the description of

    `cnstr_jobdesc_mdsplitkey()`

    for a detailed discussion of split key lengths versus buffer sizes

Returns:

1. -1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.5 cnstr_shdsc_wifi_encap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.11i WiFi packet encapsulation:

```
int32_t cnstr_shdsc_wifi_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                               struct wifi_encap_pdb *pdb,
                               struct cipherparams *cipherdata);
```

Inputs:

- `descbuf`

  - Pointer to buffer used for descriptor construction

- `bufsize`

  - Pointer to size to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.

- `cipherdata`

  - Pointer to blockcipher transform definitions

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.6 cnstr_shdsc_wifi_decap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.11i WiFi packet decapsulation:

```
int32_t cnstr_shdsc_wifi_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                               struct wifi_decap_pdb *pdb,
                               struct cipherparams *cipherdata);
```

Inputs:

- `descbuf`

  - Pointer to buffer used for descriptor construction

- `bufsize`

  - Pointer to size to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the decapsulation PDB.

- `cipherdata`

  - Pointer to blockcipher transform definitions

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.7 cnstr_shdsc_wimax_encap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.16 WiMAX message encapsulation:

```
int32_t cnstr_shdsc_wimax_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                                        struct wimax_encap_pdb *pdb,
                                        struct cipherparams *cipherdata,
                                        u_int8_t mode);
```

Inputs:

- `descbuf`

- Pointer to buffer used for descriptor construction

- `bufsize`

  - Pointer to size value to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.

- `cipherdata`

  - Pointer to cipher parameters. Only

  `key`

  and

  `keylen`

  are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.8 cnstr_shdsc_wimax_decap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.16 WiMAX message decapsulation:

```
int32_t cnstr_shdsc_wimax_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                                        struct wimax_decap_pdb *pdb,
                                        struct cipherparams *cipherdata,
                                        u_int8_t mode);
```

Inputs:

- `descbuf`

  - Pointer to buffer used for descriptor construction

- `bufsize`

  - Pointer to size value to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the decapsulation PDB.

- `cipherdata`

  - Pointer to cipher parameters. Only

  `key`

and

```
keylen
```

are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.9 cnstr_shdsc_macsec_encap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.1AE MACsec message encapsulation:

```
int32_t cnstr_shdsc_macsec_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                                         struct macsec_encap_pdb *pdb,
                                         struct cipherparams *cipherdata);
```

- `descbuf`

  - Pointer to buffer used for descriptor construction

- `bufsize`

  - Pointer to size value to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.

- `cipherdata`

  - Pointer to cipher parameters. Only

  `key`

  and

  `keylen`

  are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.10 cnstr_shdsc_macsec_decap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.1AE MACsec message decapsulation:

```
int32_t cnstr_shdsc_macsec_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                                         struct macsec_decap_pdb *pdb,
                                         struct cipherparams *cipherdata);
```

Inputs:

- `descbuf`

- Pointer to buffer used for descriptor construction

- `bufsize`

  - Pointer to size value to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.

- `cipherdata`

  - Pointer to cipher parameters. Only

  `key`

  and

  `keylen`

  are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.11 cnstr_shdsc_snow_f8()

Construct a shared descriptor capable of performing SNOW UEA2 confidentiality message processing:

```
int32_t cnstr_shdsc_snow_f8(u_int32_t *descbuf, u_int16_t *bufsize,
                                u_int8_t *key, u_int32_t keylen,
                                enum algdir dir, u_int32_t count,
                                u_int8_t bearer, u_int8_t direction);
```

Inputs:

- `descbuf`

  - pointer to descriptor-under-construction buffer

- `bufsize`

  - points to size to be updated at completion

- `key`

  - cipher key

- `keylen`

  - size of key in bits

- `dir`

  - cipher direction (DIR_ENCRYPT/DIR_DECRYPT)

- `count`

  - UEA2 count value (32 bits)

- `bearer`

  - UEA2 bearer ID (5 bits)

- `direction`

  - UEA2 direction (1 bit)

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.12 cnstr_shdsc_snow_f9()

Construct a shared descriptor capable of performing SNOW UIA2 message authentication:

```
int32_t cnstr_shdsc_snow_f9(u_int32_t *descbuf, u_int16_t *bufsize,
                                    u_int8_t *key, u_int32_t keylen,
                                    enum algdir dir, u_int32_t count,
                                    u_int32_t fresh, u_int8_t direction);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to descriptor size to be updated upon completion

- `key`

  - Cipher key

- `keylen`

  - Size of cipher key

- `dir`

  - Cipher direction (

  `DIR_ENCRYPT/DIR_DECRYPT`

)

- `count`

  - UEA2 count value (32 bits)

- `fresh`

  - UEA2 fresh value ID (32 bits)

- `direction`

  - UEA2 direction (1 bit)

- `clear`

  - Nonzero if descriptor buffer clear requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.13 cnstr_shdsc_kasumi_f8()

Construct a shared descriptor capable of performing Kasumi f8 confidentiality message processing:

```
int32_t cnstr_shdsc_snow_f8(u_int32_t *descbuf, u_int16_t *bufsize,
                                    u_int8_t *key, u_int32_t keylen,
                                    enum algdir dir, u_int32_t count,
                                    u_int8_t bearer, u_int8_t direction);
```

Inputs:

- `descbuf`

  - pointer to descriptor-under-construction buffer

- `bufsize`

  - points to size to be updated at completion

- `key`

  - cipher key

- `keylen`

  - size of key in bits

- `dir`

  - cipher direction (

  `DIR_ENCRYPT/DIR_DECRYPT`

)

- `count`

  -f8count value (32 bits)

- `bearer`

  - f8 bearer ID (5 bits)

- `direction`

  - f8 direction (1 bit)

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.14 cnstr_shdsc_kasumi_f9()

Construct a shared descriptor capable of performing Kasumi f9 message authentication:

```
int32_t cnstr_shdsc_snow_f9(u_int32_t *descbuf, u_int16_t *bufsize,
                                    u_int8_t *key, u_int32_t keylen,
                                    enum algdir dir, u_int32_t count,
                                    u_int32_t fresh, u_int8_t direction);
```

Inputs:

- descbuf - Pointer to buffer for descriptor construction
- bufsize - Pointer to descriptor size to be updated upon completion
- key - cipher key
- keylen - size of cipher key
- dir - cipher direction (DIR_ENCRYPT/DIR_DECRYPT)
- count - f9 count value (32 bits)
- fresh - f9 fresh value ID (32 bits)
- direction - f9 direction (1 bit)

Returns:

1. -1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.15 cnstr_shdsc_cbc_blkcipher()

Construct a shared descriptor capable of performing CBC blockcipher confidentiality processing:

```
int32_t cnstr_shdsc_cbc_blkcipher(u_int32_t *descbuf, u_int16_t *bufsize,
                                          u_int8_t *key, u_int32_t keylen,
                                          u_int8_t *iv, u_int32_t ivlen,
                                          enum algdir dir, u_int32_t cipher,
                                          u_int8_t clear);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to descriptor size to be updated upon completion

- `key`

  - Pointer to cipher key

- `keylen`

  - Size of cipher key

- `iv`

  - Pointer to IV data

- `ivsize`

  - Size of IV

- `dir`

  - Cipher direction (

  `DIR_ENCRYPT/DIR_DECRYPT)`

- `cipher`

  - Cipher selection (

  `OP_ALG_ALGSEL_AES/DES/3DES)`

- `clear`

  - Nonzero if descriptor buffer clear is requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.16 cnstr_shdsc_hmac()

Construct a shared descriptor capable of performing hashed message authentication processing:

```
int32_t cnstr_shdsc_hmac(u_int32_t *descbuf, u_int16_t *bufsize,
                                u_int8_t *key, u_int32_t cipher, u_int8_t *icv,
                                u_int8_t clear);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to size of descriptor to be updated upon

- `key`

  - Pointer to key data. Note that key length will be automatically selected based on the HMAC cipher chosen.

- `cipher`

  - HMAC cipher selection, one of

  `OP_ALG_ALGSEL_MD5/SHA1/SHA224/SHA256/SHA384/SHA512`

- `icv`

  - HMAC comparison for ICV, NULL if no check desired

- `clear`

  - Nonzero if descriptor buffer clear is requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.17 cnstr_pcl_shdsc_3gpp_rlc_decap()

Note: this is a P4080 tapeout #1 operation, to be replaced in a future release.

Construct a shared descriptor capable of performing 3GPP RLC message decapsulation operations:

```
int32_t cnstr_pcl_shdsc_3gpp_rlc_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                                       u_int8_t *key, u_int32_t keysz,
                                       u_int32_t count, u_int32_t bearer,
                                       u_int32_t direction,
                                       u_int16_t payload_sz, u_int8_t clear);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to size of descriptor to be updated upon completion

- `key`

  - Pointer to f8 cipher key

- `keysz`

  - Size of cipher key

- `count`

  - f8 count value

- `bearer`

  - f8 bearer value

- `direction`

  - f8 direction value

- `payload_sz`

  - Size of payload to be processed (descriptor generated does not use VLF).

- `clear`

- clear descriptor buffer before construction

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.2.2.18 cnstr_pcl_shdsc_3gpp_rlc_encap()

Note: this is a P4080 tapeout #1 operation, to be replaced in a future release.

Construct a shared descriptor capable of performing 3GPP RLC message encapsulation operations:

```
int32_t cnstr_pcl_shdsc_3gpp_rlc_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                                       u_int8_t *key, u_int32_t keysz,
                                       u_int32_t count, u_int32_t bearer,
                                       u_int32_t direction,
                                       u_int16_t payload_sz);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to size of descriptor to be updated upon completion

- `key`

  - Pointer to f8 cipher key

- `keysz`

  - Size of cipher key

- `count`

  - f8 count value

- `bearer`

  - f8 bearer value

- `direction`

  - f8 direction value

- `payload_sz`

  - Size of payload to be processed (descriptor generated does not use VLF).

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

9.9.10.9.3 Disassembler

9.9.10.9.3.1 caam_desc_disasm()

```
caam_desc_disasm()
```

-Top-level descriptor disassembler

```
        void caam_desc_disasm(u_int32_t *desc, u_int32_t opts);
```

Inputs:

- `desc`

  -points to the descriptor to disassemble. First command must be a header, or shared header, and the overall size to disassemble is determined by the header. Does not handle a QI preheader as its first command, and cannot yet follow links in a list of descriptors.

- `opts`

  - selects options to add to the disassembled output:

  `DISASM_SHOW_OFFSETS`

  - shows the index/offset of each instruction in the descriptor preceding the textual disassembly. This is useful for visualizing flow control changes in a descriptor, since any offset to a specific instruction in the disassembly will be displayed both as a relative number of instructions, and as the offset of the specific instruction.

  `DISASM_SHOW_RAW`

  - shows the hexadecimal value of each instruction before the displayed value of the instruction itself.

9.9.11 Runtime Assembler Library Reference

Use the Runtime Assembler Library to write SEC descriptors.

9.9.11.1 Runtime Assembler Library Reference

Use the Runtime Assembler Library to write SEC descriptors. This reference describes the structure, concept, functionality, and high level API.

For more information, click here.

9.9.12 USDPAA PME Loopback User Guide

9.9.12.1 Introduction

The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document describes the "pme_loopback" application. This application serves as an example for working with the PME USDPAA interface, as well as providing a benchmark for PME USDPAA system performance.

9.9.12.1.1 Purpose

This document describes the USDPAA pme_loopback application.

The material is technical in nature. The reader is assumed to be familiar with:

- General Linux software development, operation, and configuration for Power architecture devices in particular.
- Familiarity with the concept of device drivers and their role in operating systems.
- Linux network administration and use.
- The NXP Linux SDK for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.
- The USDPAA User Guide document

9.9.12.1.2 Change history

Table 293. Change History

Version	Updates
1.0	Initial creation of pme_loopback UG.

9.9.12.2 Overview of pme_loopback

The pme_loopback is an interactive command line driven USDPAA application which generates and consumes frames to/from the PME hardware accelerator via the PME USDPAA APIs. The pme_loopback application focuses on demonstrating I/O performance through the PME from applications running on multiple cores. The application creates up to one core affine thread per core which sends a pre-generated frame to PME for scanning and processes the resulting scan result. The application does very little processing of the scan responses. it only does enough to determine if a match was found and updates state information.

When instructed via the CLI, each application thread does the following:

- initializes a pme_ctx object
- prepares an FD and associated payload data for PME scanning
- commences sending pme scan requests and processing the scan results
- terminates sending scan request and process all expected scan result
- displays performance results
- disables and frees pme_ctx object



Figure 273. pme_loopback block diagram

9.9.12.2.1 pme_loopback application flow

The pme_loopback application transitions into various states as instructed by the CLI commands sent.



Figure 274. pme_loopback state transition diagram

Not all commands are valid depending on the current state. Unexpected behaviour may occur if an invalid command is sent. At this time the application does not validate if the request should not be performed. Below is the suggested method of running the pme_loopback application. Detailed description of pme_loopback cli commands can be found in pme_loopback application syntax on page 1804.

1. start the pme_loopback application with the desired number of threads
2. Create the pme_ctx objects associated with each thread via create_ctx_direct_mode or create_ctx_flow_mode commands.
3. prepare the threads FD data for scanning via the prep_scan or prep_scan_2 commands

4. start the scanning loop via start_scan command
5. after letting the application scan data through PME for a desired amount of time (e.g. 30 seconds), stop the scanning process via the stop_scan command.
6. collect the performance results displayed by the application
7. free the memory used by the threads for scanning via the free_mem command
8. return to step 3 if there is a desire to scan data with different characteristics (e.g. SUI size, SUI pattern, etc) as defined by the prep_scan and prep_scan_2 commands. Otherwise proceed to the next step.
9. Delete the pme_ctx objects associated with each thread via the delete_ctx command.
10. To re-initialize the pme_ctx objects associated with each thread return to step 2 or continue to next step
11. remove the threads and quit the application. The quit command will do both of these steps.

9.9.12.3 pme_loopback application syntax

The pme_loopback USDPAA application is a command line driven application. Each command instructs the application to perform a task which is usually tightly co-related to a specific PME USDPAA API.

Summary of commands:

```
        pme_loopback_test [core_ids]
        create_ctx_direct_mode [thread_ids]
        create_ctx_flow_mode sesson_id ren [thread_ids]
        prep_scan sui_size_in_bytes pattern_width low_threshold high_threshold use_compound_frame
[thread_ids]
        prep_scan_2 sui_size_in_bytes pattern_data low_threshold high_threshold
use_compound_frame [thread_ids]
        start_scan [thread_ids]
        stop_scan [thread_ids]
        display_stats [thread_ids]
        clear_stats [thread_ids]
        free_mem [thread_ids]
        delete_ctx [thread_ids]
        rm [core_ids]
        add [core_ids]
        list
        help
        quit
```

9.9.12.3.1 pme_loopback_test

USDPAA pme_loopback test application

Synopsis

```
pme_loopback_test [core_ids]
```

Description

Start the pme_loopback USDPAA application on the specified cores. This is an interactive command line driven application. A prompt ">" is displayed awaiting instructions. If no option is specified a single USDPAA thread is running on core 0.

OPTIONS

```
core_ids
```

Identifies which cores shall have a USDPAA thread created on. It identifies either a single core or a list of cores. The format is as follows:

```
single_core | first_core..last_core
single core = minimum core id up to maximum core id
first_core < last_core
```

EXAMPLES

To start pme_loopback on cores 0,1,2,3,4,5,6 and 7

```
pme_loopback_test 0..7
```

To start pme_loopback on core 2

```
pme_loopback_test 2
```

9.9.12.3.2 create_ctx_direct_mode

Initialize a pme_ctx object on the specified threads.

Synopsis

```
create_ctx_direct_mode [thread_ids]
```

Description

Each specified thread will invoke the pme_ctx_init() API. This API initializes the pme_ctx object which is associated to the thread and will specify the DIRECT pme_ctx flag. If no option is specified the command is sent to all the threads.

OPTIONS

```
thread_ids
```

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
where: first_thread < last_thread
```

EXAMPLES

To initialize pme_ctx object in direct mode on all pme_loopback USDPAA threads:

```
>create_ctx_direct_mode
```

To initialize a pme_ctx object in direct mode on the USDPAA threads which are on cores 0 to 7:

```
>create_ctx_direct_mode 0..7
```

To initialize a pme_ctx object in direct mode on the USDPAA thread which is on core 5:

```
>create_ctx_direct_mode 5
```

9.9.12.3.3 create_ctx_flow_mode

Initialize a pme_ctx object in flow mode on the specified threads.

Synopsis

```
create_ctx_flow_mode session_id ren [thread_ids]
```

Description

Each specified thread will invoke the pme_ctx_init() API. This API initializes the pme_ctx object which is associated to the thread. The pme_ctx will be initialized in flow mode and will have the specified session_id and whether its residue is on or off is determined by the ren parameter. If no thread_ids option is specified the command is sent to all the threads.

`session_id`

Index where the per-session context for this Flow will be accessed by SRE in the Session Context Table. The minimum session_id is 0. The maximum session_id is equal to /dev/fsl-pme-dev/sre_session_ctx_num - 1.

ex: `# cat /dev/fsl-pme-dev/sre_session_ctx_num 80`

Therefore the maximum session_id is 79.

`ren`

Indicates is residue is enabled or not. zero (0) indicates residue is disabled. One (1) indicated residue is enabled.

OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

`single_thread | first_thread..last_thread`

where: `first_thread < last_thread`

EXAMPLES

To initialize pme_ctx object in flow mode on all pme_loopback USDPAA threads with session_id zero and residue disabled:

```
>create_ctx_flow_mode 0 0
```

To initialize a pme_ctx object in flow mode with session_id zero and residue enabled on the USDPAA threads which are on cores 0 to 7:

```
>create_ctx_flow_mode 0 1 0..7
```

To initialize a pme_ctx object in flow mode with session_id 1 and residue enabled on the USDPAA thread which is on core 5:

```
>create_ctx_direct_mode 1 1 5
```

9.9.12.3.4 prep_scan

Prepare the specified threads for pme scanning.

Synopsis

```
prep_scan sui_size_in_byte pattern_width low_threshold high_threshold use_compound_frame [thread_ids]
```

Description

Prepares the specified threads for pme scanning. This command will allocate memory for the SUI and will construct an appropriate FD: a compound frame is used if specified. The resulting FD will be used to repeatedly send pme scans. The thread will send scan requests until the high_threshold is reached. It will then switch to processing scan results until the low_threshold is reached at which point it will return to sending scans. The thread remains in this loop until the "stop_scan" command is received.

The content of the SUI is determined as follows:

There is an internal 50 character alphabet

```
1 2 3 4 5 6 7 8 9 0 a b c d e f g h i j k l m n o p q r s t u v w x y z ! @ # $ % ^ & * ( ) [ ] { } ?
" ;
```

The SUI which is of sui_size_in_byte byte long is filled with the above alphabet in an alternating pattern. Any remaining bytes are filled with the period "." symbol. The pattern is as follows:

- first character in the alphabet is repeated every 1 x pattern_width bytes

- second character in the alphabet is repeated every 2 x pattern_width bytes

- etc...

For example if a sui_size of 65 and a pattern_width of 5 is chosen the resulting pattern will emerge:

"1" is repeated every 5 bytes.

"2" is repeated every 10 bytes.

"3" is repeated every 15 bytes.

...etc

```
        1XXXX
        12XXX
        1X3XX
        12X4X
        1XXX5
        123XX
        1XXXX
        12X4X
        1X3XX
        12XX5
        1XXXX
        1234X
        1XXXX
```

Then the pme database can be populated with the pmm tool with appropriate signatures. For instance if the signature /4/ is set in the database and the above SUI is sent then there will be a match frequency of every 20 bytes.

If no thread_ids option is specified the command is sent to all the threads.

`sui_size_in_bytes`

This number of bytes will be allocated for the SUI which will be used to construct a frame descriptor. Initially the entire SUI is filled with the period character "."

`pattern_width`

The SUI is filled with an alternating pattern which is of this width. If zero, no alternating pattern will be used, the default all period "." pattern is used. The maximum value is 50.

`low_threshold`

The specified threads process scan results until the number of outstanding scans reduces to this amount.

`high_threshold`

The specified threads generate scan requests until the number of outstanding requests reaches this count: at which point the thread starts to process scan results.

`use_compound_frame`

A frame descriptor can be either be contiguous or a compound frame. If use_compound_frame is 0 then a contiguous frame format is used, otherwise a compound frame format is used. The compound frame output frame is of zero size bytes.

OPTIONS

`thread_ids`

Identifies which core affined thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

`single_thread | first_thread..last_thread`

```
where: first_thread < last_thread
```

EXAMPLES

The following example does the following:

- prepare the SUI in all threads with SUI size of 65 bytes
- no pattern width will be used (default all period)
- a low_threshold in flight SUI count of 15and a high_threshold in flight SUI count of 30
- compound_frame is not used (i.e. contiguous is used

```
>prep_scan 65 0 15 30 0
```

9.9.12.3.5 prep_scan_2

Prepare the specified threads for pme scanning.

Synopsis

```
prep_scan_2 sui_size_in_bytes pattern_data low_threshold high_threshold use_compound_frame
[thread_ids]
```

Description

This is a second version of the prep_scan command. The command is similar to prep_scan except that this version has the pattern_data parameter instead of pattern_width. The pattern_data is the literal string of characters that will copied into the SUI. If the pattern_data is larger than the SUI size than the pattern_data will be truncated. If the pattern_data is shorter than the SUI, the SUI is filled with the period "." character. This command will allocate memory for the SUI and will construct an appropriate frame descriptor (FD): a compound frame is used if specified. The resulting FD will be used to repeatedly send pme scans. The thread will send scan requests until the high_threshold is reached. It will then switch to processing scan results until the low_threshold is reached at which point it will return to sending scans. The thread remains in this loop until the "stop_scan" command is received.

If no thread_ids option is specified the command is sent to all the threads.

`sui_size_in_bytes`

This number of bytes will be allocated for the SUI which will be used to construct a frame descriptor. Initially the entire SUI is filled with the period character "."

`pattern_data`

A literal string of characters that will be copied in the SUI.

`low_threshold`

The specified threads process scan results until the number of outstanding scans reduces to this amount.

`high_threshold`

The specified threads generate scan requests until the number of outstanding requests reaches this count: at which point the thread starts to process scan results.

`use_compound_frame`

A frame descriptor can be either be contiguous or a compound frame. If use_compound_frame is 0 then a contiguous frame format is used, otherwise a compound frame format is used. The compound frame output frame is of zero size bytes.

OPTIONS

`thread_ids`

Identifies which core affined thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

EXAMPLES

The following example does the following:

- prepare the SUI in all threads with SUI size of 10 bytes
- the SUI pattern being sent is abcdefghij
- a low_threshold in flight SUI count of 15and a high_threshold in flight SUI count of 30
- compound_frame is not used (i.e. contiguous is used

```
>prep_scan_2 10 abcdefghij 15 30 0
```

9.9.12.3.6 start_scan

Instruct the specified threads to commence sending pme scan requests and processing the results.

Synopsis

```
start_scan [thread_ids]
```

Description

Each specified thread will enter the following loop:

```
        sending scan loop
        do
        invoke the pme_ctx_scan
        increment in_flight_scans by one
        while (in flight scans < high_threshold)
        processing scan response loop
        do
        qman_poll_dqrr(16)
        while (in_flight_scans >= low_threshold)
```

The qman_poll_dqrr() api will invoke the function callback specified in the pme_ctx object. This callback will decrement the in_flight_scans counter and also update some statistics such as number of notifications and truncations received.

The threads will remain in this loop until the stop_scan command is received.

OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

EXAMPLES

To instruct all threads to start their scanning loop:

```
>start_scan
```

To instruct threads 0 to 4 to start its scanning loop:

```
>start_scan 0..4
```

To instruct thread 2 to start its scanning loop:

```
>start_scan 2
```

9.9.12.3.7 stop_scan

Instruct the specified threads to stop sending pme scan requests and complete processing all remaining scan results.

Synopsis

```
stop_scan [thread_ids]
```

Description

Each specified thread will stop sending scan requests and will process all scan results until the in_flight_scan counter reaches zero. The following performance statistics are displayed:

- Total units scanned
- Total number of SUIs sent to the PME device.
- Total time
- The time in seconds from the moment the first scan unit is sent until the last scan response is processed.
- Scan Units per second
- Total units scanned / Total time
- Bandwidth

The number of SUIs send and corresponding responses processed measured in Mbps.

i.e. 9512 Mbps

OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

EXAMPLES

To instruct all threads to stop their scanning loop:

```
>stop_scan Total units scanned: 24622356 Total time: 31.007910 sec Scan Units per second: 794066
Bandwidth: 9528 Mbps
```

9.9.12.3.8 free_mem

Instruct the specified threads to free the memory previously allocated via the prep_scan or prep_scan_2 command.

Synopsis

```
free_mem [thread_ids]
```

Description

Each specified thread will free their memory previously allocated during the prep_scan and prep_scan_2 command. This command needs to be run to undo any previous prep_scan or prep_scan_2 commands.

OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where:

```
first_thread < last_thread
```

EXAMPLES

To instruct all threads to free their allocated memory:

```
>free_mem
```

9.9.12.3.9 delete_ctx

Instruct the specified threads to disable and "finish" their previously initialized pme_ctx object.

Synopsis

```
delete_ctx [thread_ids]
```

Description

Each specified thread will invoked the pme_ctx_disabled() API on their pme_ctx object. Once completed they will invoked the pme_ctx_finish() API. These APIs free any internal resources used (such as frame queues) and render the pme_ctx object in an un-initialized state once again.

OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

EXAMPLES

To instruct all threads to disable and finish their pme_ctx object:

```
>delete_ctx
```

9.9.12.3.10 rm

Remove a pme USDPAA core affined thread

Synopsis

```
rm thread_ids
```

Description

Each specified thread will be destroyed by the application. The application will display that the thread on a specific cpu has been killed.

`thread_ids`

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

EXAMPLES

To instruct threads 0 to 7 to be removed:

```
>rm 0..7 Thread killed on cpu 0 Thread killed on cpu 1 Thread killed on cpu 2 Thread killed on cpu 3
Thread killed on cpu 4 Thread killed on cpu 5 Thread killed on cpu 6 Thread killed on cpu 7
```

9.9.12.3.11 add

Add a pme USDPAA core affined thread

Synopsis

add core_ids

Description

Each specified core will have a pme USDPAA thread created by the application.

core _ids

Identifies which cores shall have an affine USDPAA thread created. The format is as follows:

single_thread | first_thread..last_thread

where: first_thread < last_thread

EXAMPLES

To instruct the application to create threads on cores 1 to 2:

>add 1..2 Thread alive on cpu 1 Thread alive on cpu 2

9.9.12.3.12 list

Display a list of pme USDPAA core affine threads.

Synopsis

```
list
```

Description

A command is sent to each thread. In response the thread will display on which core it is running on.

EXAMPLES

```
>list Thread alive on cpu 0 Thread alive on cpu 1 Thread alive on cpu 2 Thread alive on cpu 3 Thread
alive on cpu 4 Thread alive on cpu 5 Thread alive on cpu 6 Thread alive on cpu 7
```

9.9.12.3.13 display_stats

Instruct the specified threads to display their internal statistics

Synopsis

```
display_stats [thread_ids]
```

Description

Each specified thread will display their internal statistics.

OPTIONS

```
thread_ids
```

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

EXAMPLES

To instruct thread 0 to display its internal statistics:

```
>display_stats 0 in_flight = 0 rx_packets = 0 total_notifs = 0 num_queue_empty = 0 num_erns = 0
num_truncates = 0 full_fifo = 0 Time: 0.000000 secs
```

9.9.12.3.14 clear_stats

Instruct the specified threads to clear their internal statistics

Synopsis

```
clear_stats [thread_ids]
```

Description

Each specified thread will clear their internal statistics.

OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The thread_id has the same value as the core_ids indicated in pme_loopback_test on page 1804. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

EXAMPLES

To instruct all threads to clear their internal statistics:

```
>clear_stats
```

9.9.12.3.15 help

Display the list of pme_loopback commands

Synopsis

```
help
```

Description

Prints out all pme_loopback commands

OPTIONS

EXAMPLES

To display the list of command:

```
>help
```

Available commands:

```
help add list rm create_ctx_flow_mode create_ctx_direct_mode delete_ctx prep_scan prep_scan_2
start_scan stop_scan free_mem display_stats clear_stats
```

9.9.12.3.16 quit

Quit the pme_loopback application

Synopsis

```
quit
```

Description

This will shutdown the pme_loopback application. All threads will first be removed and then the application will exit.

9.9.12.4 Running pme_loopback

Log in to the p4080 DS environment as "root".

```
          login: root
          Password:
          [root@p4080 root]#
```

Clear the pme database via pmm application:

```
          [root@p4080 root]#pmm
Successfully created the PMM DB.
pmm> commit
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
pmm> quit
Terminating the PMM application.
[root@p4080 root]#
```

Start the pme_loopback application with threads on cores 0 to 7:

```
          /usr/bin/pme_loopback_test 0..7
          Qman: FQID allocator includes range 512:128
          Bman: BPID allocator includes range 56:8
          FSL dma_mem device mapped (phys=0xf8000000,virt=0x70000000,sz=0x4000000)
          Thread alive on cpu 0
          Thread alive on cpu 1
          Thread alive on cpu 2
          Thread alive on cpu 3
          Thread alive on cpu 4
          Thread alive on cpu 5
          Thread alive on cpu 6
          Thread alive on cpu 7
          >
```

A CLI (Command-Line Interface) is presented in order to allow you to enter the next pme_loopback commands.

Create a pme context in direct mode on each pme USDPAA thread.

```
          > create_ctx_direct_mode
```

Prepare the data for scanning.

```
          > prep_scan 1024 0 15 30 0
```

Start the scanning

Once the command to start scanning is executed it will stay in this mode scanning data until the stop_scan command is entered.

```
          > start_scan
```

[... wait 30 seconds (though as low as 10 seconds can also be used) before typing in stop command below. To allow enough data to be processed by the PME to make the initial command generation overhead amortized over adequate scans. (i.e. the result is a steady state performance result)]

Command to stop Scan

```
          > stop_scan
          Total units scanned: 34608812
```

```
Total time: 29.804104 sec
Scan Units per second: 1161209
Bandwidth: 9512 Mbps
```

Need to free scan memory which was allocated

```
> free_mem
```

Tear down application, need to delete contexts

```
> delete_ctx
```

Quit application

```
> quit
```

9.9.13 USDPAA IPFwd Longest Prefix Match User Manual

9.9.13.1 NXP P4080/P5020/P3041 USDPAA IPFwd Longest Prefix Match User Manual

9.9.13.1.1 Introduction

This user manual describes USDPAA IPFwd based upon Longest Prefix Match methodology. This IPFwd application is different from the other route cache based IPFwd. The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA LPM IPFwd application
- A summary of usage of Shared MAC
- A summary of usage of MAC-less interface
- Execution steps for USDPAA LPM IPFwd application from the NXP SDK package on the P4080 DS/P3041DS/P5020DS

This document describes the USDPAA application which demonstrates:

1. IP Forwarding application using Longest prefix match as route decision algorithm
2. MAC less communication between USDPAA application and kernel
3. Sharing of same physical Ethernet port using shared-mac mode

9.9.13.1.2 Overview

The USDPAA LPM based IPv4 forwarding application is a multi-threaded application that routes IPv4 packets from one ethernet interface to another on all QorIQ platforms. The LPM routing algorithm uses the prefix for destination ip address to do the route look up. Any combination of the cores can run a LPM based USDPAA IPv4 Forwarding application thread. The packets that reach the USDPAA application can be forwarded to destination IP address using LPM route algorithm.

LPM IPFwd packet-processing:

· Receives ethernet frames on ethernet interfaces.

· On the basis of defined PCD rules in FMAN's PCD table, IPv4 traffic would be sent to USDPAA application through PCD Frame queue range.

· For IPv4 frames, processing takes place as defined in section Overview of packet flow: on page 1822

This application also demonstrates the following features:

. how the traffic coming from a common MAC port can be split in between kernel and USDPAA application on the basis of defined PCD rules. The packets that reach the USDPAA application can be forwarded to destination IP address using LPM route algorithm.

. ping between linux and USDPAA using MAC-less interface.



9.9.13.1.3 How is it different from existing Route cache based IPFwd?

This USDPAA application uses Longest Prefix Match algorithm for doing route lookup by using prefix for destination IP address in contrast to the existing route cache based IPFwd which takes route decision based upon hash results calculated by FMAN using source IP address and destination IP address in the frame.

The user will have to use new commands for route addition in LPM table (check the command section)

9.9.13.1.4 Longest Prefix Match algorithm

Instead of using traditional Radix-Trie algorithm, here we choose to use one simpler LPM algorithm. It uses 5 level tables: First level table – 65536 entries array, indexed by the top 16 bits of IP address. Second level table – 32 entries array, indexed by bit12~15 of IP address. Third level table – 32 entries array, indexed by bit8~11 of IP address. Fourth level table – 32 entries array, indexed by bit4~7 of IP address. Fifth level table – 32 entries array, indexed by bit0~3 of IP address. The 2nd level to 5th level tables are only created when its first level table entry has valid value. See below figure:



At init time, only the first level (i.e. top16 bits) array is created which contains 65536 null entries. While adding route entries to the FIB table, the 2nd level to 5th level arrays will be created accordingly. This is a typical ASIC design algorithm of LPM which is fast and simple to search while costs far more memory. The worst case is to index and compare 5 times when searching an IP address, but it's still fast enough.

Examples:

1) Add one ClassA route 10.0.0.0/8 to the route table. (gateway is 1.1.1.1, destination port is 1) The first 16bit of 10.0.0.0 (0x0a000000) is 0x0a00 (2560). And the mask is 8 bit which is smaller than the 1st level bit-length (16b), so below entries (from 0x0a00 to 0x0aff) will be created in the FIB table:



From entry No.2560 to No.2815 (total 255 entries) are filled with same content (flag, gwIP, dstPort, ptr …). Now if a packet with DIP of 10.1.1.1 comes in, its first 16 bit value is 0x0a01 (2561). So, the No.2561 entry of the 1st level table will be checked. If it's a 'leaf' node (now it is), then the best-match is found. And the packet will be forwarded to the 'dstPort' after replacing the SMAC and DMAC. And, any DIP of 10.x.x.x will all be forwarded to port 1 with gwIP 1.1.1.1 based on above table.

2) Now, a new route 10.1.1.0/24 is added to the FIB table. The table will be like this:



The new route will overwrite the No.2561 (0x0a01) entry with 'flag' of 'nonLeaf' and a pointer to 'next-level-table'. Now a 16-entry-block memory will be allocated as the 'next-level-table' because the 2nd level is 4bit indexed. And the base address of this new block will be set to the 'nxtLvlTblPtr' of No.2561. Because the next 4 bit of the new route is 0 (bit16 to bit19 of 0x0a010100), so the 1st entry in the 2nd level table is used as another 'nonLeaf' entry. While all the other entries in the 2nd level table should be filled with the same value of its 'parent' route (10.0.0.0/8). The netmask is 24bit which is larger than 16+4, so the 3rd level table should also be allocated (16 entries). And the next 4bit of the new route is 1 (bit20 to bit23 of 0x0a010100), so the 2nd entry will be used for the new route. And because the netmask (24bit) is no-larger-than 16+4+4, so this entry will be the 'Leaf' entry of this new route (see above figure in green). And the according values (gwIP, dstPort, etc.) will be filled in that entry. Now a frame with DIP of 10.1.1.100 comes. There will have 3 lookups to get the final result:

- Index with first 16bit of DIP, whose value is 0x0a01. 'Non-leaf' means to continue the next-level lookup.
- Index with the next 4bit of DIP, whose value is 0. Then 'Non-leaf' again.
- Index with the next 4bit of DIP, whose value is 1. Then the 'leaf' node is found and the lookup reaches an end.

Now a frame with DIP of 10.1.192.10 comes. You can see it will find the 'leaf' node in the 2nd level table and get the route of net-address 10.0.0.0/8. And a frame with DIP of 10.1.10.10 will find its 'leaf' node in the 3rd level table and also get the route of net-address 10.0.0.0/8 as we expected. The multi-branch trie algorithm provides a very fast way of route-lookup but a relatively complicated way of route-add/deletion.

9.9.13.1.5 Shared MAC Overview

The kernel and USDPAA should be able to receive traffic of their interest from a shared Ethernet port. On the basis of defined PCD rules in FMAN's PCD table, IPv4 traffic (for non-owned IP addresses) would be sent to USDPAA application through PCD Frame queue range and rest of the traffic (i.e. ICMP etc) would be sent to kernel through default Frame queue.



9.9.13.1.6 How to run shared MAC interface ?

1. Make sure to use the corresponding dtb to "p4080ds-usdpaa-shared-interfaces.dts" file from kernel source. In this example dts file ethernet@9 depicts shared MAC node.

   NOTE

   For p3-p5, use p3041ds-usdpaa-shared-interfaces.dts and p5020ds-usdpaa-shared-interfaces.dts respectively. ethernet@5 depicts shared MAC node.

   NOTE

   For B4, use b4860qds-usdpaa-shared-interfaces.dts where ethernet@9 depicts shared MAC node. And for T4, use t4240qds-usdpaa-shared-interfaces.dts where ethernet@15 depicts shared MAC node.

2. When linux comes up, check the output of "ifconfig -a".

   root@p4080ds:~# ifconfig -a

   fm2-10g Link encap:Ethernet HWaddr 00:e0:0c:00:96:09

   inet6 addr: fe80::2e0:cff:fe00:9609/64 Scope:Link

   UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

   RX packets:0 errors:0 dropped:0 overruns:0 frame:0

   TX packets:5 errors:1 dropped:0 overruns:0 carrier:0

   collisions:0 txqueuelen:1000

   RX bytes:0 (0.0 B) TX bytes:378 (378.0 B)

   Memory:fe5f0000-fe5f0fff

3. Run fmc

   $cd /usr/etc

   fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_shared_mac_ipv4.xml -a

   For p3-p5 use this command:

fmc -c usdpaa_config_p3_p5_serdes_0x36_shared_mac.xml -p usdpaa_policy_hash_shared_mac_ipv4.xml -a

For B4 use this command:

fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml –p usdpaa_policy_hash_shared_mac_ipv4.xml –a

For T4 use this command:

fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml –p usdpaa_policy_hash_shared_mac_ipv4.xml –a

4. Run lpm-ipfwd application

   $lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i fm2-10g

   For p3-p5

   $lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i fm1-10g

   For B4

   $lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 –c usdpaa_config_b4_serdes_0x2a_0x98.xml –p usdpaa_policy_hash_shared_mac_ipv4.xml -i fm1-mac10

   For T4

   $lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 –c usdpaa_config_t4_serdes_1_1_6_6.xml –p usdpaa_policy_hash_shared_mac_ipv4.xml -i fm2-mac10

5. Assign ip address to USDPAA side as well as kernel side as per script "usdpaa_policy_hash_shared_mac_ipv4.xml".

   In this script, the coarse classification "ip_dest_clsf" corresponding to this port uses "192.168.44.3" for USDPAA and "192.168.44.4" for kernel.

   NOTE

   Check ip address for p3-p5 as per scripts usdpaa_config_p3_p5_serdes_0x36_shared_mac.xml and usdpaa_policy_hash_shared_mac_ipv4.xml.

   NOTE

   For B4/T4, use “192.168.44.3” for USDPAA and “192.168.44.4” for kernel as defined in usdpaa_policy_hash_shared_mac_ipv4.xml.

   ssh to p4080 and give these commands:

   $lpm_ipfwd_config -E -a true

   output:

   FMAN Interface number: 11

   , PortID=1:5 is FMan interface node with MAC Address 00:e0:0c:00:96:09

   NOTE

   Check pid from application print "Message queue to send: /mq_snd_2536"

   $ lpm_ipfwd_config -P 2536 -F -a 192.168.44.1 -i 11

   $ lpm_ipfwd_config -P 2536 -G -s 192.168.44.3 -m 02:00:c0:a8:a0:02 -r true

   $ lpm_ipfwd_config -P 2536 -B -c 1 -d 192.168.44.3 -n 16 -g 192.168.44.3

   $ifconfig fm2-10g 192.168.44.4

6. Now run traffic on fm2-10g using the above ip addresses and can see traffic splitting amongst kernel and USDPAA.

9.9.13.1.7 MAC-less use case

This section describes how MAC-less interface is being used in this application. The user space configuration commands and USDPAA application can communicate with each other through the management interface which can be a MAC interface (SGMII, RGMII etc) or MAC-less interface. In this application we are just using MAC-less interface. The user can do such a communication with USDPAA LPM application by using lpm_ipfwd_config binary. For command reference, please check section IPv4 forward application Configuration command on page 1720.

9.9.13.1.8 How to ping MAC-less interface ?

. Make sure to use the corresponding dtb to "p4080ds-usdpaa-shared-interfaces.dts" file from kernel source. In this example dts file ethernet@10 depicts MAC-less node.

. NOTE: For B4, use b4860qds-usdpaa-shared-interfaces.dts and for T4, use t4240qds-usdpaa-shared-interfaces.dts. In both cases, ethernet@16 is the macless node.

. When linux comes up, check the output of "ifconfig -a". The interface containing similar mac address as given under ethernet@10 node in device tree, is the MAC-less interface.

root@p4080ds:~# ifconfig -a

eth0 Link encap:Ethernet HWaddr 00:15:17:1e:22:9e

UP BROADCAST MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:42 Memory:40000000-40020000

eth3 Link encap:Ethernet HWaddr 00:11:22:33:44:55

BROADCAST MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

The above output shows that "eth3" is the MAC-less node.

. Run lpm-ipfwd application

$cd /usr/etc

$lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i eth3:[66-22-33-44-55-66]

The mac address "66-22-33-44-55-66" is the mac that will be assigned as source mac address to USDPAA side of MAC-less interface

. Assign ip address to USDPAA side as well as kernel side.

ssh to p4080 and give these commands:

$lpm_ipfwd_config -E -a true

Output:

MACLESS Interface:

name : eth3

$lpm_ipfwd_config -F -a 192.168.55.6 -n eth3

$ifconfig eth3 192.168.55.2

$ping 192.168.55.6

PING 192.168.55.6 (192.168.55.6): 56 data bytes

64 bytes from 192.168.55.6: icmp_seq=0 ttl=64 time=12.413 ms

64 bytes from 192.168.55.6: icmp_seq=1 ttl=64 time=12.431 ms

9.9.13.1.9 USDPAA LPM based IPv4 forwarding application flow

The LPM based IPFwd application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAA PPAC User Guide for more details). The configuration phase also includes PPAM (i.e. IPFwd) related initialization. Once the configuration phase is completed the IPFwd application moves to the packet processing phase. This application provides a command-line interface to enable users to add and remove routing table and ARP cache entries. For each user input, the appropriate information is communicated to the IPFwd application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPfwd application. Note that this application does not dynamically resolve ARP – missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking “IRQ mode” whenever LPM IPFwd has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to “USDPAA PPAC User Guide”. In polling mode – the application constantly looks for data to process on its dedicated QMan portal . Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

9.9.13.1.10 Overview of packet flow:

1. Static Route table entries are populated using the user space configuration commands.
2. If a packet received by the FMan is an IPv4 packet it uses 2-tuple (Src IP & Dest IP) to hash the packet to a Rx frame queue. Otherwise if packet is ICMP etc it would be sent through default Frame queue where the buffer is freed.
3. The packet enqueued to PCD FQ to reach USDPAA LPM based IPFwd application thread running on one of the cores is subjected to LPM based route lookup.
4. All of the threads enter the processing loop where they call Qman_poll till a frame is received.
5. If a frame is received, the application checks whether the destination IP address exist in the FIB table. For this it calls ip_route_lookup(), which does the route look up using LPM algorithm. The LPM IPFWD application does not change its FIB table in response to seeing the first packet in a flow. Instead the FIB table is set only by commands, as mentioned in section Syntax on page 1720.
6. If the route entry for this frame is not present in the FIB table, the frame is dropped. It then continues with Qman_poll.
7. If the route entry for this frame exists in the FIB table, the frame is sent for forwarding.
8. If the frame is to be forwarded, it is checked if ARP entry exists in ARP table for destination IP address. LPM IPFWD application will not dynamically resolve the ARP. So if sending packets to forward using a regular computer, the user will have to create static ARP entries. On host computer, ARP table can be updated by sending the ARP request. LPM IPFWD will respond to external ARP requests
9. If ARP entry exists, TTL is decremented in L3 header.
10. Finally, the L2 header is updated, which includes changing the dst MAC address in L2 header. The frame is then enqueued to the TX FQ.

9.9.13.1.11 Overview of PPAC

The source code to LPM-IPFwd has been reorganized into two parts; the “PPAC” (Packet-Processing Application Core) and a “PPAM” (Packet-Processing Application Module). The PPAM portion implements the LPM-IPFWD application specific logic of processing the packet using longest prefix match and forward it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document “USDPAA PPAC User Guide”.

9.9.13.1.12 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired.

9.9.13.1.13 Order Preservation in LPM-IPFWD

This section describes how user can enable Order Preservation in LPM-IPFWD application. By default Order Preservation is disabled in LPM-IPFWD application and in order to enable it the user will have to re-compile the binary by making following changes to the source code.

In file, usdpaa/apps/include/ppac.h you can find these two lines.

```
/* Application options */
#undef PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */
#undef PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
```

Change the above to

```
#define PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */
#define PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
```

And then compile usdpaa once again. Now run the LPM-IPFWD application with Order Preservation.

9.9.13.1.14 Order Restoration in LPM IPFWD

Order restoration is the functionality of QMan software portal interface which restores the relative temporal order of a flow of frames (sequence of frames) to that observed before transmitting to the destination Frame Queue and Order Definition Point (ODP) takes note of the correct order of packets before start processing by using the sequence number. Use of “HOLDACTIVE” is mutually exclusive with another QMan option “AVOIDBLOCK”, which is selected by default in PPAC. To enable order-restoration, the user will have to re-compile the binary by making following changes to the source code.

```
/* Application options */
#undef PPAC_HOLDACTIVE          /* Process each FQ on one portal at a time */
#undef PPAC_ORDER_PRESERVATION    /* HOLDACTIVE + enqueue-DCAs */
#undef PPAC_2FWD_ORDER_RESTORATION    /* Use ORP */
#define PPAC_AVOIDBLOCK          /* No full-DQRR blocking of FQs */
```

Change the above to

```
#undef PPAC_HOLDACTIVE          /* Process each FQ on one portal at a time */
#undef PPAC_ORDER_PRESERVATION    /* HOLDACTIVE + enqueue-DCAs */
#define PPAC_ORDER_RESTORATION    /* Use ORP */
#define PPAC_AVOIDBLOCK          /* No full-DQRR blocking of FQs */
```

And then compile usdpaa once again. Now run the LPM-IPFWD application with Order Restoration. Implementation note: Order restoration has been implemented such that each PCD Frame queue has a corresponding ORP (order restoration point) frame queue associated with it. Each ORP is configured with default window settings as seen below

#define PPAC_ORP_WINDOW_SIZE 7 /* 0->32, 1->64, 2->128, ... 7->4096 */

#define PPAC_ORP_AUTO_ADVANCE 1 /* boolean */

#define PPAC_ORP_ACCEPT_LATE 3 /* 0->no, 3->yes (for 1 & 2->see RM) */

Here the ORP window size is set to be 4K, auto advance window size as 4K and accept late arrival window size as 8K. This ensures that no traffic is getting dropped but are always accepted below and at Zero loss throughput. Beyond zero loss throughput, as usual packets would be dropped and thus you can see mis-ordering.

ORP FQ descriptor attributes settings:

- Prefer in cache
- No “ HOLDACTIVE ”
- No “ AVOIDBLOC K”
- ORP enabled

Assumption: To see the effect of Order Restoration in LPM-IPFwd application the user must use separate streamblocks as a source of traffic. If not done so, mis-ordering would be seen.

Key observation: It has been observed in LPM-IPFwd application that use of “HOLDACTIVE” with traffic generated using separate streamblocks, all the packets are IN sequence. Therefore, it is recommended that if user wants to see the real effect of Order restoration in LPM-IPFwd application he should use “AVOIDBLOCK” with “RESTORATION” and not “HOLDACTIVE” with “RESTORATION”

9.9.13.1.15 Monitoring Rx/Tx fill-levels and flow-control via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. This simply allows the user to monitor the overall fill-level of frame queues in the system, in particular to determine whether build-up is occuring before or after the software-processing phase. Refer to section “ Monitoring Rx/Tx fill-levels via CGR ” of USDPAA PPAC User Guide for more details. To enable this feature, in ppac.h change;

#undef PPAC_CGR

to;

#define PPAC_CGR

The CGRs can also be configured to perform flow-control using Congestion state tail drop by setting CSTD_EN bits . Each congestion group record can be configured to track either byte counts or frame counts in all frame queues in the Congestion Group. When the threshold set for each CGR is exceeded, the CS bit is set in the CGR, and the congestion group is said to have entered congestion. At this point the incoming frames are marked for discard and QMAN will generate enqueue rejections to the producer. When the group’s I_BCNT returns below the threshold (minus approximately 1/8 of the threshold to provide hysteresis), the CS bit is cleared, and the congestion group’s state exits congestion. To enable tail drop, in ppac.h change;

#undef PPAC_CGR /* Track rx and tx fill-levels via CGR */

#undef PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

to

#define PPAC_CGR /* Track rx and tx fill-levels via CGR */

#define PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

And then compile usdpaa once again. Now run the IPFWD application with CGR tail drop enabled. To test this feature PPAC CLI provides a command “cgr” which will query and display all the fields of both CGRs. On pumping the traffic to IPFWD application at full line rate, t he instantaneous group byte count value I_BCNT(Instantaneous frame/byte count) must be maintained lesser than the CGR threshold set for each congestion group. Here is one such cgr command output:

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0e1e

a_bcnt: 0x00_0000_0e1e

Tx CGR ID: 11, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_0002

a_bcnt: 0x00_0000_0004

On the other hand if this feature of Congestion Group tail drop is disabled in IPFWD application I_BCNT is never maintained below CGR threshold value with traffic at full line-rate. This can be checked by compiling the IPFWD application with;

#define PPAC_CGR /* Track rx and tx fill-levels via CGR */

#undef PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

Now on pumping traffic at full line-rate, atleast one of the CGRs must go into congestion state and its I_BCNT should be above CGR threshold value. Here is the sample output:

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 1

cscn_targ: 0x00800000

cstd_en: 0

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0006

a_bcnt: 0x00_0000_0004

Tx CGR ID: 11, selected fields;

cscn_en: 1

cscn_targ: 0x00800000

cstd_en: 0

cs: 1

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_5fb7

a_bcnt: 0x00_0000_5fb8

Thus, the above test showcases the flow control achieved by enabling congestion group tail drop in IPFWD application.

9.9.13.1.16 LPM IPFWD Application Suite



The figure above shows the structure of the LPM IPFWD USDPAA application suite. Its purpose is to forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using ipfwd_config command as mentioned in section Assign IP address to interfaces on page 1840 IP address can be assigned to all these interfaces. Each interface can have a variable netmask. The MAC addresses of these interfaces are determined by u-boot environment variables ethaddr, eth1addr, eth2addr, etc.

The ipfwd application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An ipfwd_config command should be used to set these MAC addresses. On host side, ARP table can be updated by sending ARP requests to this application.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "NXP DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAA applications. See the USDPAA User Guide for more information.
3. Finally, the fmc configuration file passed by command line argument to lpm_ipfwd_app determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by lpm_ipfwd_app are:

- FMan1-TGEC1
- FMan2-DTSEC3

- FMan2-DTSEC4
- FMan2-TGEC1

Running the lpm ipfwd application suite involves four steps:

1. Run lpm_ipfwd_app
2. Run lpm_ipfwd_config repeatedly to add routes to LPM FIB table.
3. Run the fmc application to configure the FMan hardware instances.

Specific examples showing these steps are provided in other sections of this document.

9.9.13.1.17 Possible configuration scenario for LPM based IPFWD

LPM based IPfwd application can run in different configuration scenario. The table below shows the configuration files and corresponding sample shell script existing in the repository.

xml file	Sample shell script	Number of routes (as per sample shell script)	Netmask (as per sample shell script)
usdpaa_config_p4_serdes_0xe.xml (2x10G+2x1G)	lpm_ipfwd_20G.sh	1024 routes (2x10G)	16
	lpm_ipfwd_22G.sh	1024 routes (2x10G+2x1G)	16
	lpm_ipfwd_20G_1Mroutes.sh	1M/256 (4K) routes	24
usdpaa_config_p2_p3_p5_14g.xml(4x1G+10G)	lpm_ipfwd_14G.sh	1020 routes (4x1G+10G)	16



Figure 1: usdpaa_config_p4_serdes_0xe.xml

The figure above shows the ethernet interfaces available as per configuration file " usdpaa_config_p4_serdes_0xe.xml". It contains the 1 RGMII port (FMAN1, DTSEC 2), 2 SGMII ports (FMan2, DTSECs 3- 4) and 2 XAUI ports (FMAN1, TGEC1 and FMAN2-TGEC2). It is the user's wish to use any combination of ports available with the configuration file. The above table shows the sample shell scripts that user can use for this configuration. If user uses 2x10G, following is the flow configuration.

Test Center <-> P4080

Port 2 to Port 1 (512 flows) :
Dst : 190.0.24.2 - 190.255.24.2
Dst : 191.0.24.2 - 191.255.24.2
Gw : 192.168.60.2

Port 1 to Port 2 (512 flows) :
Dst : 192.0.24.2 - 192.255.24.2
Dst : 193.0.24.2 - 193.255.24.2
Gw : 192.168.160.2

Figure 2 : Flow Configuration for 2x10G on P4080DS



Port 2 to Port 1 (256 flows):
Dst : 190.0.24.2 - 190.255.24.2
Gw : 192.168.60.2

Port 1 to Port 2 (256 flows):
Dst : 193.0.24.2 - 193.255.24.2
Gw : 192.168.160.2

Port 4 to Port 3 (256 flows):
Dst : 191.0.24.2 - 191.255.24.2
Gw : 192.168.130.2

Port 3 to Port 4 (256 flows):
Dst : 192.0.24.2 - 192.255.24.2
Gw : 192.168.140.2

Figure 3 : configuration for 2x10G and 2x1G on P4080

For running LPM IPFwd on P3041DS/P5020DS, the user can use " usdpaa_config_p2_p3_p5_14g.xml". Following is the flow configuration.



Left: 10G port (port6: fm1-10g)
Right: top-to-bottom: 1G port (port1-port6)

Figure 4 : Flow Configuration for 10G on P5020DS and P3041DS

Port 1 to Port 6 (100 flows): Dst: 195.0.24.2 – 195.99.24.2 Gw: 192.168.60.2	Port 6 to Port 1 (100 flows): Dst: 190.0.24.2 – 190.99.24.2 Gw: 192.168.10.2
Port 2 to Port 6 (100 flows): Dst: 196.0.24.2 – 196.99.24.2 Gw: 192.168.60.2	Port 6 to Port 2 (100 flows): Dst: 191.0.24.2 – 191.99.24.2 Gw: 192.168.20.2
Port 3 to Port 6 (100 flows): Dst: 197.0.24.2 – 197.99.24.2 Gw: 192.168.60.2	Port 6 to Port 3 (100 flows): Dst: 192.0.24.2 – 192.99.24.2 Gw: 192.168.30.2
Port 4 to Port 6 (100 flows): Dst: 198.0.24.2 – 198.99.24.2 Gw: 192.168.60.2	Port 6 to Port 4 (100 flows): Dst: 193.0.24.2 – 193.99.24.2 Gw: 192.168.40.2
Port 5 to Port 6 (100 flows): Dst: 199.0.24.2 – 199.99.24.2 Gw: 192.168.60.2	Port 6 to Port 5 (100 flows): Dst: 194.0.24.2 – 194.99.24.2 Gw: 192.168.50.2

How to run if user has no XAUI but only SGMII riser card?

Assume user does not have a XAUI riser card and wants to run LPM IPFwd using only the SGMII ports as shown below.



This configuration contains the 1 RGMII port (FMAN1, DTSEC 2), 4 SGMII ports (FMan2, DTSECs 1- 4). The user can modify the configuration file and sample shell scripts as per requirement. The configuration file would contain the following

```
<cfgdata>
<config>
<engine name="fm1">
<port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
<port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
<port type="1G" number="2" policy="hash_ipsec_src_dst_spi_policy8"/>
<port type="1G" number="3" policy="hash_ipsec_src_dst_spi_policy9"/>
</engine>
</config>
</cfgdata>
```

User can create a new shell script which would make routes between the 4x1G. Here is the content of the script.

```
net_pair_routes()
{
net=0
while [ "$net" -le $5 ]
do
lpm_ipfwd_config -P $pid -B -c $2 -d $1.$net.24.2 -n $3 -g \
192.168.$4.2
net=`expr $net + 1`
done
}
lpm_ipfwd_config -P $pid -F -a 192.168.110.1 -i 6
lpm_ipfwd_config -P $pid -F -a 192.168.120.1 -i 7
```

lpm_ipfwd_config -P $pid -F -a 192.168.130.1 -i 8

lpm_ipfwd_config -P $pid -F -a 192.168.140.1 -i 9

lpm_ipfwd_config -P $pid -G -s 192.168.110.2 -m 02:00:c0:a8:6e:02 -r true

lpm_ipfwd_config -P $pid -G -s 192.168.120.2 -m 02:00:c0:a8:78:02 -r true

lpm_ipfwd_config -P $pid -G -s 192.168.130.2 -m 02:00:c0:a8:82:02 -r true

lpm_ipfwd_config -P $pid -G -s 192.168.140.2 -m 02:00:c0:a8:8c:02 -r true

1024

net_pair_routes 190 1 16 110 255 # 256

net_pair_routes 191 1 16 120 255 # 256

net_pair_routes 192 1 16 130 255 # 256

net_pair_routes 193 1 16 140 255 # 256

9.9.13.1.18 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.



Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that ipfwd_app has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.27.2 up

Set the default gateway for subnet 192.168.27.1

route add default gw 192.168.27.1 eth0

2. No need to add arp on host side. The application can handle external arp requests

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.28.2 up

Set the default gateway for subnet 192.168.28.1

route add default gw 192.168.28.1 eth0

2. No need to add arp on host side. The application can handle external arp requests

The commands to perform on P4080 are:

Boot the P4080 and login as root.

Assign IP address to fm1-gb1

ifconfig fm1-gb1 <IPADD> up

Assume use of cores 1 - 7 lpm_ipfwd_app 1..7 -d 0x10000000 -b 0:0:1728 -i fm1-10g,fm2-gb2,fm2-gb3,fm2-10g

Now ssh to P4080 linux on other terminal

ssh root@<IPADD>

give IP address as assigned to fm1-gb1 in the beginning

Now assign ip address to the interfaces

First run command to check what all enabled interfaces are available

Note: check pid from application print "Message queue to send: /mq_snd_2536"

lpm_ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces

Assign IP address to the interfaces.Use the same interface number displayed as an ouput on giving the above command

lpm_ipfwd_config -P 2536 -F -a 192.168.24.1 -i 5

lpm_ipfwd_config -P 2536 -F -a 192.168.28.1 -i 9

lpm_ipfwd_config -P 2536 -F -a 192.168.27.1 -i 8

lpm_ipfwd_config -P 2536 -F -a 192.168.29.1 -i 11

Now enter routes and MAC addresses. Format of a MAC address is

aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

lpm_ipfwd_config -P 2536 -B -d 192.168.28.2 -g 192.168.28.2 -n 24 -c 1

lpm_ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true

lpm_ipfwd_config -P 2536 -B -d 192.168.27.2 -g 192.168.27.2 -n 24 -c 1

lpm_ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true

Run fmc command cd /usr/etc

fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a

Now, traffic can pass between Computer X and Computer Y. For example, on Computer X

enter:

ping 192.168.28.2

9.9.13.1.19 Running LPM IPv4 forwarding on P4080DS board

The instructions below describe how to run LPM-IPFWD. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipfwd_config application is completed.

- On linux prompt, assign IP address to fm1-gb1

$ ifconfig fm1-gb1 <IPADD> up

Now run the FMC command

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a

- Run LPM-IPFWD application

The main LPM-IPFwd application binary is called **lpm_ipfwd_app** *. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:*

$ lpm_ipfwd_app <m..n>

By default lpm_ipfwd_app uses usdpaa_config_p4_serdes_0xe.xml and usdpaa_policy_hash_lpm_ipv4.xml files.

LPM-IPFWD application command syntax:

[root@p4080 etc]# lpm_ipfwd_app --usage

Usage: lpm_ipfwd_app [-n?V] [-b x:y:z] [-c FILE] [-d SIZE] [-i FILE] [-p FILE]

[--buffers=x:y:z] [--fm-config=FILE] [--dma-mem=SIZE]

[--fm-interfaces=FILE] [--non-interactive] [--fm-pcd=FILE]

[--cpu-range] [--help] [--usage] [--version] [cpu-range]

LPM-IPFWD application run command:

[root@p4080 root]# cd /usr/etc

[root@p4080 etc]# lpm_ipfwd_app 1..7 -d 0x4000000 -b 0:0:1728 -i fm1-10g,fm2-gb2,fm2-gb3,fm2-10g

To use dma region size as 64M in USDPAA for lpm-ipfwd application, make sure to set the same or greater size in bootargs. E.g. usdpaa_mem=256M

[1] 5363

[root@p4080 etc]# Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1

Found /fsl,dpaa/ethernet@4, Tx Channel = 40, FMAN = 0, Port ID = 0

Found /fsl,dpaa/ethernet@7, Tx Channel = 63, FMAN = 1, Port ID = 2

Found /fsl,dpaa/ethernet@8, Tx Channel = 64, FMAN = 1, Port ID = 3

Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0

Configuring for 4 network interfaces

Allocated DMA region size 0x10000000

lpm_ipfwd_app starting

IPv4 FIB table init now... Done!

Message queue to send: /mq_snd_2536

Message queue to receive: /mq_rcv_2536

Thread uid:0 alive (on cpu 1)

Release 0 bufs to BPID 7

Release 0 bufs to BPID 8

Release 1600 bufs to BPID 9

Thread uid:1 alive (on cpu 2)

Thread uid:2 alive (on cpu 3)

Thread uid:3 alive (on cpu 4)

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

If, in the run application command, cpu-range is given i.e. "lpm_ipfwd_app <*m..n*>" LPM-IPFWD application starts threads on cpu-range m..n. The main thread (by default on CPU 1) then does global initialization needed by the application, including starting other application threads.

If, on the other hand, run application command is given without any cpu-range i.e. "lpm_ipfwd_app" LPM-IPFWD application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren't blocked):

> list

Thread alive on cpu 1

Thread alive on cpu 2

Thread alive on cpu 3

Thread alive on cpu 4

Thread alive on cpu 5

Thread alive on cpu 6

To enable all interfaces

> macs on

To disable all interfaces

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

- Once the application starts, it can receive the configuration commands. Run application configuration script.

For creating route entries, the binary *lpm_ipfwd_config* is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the LPM-IPFwd application.

The shell script mentioned below contains sample commands to add route entries. Detailed description of all lpm_ipfwd_config commands is provided in section Syntax on page 1720.

SSH to p4080 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

lpm_ipfwd_20G.sh "pid"

$ lpm_ipfwd_ 20G.sh 2536

There are example shell scripts available. They can assign IP addresses to the interfaces, add an ARP entry and can use variable netmask while creating route entries. The following table summarizes the settings done by these scripts. Check sectionPossible configuration scenario for LPM based IPFWD on page 1827 for more details.

For the LPM-IPFwd application to forward traffic successfully, traffic destined for the P4080DS ports must have the appropriate destination IP addresses.

Console messages are printed for each entry added to the routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "LPM-IPFwd Route Creation completed" is printed on the console.

At this point, traffic can be sent to the ethernet interfaces, IPv4 packets would be processed by the LPM application on the cpu-range specified by the user on the application command-line.

9.9.13.1.20 Running LPM IPv4 forwarding on P3041/P5020 board

The instructions below describe how to run LPM-IPFWD on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the lpm_ipfwd_config application is completed. On linux prompt, assign IP address to eth0

$ ifconfig eth0 <IPADD> up

Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a

Run lpm-IPFWD

$ lpm_ipfwd_app <m..n> -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_lpm_ipv4.xml -d 0x4000000 -b 0:0:1728 -i fm1-gb0, fm1-gb1,fm1-gb3,fm1-gb4,fm1-10g

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to board (linux) on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

$ lpm_ipfwd_14G.sh

There is an example shell script available named as lpm_ipfwd_14G.sh creates routes for only the 4 x 1G and 1x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry . Check section Possible configuration scenario for LPM based IPFWD on page 1827 for more details. Now traffic can be run as per the routes created.

9.9.13.1.21 USDPAA LPM IP Fwd performance gap between 6 core and 8 core

USDPAA LPM IPfwd performance for 8 core is less than 6 core on e6500 series. USDPAA LPM IP Fwd application need to run using "-s" option to bridge the gap between two configuration.

9.9.13.1.22 PPAC (and IPFwd) CLI commands

The following commands are illustrated in the context of IPFwd, but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren't blocked):

> list

Thread uid:0 alive (on cpu 1)

Thread uid:1 alive (on cpu 2)

Thread uid:2 alive (on cpu 3)

Thread uid:3 alive (on cpu 4)

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

To remove a thread by its UID:

> rm uid:2

Thread uid:2 killed (cpu 3)

To remove a thread running on a given CPU:

> rm 5 Thread uid:4 killed (cpu 5)

To enable all interfaces:

> macs on

To disable all interfaces:

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

To query cgr

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0e1e

a_bcnt: 0x00_0000_0e1e

Tx CGR ID: 11, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_0002

a_bcnt: 0x00_0000_0004

9.9.13.1.23 Syntax

The syntax is as follows:

$ [root@p4080 bin]# ipfwd_config --help

Usage: ipfwd_config [OPTION...]

-B, --routeadd=TYPE adding a route

-C, --routedel=TYPE deleting a route

-E, --showintf=TYPE show interfaces

-F, --intfconf=TYPE change intf config

-G, --arpadd=TYPE adding a arp entry

-H, --arpdel=TYPE deleting a arp entry

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

9.9.13.1.24 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

lpfwd_config -P pid -E -a true

Table 294. Field Description (show all enabled interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	Show all the interfaces	Yes	true

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# lpfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces [root@p4080 bin]#

9.9.13.1.25 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

lpm_ipfwd_config -E -help

Help for show all enabled interfaces

[root@p4080 etc]# lpm_ipfwd_config -E --help

Usage: -E [OPTION...]

-a, --a=ALL All interfaces

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.13.1.26 Assign IP address to interfaces

The command to assign IP address to shared or private interfaces while running IPv4 forward is as follows:

lpm_ipfwd_config -P pid -F -a 192.168.60.1 -i <Interface number>

The command to assign IP address to MAC-less interfaces while running IPv4 forward is as follows:

lpm_ipfwd_config -P pid -F -a 192.168.60.1 -n <MAC-less interface name>

NOTE

Note: The interface name(MAC-less) or interface number to be used here must be one of the names/numbers that got displayed as the output of "show all enabled interfaces command" in section Command to show all enabled interfaces and their interface numbers on page 1721.

Table 295. Field description (assign IP address to shared or private MAC interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	IP Address	Yes	a.b.c.d
-i	Interface number	Yes	0-11 (Choose this number from "show all enabled interfaces" command output)

Table 296. Field description (assign IP address to MAC-less interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	IP Address	Yes	a.b.c.d
-n	MAC-less Interface name	Yes	eth3 (Choose this name in case of MAC-less from "show all enabled interfaces" command output)

Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

. Command to assign IP address to private or shared MAC interfaces

lpm_ipfwd_config -P 2536 -F -a 192.168.60.1 -i 5

IPADDR assigned = 0xc0a83c01 to interface num 5

Intf Configuration Changed successfully

. Command to assign IP address to MAC-less interface

lpm_ipfwd_config -P 2536 -F -a 192.168.55.6 -n eth3

IPADDR assigned = 0xc0a88506 to MACLESS intf eth3

Intf Configuration Changed successfully

9.9.13.1.27 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

lpm_ipfwd_config -F --help

Help for assign IP address to interfaces

[root@p4080 etc]# lpm_ipfwd_config -F --help

Usage: -F [OPTION...]

-a, --a=IPADDR IP Address

-i, --i=IFNUM If Number

-n, --n=IFNAME MACLESS Interface Name

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.13.1.28 Adding a Route Entry

The command to add a route while running IPv4 forward is as follows:

lpm_ipfwd_config -P pid -B -d b.c.d.e -g c.d.e.f -n maskbits -c numentry

Table 297. Field Description (Adding a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-d	Destination IP Address	Yes	a.b.c.d
-g	Gateway IP Address	Yes	a.b.c.d
-n	netmask length to be used by LPM	Yes	upto 32 bits
-c	number of fib entries	Yes	1- valid count

Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# lpm_ipfwd_config -P 2536 -B -d 192.168.24.2 -g 192.168.24.2 -n 24 -c 1024

Route Entry Added successfully

[root@p4080 bin]#

9.9.13.1.29 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

lpm_ipfwd_config -B --help

Help for Adding a Route Entry

[root@p4080 bin]# lpm_ipfwd_config -B --help

Usage: -B [OPTION]

-d, --d=DESTIP Destination IP

-g, --g=GWIP Gateway IP

-n, --n=MASKBITS Netmask length

-c, --c=NUMENTRY Number of fib entries

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.13.1.30 Deleting a Route Entry

NOT IMPLEMENTED

9.9.13.1.31 Help for Deleting a Route Entry

Delete route command is not implemented

9.9.13.1.32 Adding an ARP Entry

The command to add an ARP entry while running IPv4 forward is as follows:

lpm_ipfwd_config -P pid -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]

Table 298. Field Description (Adding an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d
-m	Mac Address	Yes	aa:bb:cc:dd:ee
-r	Replace existing entry	No	true/ false {Default: false}

Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# lpm_ipfwd_config -P 2536 -G -s 192.168.24.2 -m 02:00:c0:a8:33:fd -r true

ARP Entry Added successfully

9.9.13.1.33 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

lpm-ipfwd_config -G --help

Help for Adding an ARP Entry

[root@p4080 etc]# lpm_ipfwd_config -G --help

Usage: -G [OPTION...]

-m, --m=MACADDR MAC Address

-r, --r=Replace Replace Exiting Entry - true/ false {Default:

false}

-s, --s=IPADDR IP Address

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.13.1.34 Deleting an ARP Entry

The command to delete an ARP while running the IPv4 forward is as follows:

lpm_ipfwd_config -P pid -H -s a.b.c.d

Table 299. Field Description (Deleting an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d

Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# lpm_ipfwd_config -P 2536 -H -s 192.168.24.2

Arp Entry Deleted successfully

[root@p4080 bin]#

9.9.13.1.35 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

lpm_ipfwd_config -H --help

Help for Deleting an ARP Entry

[root@p4080 etc]# lpm_ipfwd_config -H --help

Usage: -H [OPTION...]

-s, --s=IPADDR IP Address

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

9.9.13.1.36 References

1. USDPAA PPAC User Guide
2. QMan/BMan API Guide

9.9.14 NXP USDPAA FRA Configuration User Manual

9.9.14.1 Introduction

9.9.14.1.1 Purpose

This document is intended to introduce how to configure the NXP RMan Application (FRA).

9.9.14.2 FRA Configuration

The NXP RMan Application is a multi-thread USDPAA application based on RMan driver. The RapidIO message manager (RMan) supports a message passing programming model for inter-processor and inter-device communication. The FRA configuration file is to configure the behavior of passing messages. This file follows standard XML rules and is composed of several elements. Each element begins with a start tag and can contain attributes and/or child elements. If the element contains child elements, it must have a corresponding end-tag after them. An element without child elements must end with a slash (/).

Note that element and attribute names are always case sensitive.

The FRA configuration file always begins with the <fra_cfg> root element. As such, the end-tag of the fra_cfg element must appear at the end of the file.

Attributes: No required attributes

For example:

```
<fra_cfg>
    <rman_cfg>
         <defcfg file="/usr/etc/rman_config.xml"/>
    </rman_cfg>

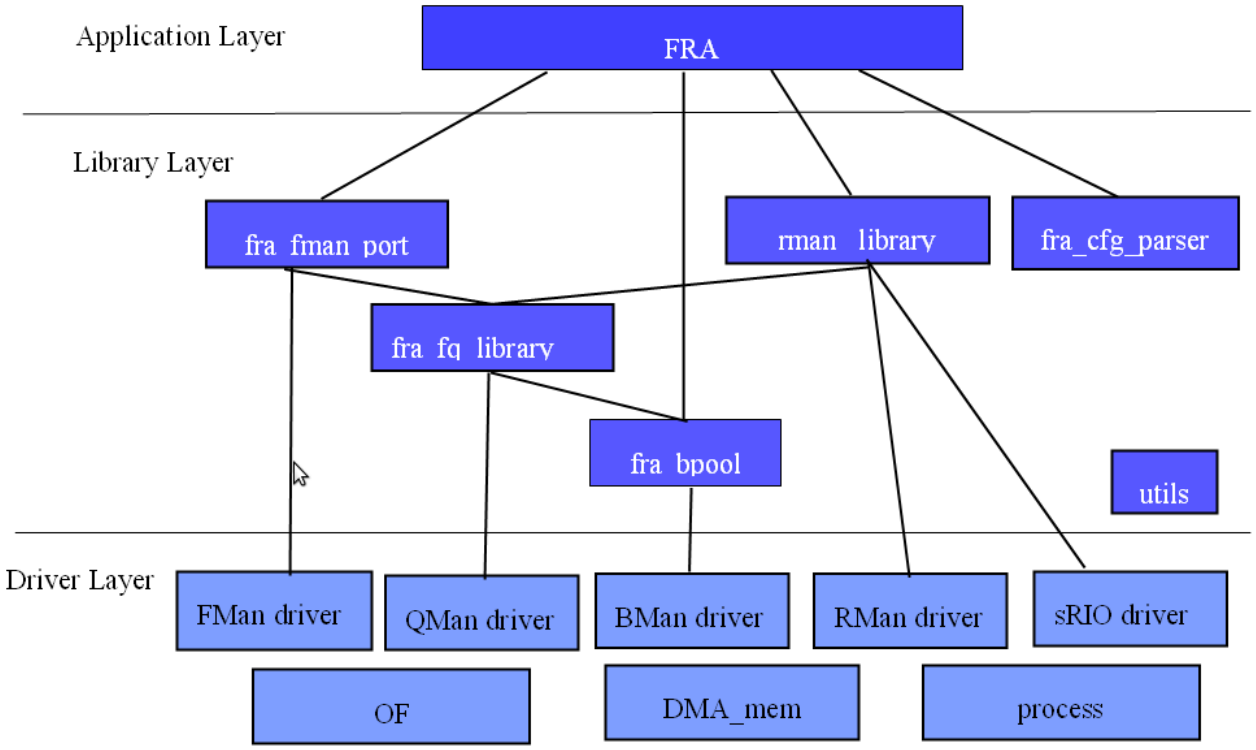
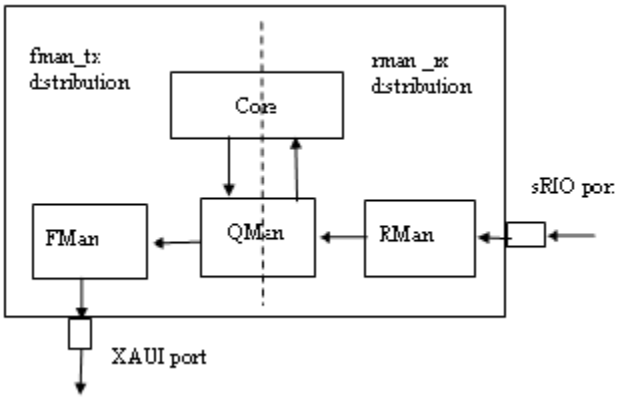
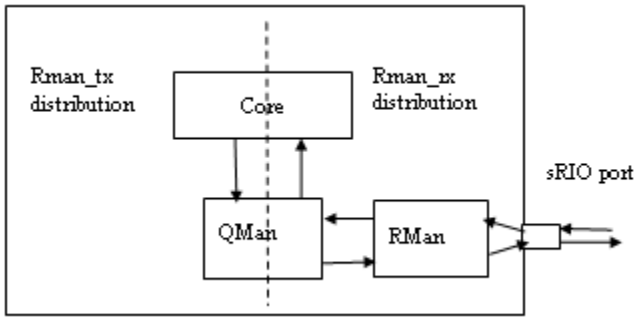
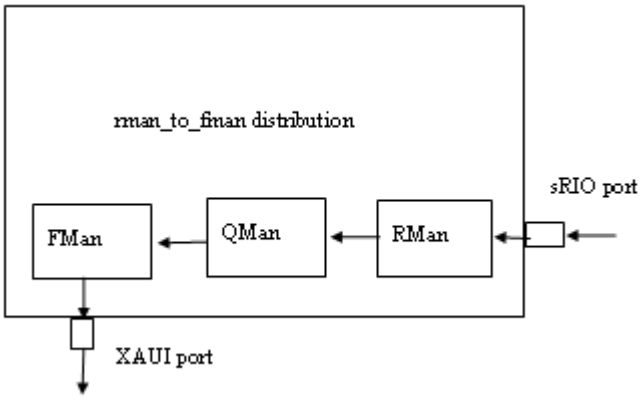
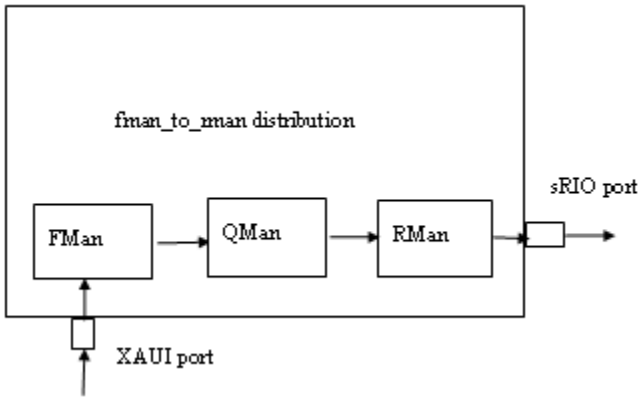
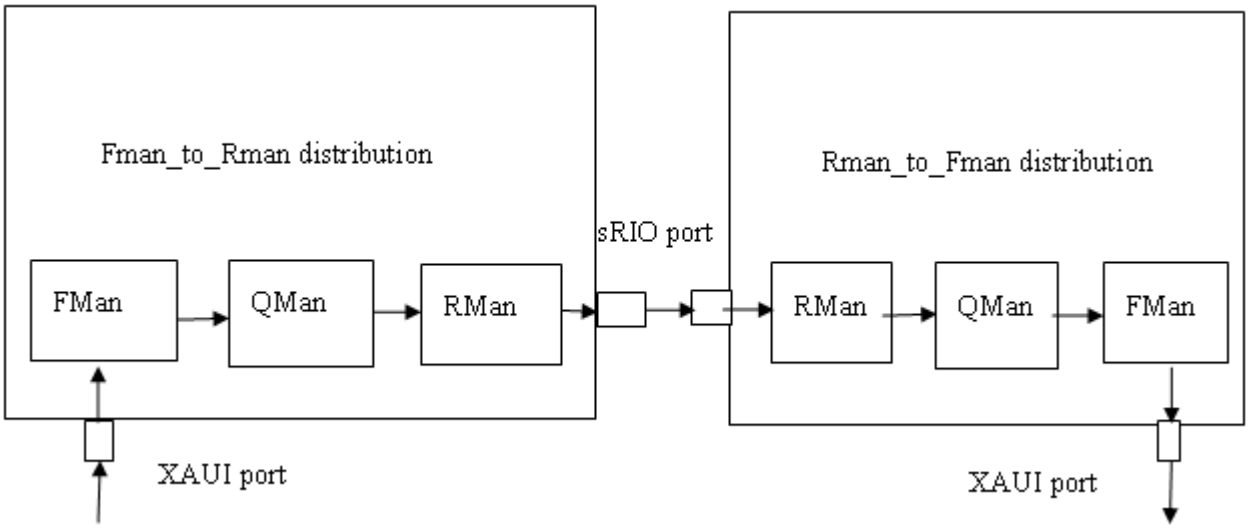
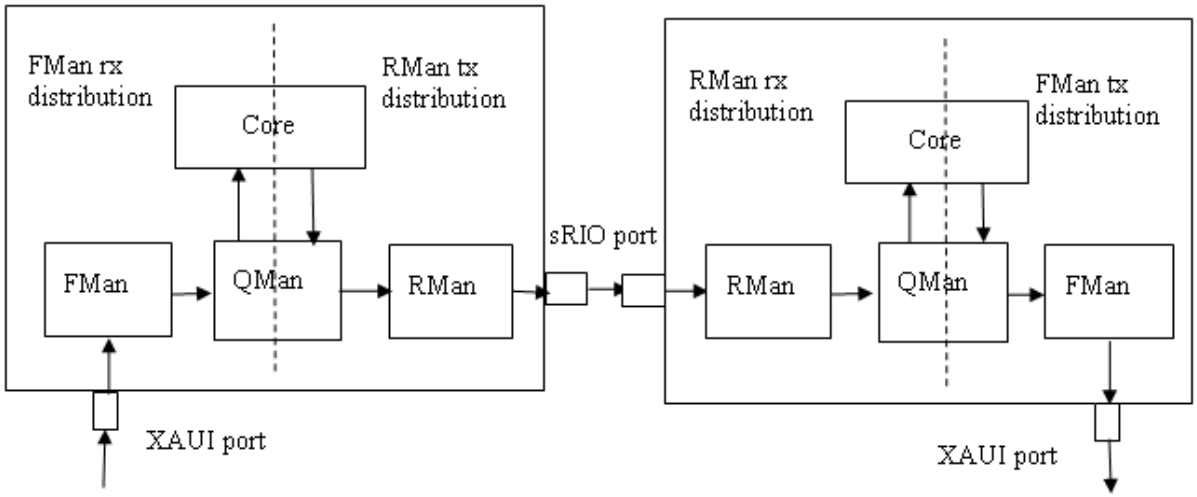
    <network_cfg>
        <defcfg file="/usr/etc/network_config.xml"/>
    </network_cfg>

    <trans_cfg>
        <defcfg file="/usr/etc/transactions_config.xml"/>
    </trans_cfg>

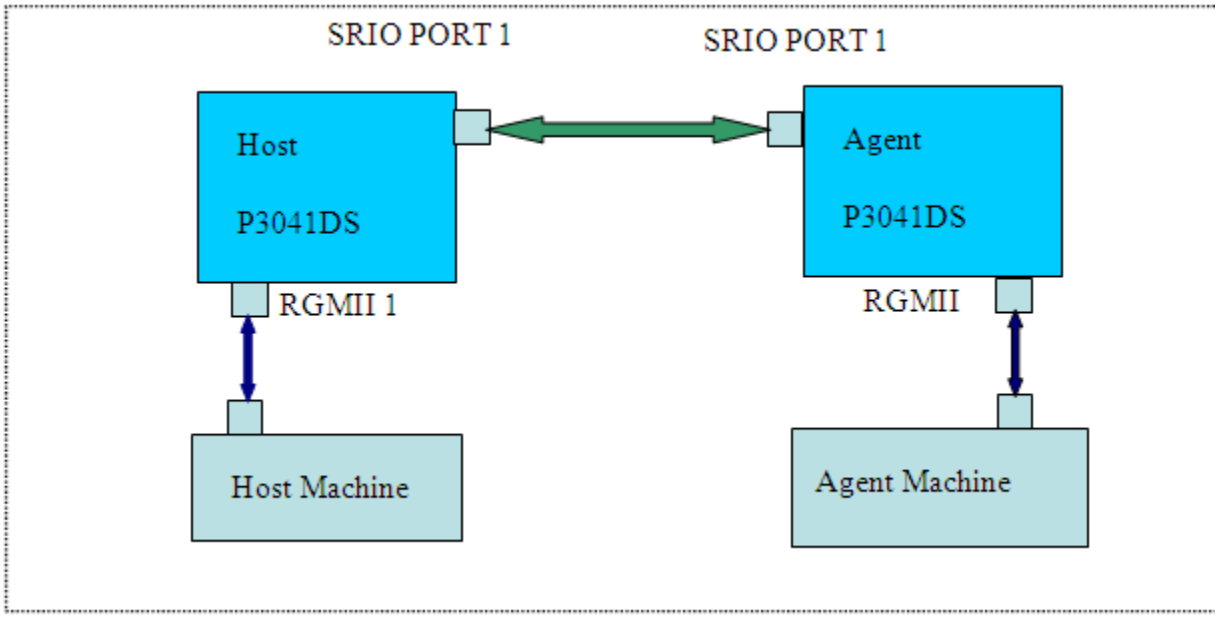
    <dists_cfg>
        <defcfg file="/usr/etc/distributions_config.xml"/>
    </dists_cfg>

    <policies_cfg>
        <defcfg file="/usr/etc/policies_config.xml"/>
        <policy name="processing2" enable="yes"/>
```

```
    </policies_cfg>
</fra_cfg>
```

This configuration file contains the following top elements:

- rman_cfg
- network_cfg
- trans_cfg
- dists_cfg
- policies_cfg

Each top element mentioned above may includes 'defcfg' element to include the default configuration file. Following 'defcfg', the different settings can be included to change the default value.

For example, in order to use sRIO port 2 to send messages, sRIO port number should be changed from default value 0 to 1. The configuration is as follows:

```
<dists_cfg>
    <defcfg file="/usr/etc/distributions_config.xml"/>

    <distribution name="fman_to_rman_dtsec0">
        <rio_port number="1"/>
    </distribution>
</dists_cfg>
```

9.9.14.2.1 Rman_cfg Element

This element is responsible for RMan general settings. It contains some child elements: fqbits, md_create, osid, bpid and sgfqid.

For example:

```
<rman_cfg>
        <fqbits type="Data-streaming" value="2"/>
        <fqbits type="Mailbox" value="2"/>
        <md_create mode="yes"/>

        <osid value="no"/>
        <bpid type="Data-streaming" value="11"/>
        <bpid type="Doorbell" value="10"/>
        <bpid type="Mailbox" value="11"/>
        <sgbpid value="12"/>
</rman_cfg>
```

OSID Element

Outbound Segmentation Interleaving Disable. Segmentation interleaving allows the message manager to interleave segments from two or more transactions with the same destination device ID (regardless of type). This may increase performance for transmission but may also consume additional reassembly resources at the destination.

A non-NXP target device may not complete Type 9 reassembly correctly when segmentation interleaving is performed on two PDUs with the same flow but different CoS. Workaround: disable segmentation interleaving by setting <osid value="yes">

Attributes:

value - (required) string, the valid value is "yes" or "no". "yes" means disable interleaving, "no" means enable interleaving.

EFQ Element

EFQ element is used to define error frame queue status. If enable EFQ, all the error information will be enqueued to a error frame queue.

Attributes:

value - (required) string, the valid value is "yes" or "no".

9.9.14.2.1.1 Fqbits Element

RMan supports algorithmic frame queue generation mode. This mode allows for steering transaction to a frame queue ID based on the RapidIO header attributes when wildcards are used with a classification rule. For The detailed information please refer to relevant manual. The current version of FRA application only support using ltr attribute of mailbox or stream ID attribute of data streaming transaction to dynamically generate the frame queue ID. The fqbits element indicates the number of rule mask bits to include in the frame queue ID.

Attributes:

value - (required) string; Defines the number of rule mask bits to include in the frame queue ID. For mailbox type 10 the valid value is 1 or 2. For data streaming is 1, 2, 3 or 4.

type - (required) string; Defines the transaction type, the valid value are 9 or 11.

9.9.14.2.1.2 Md_create Element

Md_create element is used to enable or disable RMan to write the inbound message descriptor. A performance improvement may be achieved by not writing the message descriptor.

Attributes:

mode - (required) string, the valid value is "yes" or "no". "yes" means do write the message descriptor, "no" means do not write the message descriptor.

9.9.14.2.1.3 BPID Element

BPID element is used to specify the buffer pool ID which RMan uses to store received transaction messages.

Attributes:

value - (required) string; Defines the buffer pool ID.

type - (required) string; Defines the transaction type, the valid value are 9-11.

9.9.14.2.1.4 SGBPID Element

SGBPID element is used to define the buffer pool ID which RMan uses to store the scatter/gather frame header.

Attributes:

value - (required) string; Defines the buffer pool ID.

9.9.14.2.2 Network_cfg Element

This element describes FMan ports that FRA will use. It has one child element: port

9.9.14.2.2.1 Port Element

Port Element is used to represent a FMan port.

Attributes:

name - (required) string: Defines the port name

type - (required) string: must be "MAC"

fm - (required) string: Define the FMan engine name, valid value is "0" or "1"

number - (required) string: Define the MAC index

9.9.14.2.3 Transaction Element

This element describes the RapidIO type 9/10/11 transaction information. Each transaction must be defined separately within its own elements. Each transaction is used for outbound and inbound operations.

Attributes:

name - (required) string; Defines a unique name for each transaction.

type - (required) string; Defines the transaction type, the valid value are Doorbell, Mailbox,Data-streaming.

9.9.14.2.3.1 Doorbell (type10) Transaction

A type10 outbound doorbell operation enables a producer to send a small amount of software-defined information across the interconnect fabric to a consumer's doorbell unit. It is the responsibility of the processor receiving the doorbell to determine the action to undertake. A small data payload of 2 bytes is used for doorbells.

This transaction has 1 child element: flowlvl

9.9.14.2.3.1.1 Flowlvl Element (type 10)

This element defines transaction flow level.

Attributes:

value - (required) string; Defines transaction flow level. 0: lowest flow level, 5: highest flow level

mask - (required) string; Defines flow conditional match.

0 Exact match on flow level

1 Match on less than or equal to flow level value

2 Match on greater than or equal to flow level value

3 Reserved

For example:

```
<transaction name="dbell-peer" type="Doorbell">
        <flowlvl value="5" mask="1"/>
</transaction>
```

9.9.14.2.3.2 Mailbox (type11) Transaction

A Type11 outbound RMan enables a producer to send a message across the interconnect fabric to a consumer's message hardware, called a mailbox. The receiving mailbox hardware places a message descriptor onto an inbound frame queue. A message may consist of one to sixteen segments. Messages can be queued for transmission in the producer's memory and the message hardware then processes them sequentially. Messages can also be queued in the consumer's memory while software processes them sequentially. The depths of the queues at the producer and consumer are configurable by software.

This transaction has 3 child elements: flowlvl, mbox, ltr, msglen.

9.9.14.2.3.2.1 Flowlvl Element (type 11)

This element defines transaction flow level.

Attributes:

value - (required) string; Defines transaction flow level. 0: lowest flow level, 5: highest flow level

mask - (required) string; Defines flow conditional match.

0 Exact match on flow level

1 Match on less than or equal to flow level value

2 Match on greater than or equal to flow level value

3 Reserved

9.9.14.2.3.2.2 Mbox Element (type 11)

This element defines mailbox field from packet header of mailbox transaction.

Attributes:

value - (required) string; Defines mailbox field, the valid value are 0,1,2,3

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

9.9.14.2.3.2.3 Ltr Element (type 11)

This element defines letter field from packet header of mailbox transaction. This field allows a sending processing element to concurrently send up to four messages to the same mailbox on the same processing element.

Attributes:

value - (required) string; Defines letter field. The valid value is 0-3.

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

9.9.14.2.3.2.4 Msglen Element (type 11)

This element defines message length field from packet header of mailbox transaction.

Attributes:

value - (required) string; Defines message length, and represents total number of packets comprising this message operation. The valid value is 0-15. A value of 0 indicates a single-packet message. A value of 15 (0xF) indicates a 16-packet message, etc.

mask - (required) string; Defines message length conditional match. Each bit set indicates don't care value for the inbound port-write header attribute.

0 Exact match on message length

1 Match on less than or equal to message length

2 Match on greater than or equal to message length

3 Reserved

For example:

```
<transaction name="mbox-10gec" type="Mailbox">
        <flowlvl value="0" mask="2"/>
        <mbox value="1" mask="0"/>
        <ltr value="0" mask="0" />
        <msglen value="6" mask="1"/>
</transaction>
```

9.9.14.2.3.3 Data streaming (type9) Transaction

A Type9 outbound segmentation operation enables a producer to send a packetized data unit (PDU) across the interconnect fabric to a consumer's reassembly hardware. The receiving reassembly hardware places the PDU descriptor onto an inbound frame queue. A PDU may consist of multiple segments supporting a total size of 64 Kbytes.

This transaction has 3 child elements: flowlvl, cos, streamid

9.9.14.2.3.3.1 Flowlvl Element (type 9)

This element defines transaction flow level.

Attributes:

value - (required) string; Defines transaction flow level. 0: lowest flow level, 5: highest flow level

mask - (required) string; Defines flow conditional match.

0 Exact match on flow level

1 Match on less than or equal to flow level value

2 Match on greater than or equal to flow level value

3 Reserved

9.9.14.2.3.3.2 CoS Element (type 9):

This element defines class-of-service field from the start or single packet header of data streaming transaction.

Attributes:

value - (required) string; Defines cos field, the valid value are 0-0xff

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

9.9.14.2.3.3.3 Streamid Element (type 9):

This element defines traffic stream identifier field from packet header of data streaming transaction. This is an end to end (producer to consumer) traffic stream identifier.

Attributes:

value - (required) string; Defines streamid field, the valid values is 0-0xffff

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

For example:

```
<transaction name="dstr-10gec" type="Data-streaming">
        <flowlvl value="0" mask="2"/>
        <cos value="15" mask="0"/>
        <streamid value="0" mask="0x1f"/>
</transaction>
```

9.9.14.2.4 Distribution Element

This element describes the approach of RMan/FMan processing message, it includes six ypes of distribution: RMAN_RX, RMAN_TX, FMAN_RX, FMAN_TX, RMAN_TO_FMAN, FMAN_TO_RMAN.

Attributes:

name - (required) string; Defines a unique name for each distribution.

type - (required) string; Defines the distribution type.

9.9.14.2.4.1 RMAN_RX Distribution

The RMAN_RX distribution describes RMan how to process the received message from RapidIO end point. Each enabled RMAN_RX distribution will be configured to a IBCU (Inbound Block Classification Unit). RMan has a total of 32 IBCU. If a

message matches an IBCU setting, it will be put to the corresponding queue. This type element contains four child elements: rio_port, sid, queue, transactionref

For example:

```
<distribution name="rman_to_dtsec0" type="RMAN_RX">
        <rio_port number="0" mask="1"/>
        <sid value="0" mask="0xff"/>
        <queue base="0x2500" mode="algorithmic" wq="0"/>
        <transactionref name="dstr-dtsec0"/>
</distribution>
```

9.9.14.2.4.1.1 RX Rio_port Element

This element defines the inbound messages come from which RapidIO port.

Attributes:

number - (required) string; Defines port number. The valid value is 1 or 2.

mask - (required) string; Defines bit mask for this field. If this value is 1, RMAN_RX/RMAN_TO_FMAN distribution will accepted messages from port 1 and port 2. This setting may be used for port 1 and port 2 loop-back mode.

9.9.14.2.4.1.2 RX Sid Element

This element defines the accepted messages sent by which device.

Attributes:

value - (required) string; Defines source device id,

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

9.9.14.2.4.1.3 RX Queue Element

This element defines the frame queue which the accepted messages will be put to. A frame queue is enqueued onto a Work Queue, and a work queue is grouped into a Channel. RMan supports direct mode and algorithmic mode to generate frame queue ID.

Attributes:

base - (required) string; Defines the basic queue id.

mode - (required) string; Defines the frame queue mode the valid value is "direct" or "algorithmic".

wq - (required) string; Defines the work queue

Transactionref element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

9.9.14.2.4.1.4 Transactionref Element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

9.9.14.2.4.2 RMAN_TX Distribution

This distribution describes RMan how to transmit a message. It contains four child elements: rio_port, did, queue, transactionref.

For example:

```
<distribution name="rman_to_peer_dtsec0" type="RMAN_TX">
        <rio_port number="0"/>
        <did value="1"/>
        <queue base="0x6000" count="4" wq="0"/>
        <transactionref name="dstr-dtsec0"/>
</distribution>
```

9.9.14.2.4.2.1 TX Rio_port Element:

This element defines RapidIO port number which will be used to send the messages.

Attributes:

number - (required) string; Defines RapidIO port number.

9.9.14.2.4.2.2 TX Did Element

This element defines device id that the message will send to.

Attributes:

value - (required) string; Defines destination device id.

9.9.14.2.4.2.3 TX Queue Element

This element defines the frame queue which the messages will be put to. The frame queue is enqueued onto a Work Queue. FRA supports up to 4 transmission frame queue.

Attributes:

base - (required) string; Defines the basic queue id.

count - (required) string; Defines the number of frame queue.

wq - (required) string; Defines the work queue

9.9.14.2.4.2.4 Transactionref Element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

9.9.14.2.4.3 FMAN_RX Distribution

FMAN_RX distribution describes FMan how to process the inbound message. This distribution contains two child elements: fman_port and queue.

For example:

```
<distribution name="dtsec0_to_rman" type="FMAN_RX">
        <fman_port name="dtsec0"/>
        <queue wq="4"/>
</distribution>
```

9.9.14.2.4.3.1 Fman_port Element

This element define the accepted messages come from which Fman port.

Attributes:

name - (required) string: Defines FMan port name. This is a reference to FMan port which defined in network_cfg element.

9.9.14.2.4.3.2 FMAN_RX Queue Element

This element defines the frame queue which the inbound messages will be put to. The FQID is defined in FMan policy file.

Attributes:
wq - (required) string; Defines the work queue.

9.9.14.2.4.4 FMAN_TX Distribution

FMAN_TX distribution describes FMan how to transmit the message.
This distribution contains two child elements: fman_port and queue.

For example:

```
<distribution name="dtsec0_to_network" type="FMAN_TX">
        <fman_port name="dtsec0"/>
        <queue count="2" wq="4"/>
</distribution>
```

9.9.14.2.4.4.1 Fman_port Element

This element define the accepted messages come from which Fman port.

Attributes:

name - (required) string: Defines FMan port name. This is a reference to FMan port which defined in network_cfg element.

9.9.14.2.4.4.2 [FMAN_TX Queue Element]

This element defines the frame queue which the messages will be put to. The frame queue is enqueued to a Work Queue. FRA supports up to 4 transmission frame queue.

Attributes:

count - (required) string; Defines the number of frame queue.

wq - (required) string; Defines the work queue

9.9.14.2.4.5 RMAN_TO_FMAN Distribution

The RMAN_TO_FMAN distribution describes RMan how to transfer the inbound message from RapidIO port to FMan port. It contains five child elements: rio_port, sid, queue, transactionref and fman_port

For example:

```
<distribution name="rman_to_fman0_dtsec0" type="RMAN_TO_FMAN">
       <rio_port number="0" mask="1"/>
       <sid value="0" mask="0xff"/>
       <queue base="0x1000" mode="algorithmic" wq="0"/>
       <transactionref name="mbox-dtsec0"/>
```

```
        <fman_port name="dtsec0"/>
</distribution>
```

9.9.14.2.4.5.1 RX Rio_port Element

This element defines the inbound messages come from which RapidIO port.

Attributes:

number - (required) string; Defines port number. The valid value is 1 or 2.

mask - (required) string; Defines bit mask for this field. If this value is 1, RMAN_RX/RMAN_TO_FMAN distribution will accepted messages from port 1 and port 2. This setting may be used for port 1 and port 2 loop-back mode.

9.9.14.2.4.5.2 RX Sid Element

This element defines the accepted messages sent by which device.

Attributes:

value - (required) string; Defines source device id,

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

9.9.14.2.4.5.3 RX Queue Element

This element defines the frame queue which the accepted messages will be put to. A frame queue is enqueued onto a Work Queue, and a work queue is grouped into a Channel. RMan supports direct mode and algorithmic mode to generate frame queue ID.

Attributes:

base - (required) string; Defines the basic queue id.

mode - (required) string; Defines the frame queue mode the valid value is "direct" or "algorithmic".

wq - (required) string; Defines the work queue

Transactionref element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

9.9.14.2.4.5.4 Transactionref Element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

9.9.14.2.4.5.5 Fman_port Element

This element define the accepted messages come from which Fman port.

Attributes:

name - (required) string: Defines FMan port name. This is a reference to FMan port which defined in network_cfg element.

9.9.14.2.4.6 FMAN_TO_RMAN Distribution

The FMAN_TO_RMAN distribution describes FMan how to transfer the inbound message from network port to RMan and RpiadIO port. It contains five child elements: rio_port, did, queue, transactionref and fman_port

For example:

```
<distribution name="fman_to_rman_dtsec0" type="FMAN_TO_RMAN">
        <fman_port name="dtsec0"/>
        <queue wq="0"/>
        <rio_port number="0"/>
        <did value="1"/>
        <transactionref name="mbox-dtsec0"/>
</distribution>
```

9.9.14.2.4.6.1 TX Rio_port Element:

This element defines RapidIO port number which will be used to send the messages.

Attributes:

number - (required) string; Defines RapidIO port number.

9.9.14.2.4.6.2 TX Did Element

This element defines device id that the message will send to.

Attributes:

value - (required) string; Defines destination device id.

9.9.14.2.4.6.3 TX Queue Element

This element defines the frame queue which the messages will be put to. The frame queue is enqueued onto a Work Queue. FRA supports up to 4 transmission frame queue.

Attributes:

base - (required) string; Defines the basic queue id.

count - (required) string; Defines the number of frame queue.

wq - (required) string; Defines the work queue

9.9.14.2.4.6.4 Transactionref Element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

9.9.14.2.4.6.5 Fman_port Element

This element define the accepted messages come from which Fman port.

Attributes:

name - (required) string: Defines FMan port name. This is a reference to FMan port which defined in network_cfg element.

9.9.14.2.5 Policy Element

This element describes the FRA's behavior. It was composed by one or more distribution order elements.

Attributes:

name - (required) string; Defines a unique name for each policy configuration.

enable - (required) string; Defines the policy status. The valide values are "yes" or "no"

9.9.14.2.5.1 Distribution Order Element

The dist_order element is a container for a list of distribution references. The distribution reference list contained within dist_order element is looked up sequentially and the first conforming record is the record to follow to. Thus, the order of distribution references is important.

9.9.14.2.5.1.1 Distributionref Element

The distributionref element refers to a distribution element by its name.

Attributes:

name - (required) string; Defines name of a distribution referred.

For example:

```
<policy name="processing1" enable="no">
        <!-- 10gec packets processing -->
        <dist_order>
                <distributionref name="10gec_to_rman"/>
                <distributionref name="rman_to_peer_10gec"/>
        </dist_order>
        <dist_order>
                <distributionref name="rman_to_10gec"/>
                <distributionref name="10gec_to_network"/>
        </dist_order>
</policy>
```

9.9.14.3 Revision History

Table 300. Revision History

Version	Updates
1.1	- Change distribution rx, tx, fwd to RMAN_RX, RMAN_TX, FMAN_RX, FMAN_TX respectively - Add network element to describe the FMan ports - Add FMAN_TO_RMAN and RMAN_TO_FMAN distribution to describe the messages transmission between FMan and RMan without core involvement.
1.0	Creation of FRA Configuration UM for SDK 1.1 release - RMan configuration introduction - RapidIO transaction configuration introduction - FRA distribution configuration introduction - FRA policy configuration introduction

9.9.15 NXP USDPAA FRA User Manual

9.9.15.1 Overview

This User Manual describes the NXP RMan Application (FRA) and explains how to install and configure and run FRA.

This Manual provides the following:

- A summary of the FRA application.
- Installing FRA based on SDK.
- Running and testing FRA steps

9.9.15.2 Introduction

9.9.15.2.1 Purpose

FRA NXP RMan Application is a Linux user space software to transfer packets from SRIO port to FMan port. This document describes the FRA installation, configuration, running and test.

9.9.15.2.2 Definitions and Acronyms

- FMan - Frame Manager
- BMan - Buffer Manager
- QMan - Queue Manager
- RMan - RapidIO Message Manager
- FRA - NXP RMan Application
- USDPAA - User Space Data Path Acceleration Architecture
- DPAA - Data path acceleration architecture

9.9.15.3 Overview of FRA

The USDPAA FRA application is a multi-thread USDPAA application that runs simultaneously on one or more boards which are connected with RapidIO cable. FRA application forwards IPv4 packets from one Ethernet interface to another boards using RapidIO transaction.

Overview of USDPAA

The NXP Data Path Acceleration Architecture comprises a set of hardware components which are integrated via a hardware queue manager and use a common hardware buffer manager. Software accesses the DPAA via hardware components called “software portals”. These directly provide queue and buffer manager operations such as enqueues, dequeues, buffer allocations, and buffer releases and indirectly provide access to all of the other DPAA hardware components via the queue manager. USDPAA is a software framework that permits Linux user space applications to directly access the DPAA queue and buffer manager software portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as RMan and FMan

FRA Architecture

FRA is based on USDPAA. But comparing the normal USDPAA application, in order to reuse the frame queue and buffer pool codes between processing network packets and RapidIO packets, FRA’s architecture is adjusted.



Figure 275. FRA Architeture

As Fig 1 shown, FRA is split to three layers, driver layer, library layer and application layer.

Driver Layer

Driver layer contains FMan, QMan, BMan, RMan, sRIO, DMA_mem, of, and process divers. All the driver code files are located in “usdpaa/drvier” folder, which are used by all the USDPAA applications. For FMan, QMan, BMan, DMA_mem, of, process driver information please refer to other USDPAA manuals.

sRIO driver

The serial RapidIO interface provides a RapidIO port to communicate with other RpaidIO devices. sRIO driver manages RapidIO ports hardware and provides some interfaces to configure port’s attributes and RapidIO protocol configuration such as “accept all”, “target id”, “traffic mode”. It also provides a few interface to handle interrupt events.

RMan driver

RapidIO message manager supports the message passing programming model for inter-processor and inter-device communication. RMan has a total of 32 inbound traffic classification units. A block is a group of eight classification units. The units are managed through a set of runtime registers. There are 4 outbound segmentation units. RMan driver is used to manage RMan global registers and inbound blocks. Users can call rman_dev_config() to configure global registers. This function supports setting inbound message descriptor write mode and the frame queue assembly rule of the doorbell mailbox and data-streaming transaction. RMan driver provides enable/disable/clear/status interfaces to handle interrupt and provides a few functions to request/release, enable/disable configure classification unit.

Library Layer

Library layer contains utils library, FMan port library, frame queue library, buffer pool library, RMan library, FRA configure parser library. All the libraries code files are located in “usdpaa/app/fra/lib” folder.

Utils Library

Utiles library is at the bottom of the layer. It provides the common interface for all the other libraries. At present, utils maily provides fra_dbg() function to print the debug information and cli_cmd() function to define a command.

Buffer Pool Library

Buffer pool library is intended to manage buffer pools. FRA supports up to 64 buffer pools. Each pool has buffer pool id, buffer number and buffer size three attributes which are grouped to a structure, as follows:

```
struct bpool_config {
       int bpid;
       uint32_t num;
       uint32_t sz;
};
```

Users can pass the array of the structure bpool_config to function bpools_init() to initialize a series of buffer pools. Buffer pool library will initialize the buffer pools and allocate DMA memory and release them to the pools. Buffer pool interface also provides acquire/free buffer functions to allocate and release buffers.

Frame queue Library

Frame queues that used by FMan and RMan can be separated into three types: nonpcd, pcd and tx.

Nonpcd frame queues are used to receive the error or complete status frame which are generally enqueued by FMan or RMan. Those frame queues normally has a higher priority and uses dynamically frame queue id, and only support QM_FQCTRL_CTXASTASHING option.

Pcd frame queues are used to store inbound data frame, such as network packets frame or RapidIO transaction message frames. Those frame queues support more control option such as QM_FQCTRL_AVOIDBLOCK, QM_FQCTRL_PREFERINCACHE and QM_FQCTRL_CTXASTASHING.

Tx frame queues are used to send outbound data frame which Fman or Rman dequeus and send out. These frame queues need to set contex A and B accroding hardware specifition.

This libary supports send/free frame via calling fra_send_frame() and fra_drop_frame() respectively.

FMan Port Library

Each FMan port is seperated into rx and tx functional modules. Rx module includes pcd rx sockets and nonpcd rx sockets. Pcd rx sockets is used to receive the Ipv4/Ipv6 real data frames. Nonpcd rx sockets is used to receive the frame that describe error status. The tx module is similar to rx module.

RMan Library

Rman library performs the iniliazation of the RMan and sRIO ports. There is a series of A interfaces provided to recieve and transmit messages.

The main interfaces as follows:
1. RMan rx API
RMan rx socket contains a set of frame queues to receive the RapidIO messages. Each RMan rx socket corresponds to a classification unit.
1.1 rman_rx_init
This function requests a RMan hardware resource-classification unit and then creates the rx frame queues. Returns the pointer of rman_rx on success or NULL on failure.
1.2 rman_rx_listen
Configure classification unit to receive the specific RapidIO messages which come from specified sid did and port.
1.3 rman_rx_enable
Enable rman_rx to receive messages.
1.4 rman_rx_disable

Stop rman_rx receiving messages, but don't release ibcu resource.
1.5 rman_rx_finish
Stop classification unit receiving, release rx frame queues ibcu resource and rman_rx socket.
1.6 rman_rx_get_fqs_num
RMan supports two mode direct and algorithmic to generate frame queue ID, if direct mode returns 1, if algorithmic mode, calculates and returns the number of frame queues according to algorithmic rule and transaction configuration.
1.7 rman_rx_get_ibcu
Return the classification unit index which is assigned to the specific RMan rx socket
1.8 rman_rx_get_opt
Structure hash_opt is used to describe which frame queue the received frame will be enqueued to Each RMan rx may include one or some rx frame queues, each rx frame queue has a hash opt attribute.
2. RMan tx API
RMan tx socket contains a set of frame queues to send the RapidIO messages using the same transaction.
2.1 rman_tx_init
Create tx frame queues and tx status frame queue
2.2 rman_tx_status_listen
Set rman_tx to receive the completed or/and error status frame
2.3 rman_tx_connect
Connect rman_tx socket to the destination device
2.4 rman_tx_finish
Release RMan tx and tx status frame queues.
2.5 opt_bindto_rman_tx
Bind the opt to the specific rman tx

FRA Configuration Parser Library

Fra configuration parser interface is to parse the fra configuration file which contains values of the RapidIO transaction header fields and distribution settings and policy setting.

Application Layer

FRA application is similar with normal USDPAA application. It also supports add/rm/list commands to manage thread, supports q or quit to exit. FRA adds status command to print the fra information including Rman configuration, RapidIO ports information and distributions information.
Fra application supports a few options defined by macro in fra_cfg.h file.
ENABLE_FRA_DEBUG - to print debug information
FRA_CORE_COPY_MD – to use processor to copy the RMan descriptor
FRA_MBOX_MULTICAST – to support multi-cast mode
FRA_VIRTUAL_MULTI_DID – to enable virtual multi-did.
The frame queue options are also defined in this files. For the detailed information please refer to USDPAA user manual.
This file also contains buffer pool settings.
bp 10 is used to store doorbell messages
bp 11 is used to store data-streaming/mailbox messages
bp 12 is used to store s/g tables.

```
#define DMA_MEM_BP4_BPID10
#define DMA_MEM_BP4_SIZE80
#define DMA_MEM_BP4_NUM0x100 /* 0x100*80==20480 (20KB) */
#define DMA_MEM_BP5_BPID11
#define DMA_MEM_BP5_SIZE1600
#define DMA_MEM_BP5_NUM0x2000 /* 0x2000*1600==13107200 (12.5M) */
#define DMA_MEM_BP6_BPID12
#define DMA_MEM_BP6_SIZE64
#define DMA_MEM_BP6_NUM0x2000 /* 0x2000*64==524288 (0.5MB) */
```

```
/* DMA memory size */
#define FRA_DMA_MAP_SIZE0x4000000 /* 64MB */
```

FRA Configuration

Fra configuration includes RMan global configuration, transaction settings distribution setting and policy setting. FRA Configuration User Manual detailedly describes each element

RMan Configuration

This element is responsible for RMan general settings. It contains a few child elements: fqbits and md_create and bpid settings.

For example:

```
<rman_cfg>
        <fqbits type="Data-streaming" value="2"/>
        <fqbits type="Mailbox" value="2"/>
        <md_create mode="yes"/>
        <bpid type="Data-streaming" value="11"/>
        <bpid type="Doorbell" value="10"/>
        <bpid type="Mailbox" value="11"/>
        <sgbpid value="12"/>
</rman_cfg>
```

Transaction Configuration

FRA configuration supports doorbell mailbox and data-streaming transaction. Almost all the values of the header fields of each transaction can be defined in this element.

For example:

```
<transaction name="dstr-10gec" type="Data-streaming">
        <flowlvl value="0" mask="2"/>
        <cos value="15" mask="0"/>
        <streamid value="0" mask="0x1f"/>
</transaction>
```

Distribution Configuration

This element describes the approach of processing message of RMan or FMan, it includes six types of distribution: rman_rx, rman_tx, fman_rx, fman_tx, fman_to_rman, rman_to_fman.

Policy Configuration

A distribution describes the one device module FMan or RMan how to process a series of specified packets. A distribution order element containing a sequence of distribution describes the board how to process specified packets. Policy element including one or more distribution order elements describes the one or two boards how to process all the packets. The typical dist order elements and the packets processing flows are as follows:

1. rman_rx and fman_tx distrubution forms a packets processing flow

```
<dist_order>
        <distributionref name=" rman_to_10gec "/>
        <distributionref name="10gec_to_network "/>
</dist_order>
```



Figure 276. RMan-core-FMan Processing Packet Flow

2. rman_rx and rman_tx distributions forms a loopback processing flow

```
<dist_order>
        <distributionref name="rman_to_10gec"/>
        <distributionref name="rman_to_peer_10gec"/>
</dist_order>
```



Figure 277. RMan-core-RMan Processing Packet Flow

3. rman_to_fman distrbution forms a packets processing flow without core involvement.

```
<dist_order>
        <distributionref name="rman_to_fman0_10gec"/>
</dist_order>
```



Figure 278. RMan-FMan Processing Packet Flow

4. fman_to_rman distrbution forms a packets processing flow without core involvement.

```
<dist_order>
        <distributionref name="fman_to_rman0_10gec"/>
</dist_order>
```



Figure 279. FMan-RMan Processing Packet Flow

Packet Flow

The FRA application can run on two boards at the same time, the two boards host and agent are connected with RapidIO cable. FRA first parses the configuration, and then creates threads to do packet processing. FRA provides two packets processing flows – processing1 and processing2.



Figure 280. Processing1 Packet Flow

As the Fig6 shown, the processing1 is that RMan directly enqueues the packets to FMan dedicated transmission channel, FMan directly enqueues the packets to RMan dedicated transmission channel, and the core does not need to participate in the processing.



Figure 281. Processing2 Packet Flow

As the Fig7 shown, the processing 2 is that FMan and RMan enqueues the packets to a frame queue of pool channel, and the core will dequeue and process the packets, and then enqueue them to the FMan or RMan dedicated transmission channel to send out.

FRA Configuration Files

FRA configuration files are located in “usdpaa/app/fra/app_config” folder. There are five files.
fra_config_dstr_processing1.xml – do packets processing1 using data-streaming transaction
fra_config_dstr_processing2.xml - do packets processing2 using data-streaming transaction
fra_config_mbox_processing1.xml - do packets processing1 using mailbox transaction
fra_config_mbox_processing2.xml - do packets processing1 using mailbox transaction

fra_config_dstr_port1_port2_loopback.xml – do packets loopback processing using data-streaming transaction

fra_config_port_write_test.xml - do Port-Write testing

Features

The FRA features are as follows:
• Supports all the network interfaces settings.
• Supports sRIO ports and RMan error interrupt handler.
• Supports two sRIO ports loopback mode on one board.
• Supports setting RMan writing the inbound message descriptor mode
• Supports Doorbell transaction with the following features:
—Supports setting flow level using configuration file
• Supports Mailbox transaction with the following features:
— Supports setting flow level, mailbox, letter, message length fields using configuration file
—Supports setting the frame queue mode “direct” or “algorithmic”.
—Supports multi-cast mode
• Supports Data streaming transaction with the following features:
— Supports setting flow level, cos, streamid fields using configuration file
—Supports setting the frame queue mode “direct” or “algorithmic”.
• Supports setting the behavior of each network port.

• Supports Port-Write transaction

9.9.15.4 Running FRA on Two Boards

9.9.15.4.1 Installation

The DPAA SDK has included the FRA code. So just need to install SDK.

9.9.15.4.2 Compilation

Please follow the SDK build guide to build the software. This will perform a default build of all files needed to boot Linux and run the FRA software. These files include dtb, FMan ucode image, u-boot image, kernel image, and rootfs image.

When build kernel, make sure these options are selected.

```
Device Drivers  --->
          <*> Userspace I/O drivers  --->
                    <*>   Freescale Serial RapidIO support
          [*] Staging drivers  --->
                    [*]     Freescale RapidIO Message Manager support
```

9.9.15.4.3 Configuring FRA

The FRA application uses three configuration files. One is to configure how FRA to process packets, the default name is fra_config_dstr_processing1.xml. Currently, default packet flow is processing 1. If you want to test the processing 2, you should run FRA with fra_config_dstr_processing2.xml configuration file. Similarly, if we want to use mailbox transaction to transmit the IP packets, using fra_config_mbox_processing1.xml or fra_config_mbox_processing2.xml. All the FRA configuration files located in /usr/etc folder. More detailed configuration refers to NXP FRA Configuration User's Manual.

The other two files are to configure FMD action, default names are usdpaa_config_p3_p5_serdes_0x33.xml and usdpaa_policy_hash_ipv4.xml. these two file we can refer to frame manager configuration manual.

If we want to specify the configuration file, use the following command on the both boards:

```
# fra -c usdpaa_config_p3_p5_serdes_0x33.xml -p usdpaa_policy_hash_ipv4.xml -f
fra_config_dstr_processing1.xml
```

9.9.15.4.3.1 Selecting Ethernet interfaces for FRA

The following device tree snippet shows a Linux private interface and also an interface used privately by FRA.

```
ethernet@0 {
        compatible = "fsl,p3041-dpa-ethernet-init", "fsl,dpa-ethernet-init";
        fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
        fsl,qman-frame-queues-rx = <0x50 1 0x51 1>;
        fsl,qman-frame-queues-tx = <0x70 1 0x71 1>;

};
thernet@1 {
        compatible = "fsl,p3041-dpa-ethernet", "fsl,dpa-ethernet";
        fsl,fman-mac = <&enet0>;

};
```

The first Ethernet is used by FRA. The second is used by the Linux Ethernet driver.

The following list shows the correspondence between Ethernets in the device tree and physical Ethernet MACs on FMan hardware instances on P3041.

Table 301. Correspondence between Ethernets in DTS and physical Ethernet

SW Interface #	FMAN1
0	DTSEC 0
1	DTSEC 1
2	DTSEC 2
3	DTSEC 3
4	DTSEC 4
5	10GEC

9.9.15.4.4 Prepare the Hardware

The FRA application needs two PC or one PC with multiple Ethernet interfaces, two p3041DS or P5020DS boards, when using 0x33 RCW each board should been inserted a RapidIO card to slot6. Check board's switch, sw2 is 0b00100001î(slot7->PEX1, slot6->SRIO, slot4->PEX3, slot2->XAUI), Sw5 is 0b00010100î (Serdes reference clock for bank1 is 100MHz, bank2 is 125MHz bank3 is 125MHz). For P5020DS board sw2 is also 0b00100001. For P2041RDB, using 0x02 RCW, RapidIO card should be inserted to slot1. and the following command should be run

```
cpld lane_mux 6 0;
cpld lane_mux a 0;
cpld lane_mux c 0;
cpld lane_mux d 0;
cpld reset altbank;
```

FRA supports 15G network interfaces. But 0x33 RCW only provides 12G network interfaces: RGMII1 (DTSEC3), RGMII2 (DTSEC4) and XAUI. As the Fig8 shown, we can use DTSEC3 to receive and transmit the IP packets. The host machine should connect to host board RGMII1 port, and agent machine should connect to agent board RGMII 1 port too.



Figure 282. Serdes 0x33 protocol hardware setup

9.9.15.4.5 Managing RCW and U-boot Image

The RCW, FMan microcode, and u-boot binary files must be programmed into the P3041DS NOR-flash We can use Processor Expert (PEx) to create serdes 0x33 rcw file and then convert to rcw image.

```
serdes 0x33 RCW of P3041DS is :
00000000: AA55 AA55 010E 0100 1260 0000 0000 0000
00000010: 241C 0000 0000 0000 CC98 4A00 0300 2000
00000020: FE80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 CF8E 2F88

serdes 0x33 RCW of P5020DS is :
00000000: AA55 AA55 010E 0100 0C54 0000 0000 0000
00000010: 1E12 0000 0000 0000 CC98 4A00 0300 2000
00000020: FE80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 1AC8 E13F


serdes 0x02 RCW of P2041RDB is:
00000000: AA55 AA55 010E 0100 1260 0000 0000 0000
00000010: 241C 0000 0000 0000 0899 30C0 C7C0 2000
00000020: FE80 0000 4000 0000 0000 0000 0000 0000
00000030: 0000 0000 D003 0F07 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 57B0 F7D9
```

Convert rcw xxd format to image

```
$ xxd -r rcw.xxd > rcw-0x33.bin
```

The P3041DS divides the flash into multiple banks. This permits multiple copies of RCW and u-boot files to be kept in flash. It is not required, but NXP suggests leaving a functional u-boot in bank0 (which is used when the board is powered on) and program new rcw and u-boot files into bank 4. This method ensures that a working u-boot (in bank 0) is retained in case of a mistake in programming bank 4. To program bank 4, first do a power-on reset to boot from bank 0. Use the u-boot in bank 0 to program the files into bank 4. Networking parameters need to be set for the u-boot in bank 0 so that it can tftp. U-boot uses environment variables to control network settings. Variables are given values with the ìsetenvî command. The choices for these settings must match your local network. Note in particular serverip. A machine with that IP address is assumed to be running a tftp server containing the files to be transferred via tftp. The major U-boot networking variables are:

```
 => setenv ethact <active Ethernet interface for u-boot use. Should be FM1@DTSEC5 if using the
recommended SerDes protocol number 0xe>
=> setenv ethaddr <MAC address 0>
=> setenv ethNaddr <MAC address N=1 to 5>
...
=> setenv ipaddr <your ip address>
=> setenv netmask <your netmask>
=> setenv gatewayip <your default gateway ip address>
=> setenv serverip <the address of your TFTP server>
=> sav
```

The following instructions show how to reprogram bank 4. Remember to do this only when booted from bank 0.

```
 # Be booted from bank 0!!!!
=> tftp 0x02000000 u-boot.bin
=> protect off 0xebf80000 +$filesize
=> erase 0xebf80000 +$filesize
=> cp.b 0x02000000 0xebf80000 $filesize
=> tftp 0x02000000 rcw_0x33.bin
=> protect off 0xec000000 +$filesize
=> erase 0xec000000 +$filesize
=> cp.b 0x02000000 0xec000000 $filesize
 => tftp 0x02000000 fsl_fman_ucode.bin
=> protect off 0xeb000000 +$filesize
 => erase 0xeb000000 +$filesize
=> cp.b 0x02000000 0xeb000000 $filesize
 # Reset to bank 4:
=>pix altbank
```

U-boot in bank 4 will then run and one can proceed to boot Linux and run the FRA software.

9.9.15.4.6 Booting Linux

Set following environment settings on board's bank4, Setup networking variables for tftp in bank4:

```
=> setenv ethact <active Ethernet interface for u-boot use>
=> setenv ethaddr <MAC address 0>
=> setenv ethNaddr <MAC address N=1 to 5>
...
=> setenv ipaddr <your ip address>
=> setenv netmask <your netmask>
=> setenv gatewayip <your default gateway ip address >
=> setenv serverip <the address of your TFTP server>
=> sav
```

Setup and save other uboot variables to tftp the USDPAA device tree, root file system and Linux image and boot Linux.

```
=>setenv bootargs "root=/dev/ram rw console=ttyS0,115200"
=>setenv fraboot "tftp 1000000 uImage-p3041ds.bin; tftp 2000000 fsl-image-core-p3041ds.ext2.gz.u-
```

```
boot;  tftp c00000 uImage-p3041ds-usdpaa.dtb; bootm 1000000 2000000 c00000"
=>run fraboot
```

Following are some booting information points:

u-boot boot log:

```
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 1024 KB enabled
SRIO1: enabled
SRIO2: disabled
NAND:  1024 MiB
MMC:  FSL_ESDHC: 0
```

Linux kernel boot log:

```
fsl-of-srio ffe0c0000.rapidio: Rapidio UIO driver initialized
...
fsl-of-rman ffe1e0000.rman: Of-device full name /soc@ffe000000/rman@1e0000 initialized
fsl-of-rman ffe1e0000.rman: RMan inbound block0 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block1 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block2 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block3 initialized.
```

9.9.15.4.7 Running FRA

Logging into Linux

At the Linux prompt, login as "root".

```
Yocto (Built by Poky 6.0) 1.1 p3041ds ttyS0

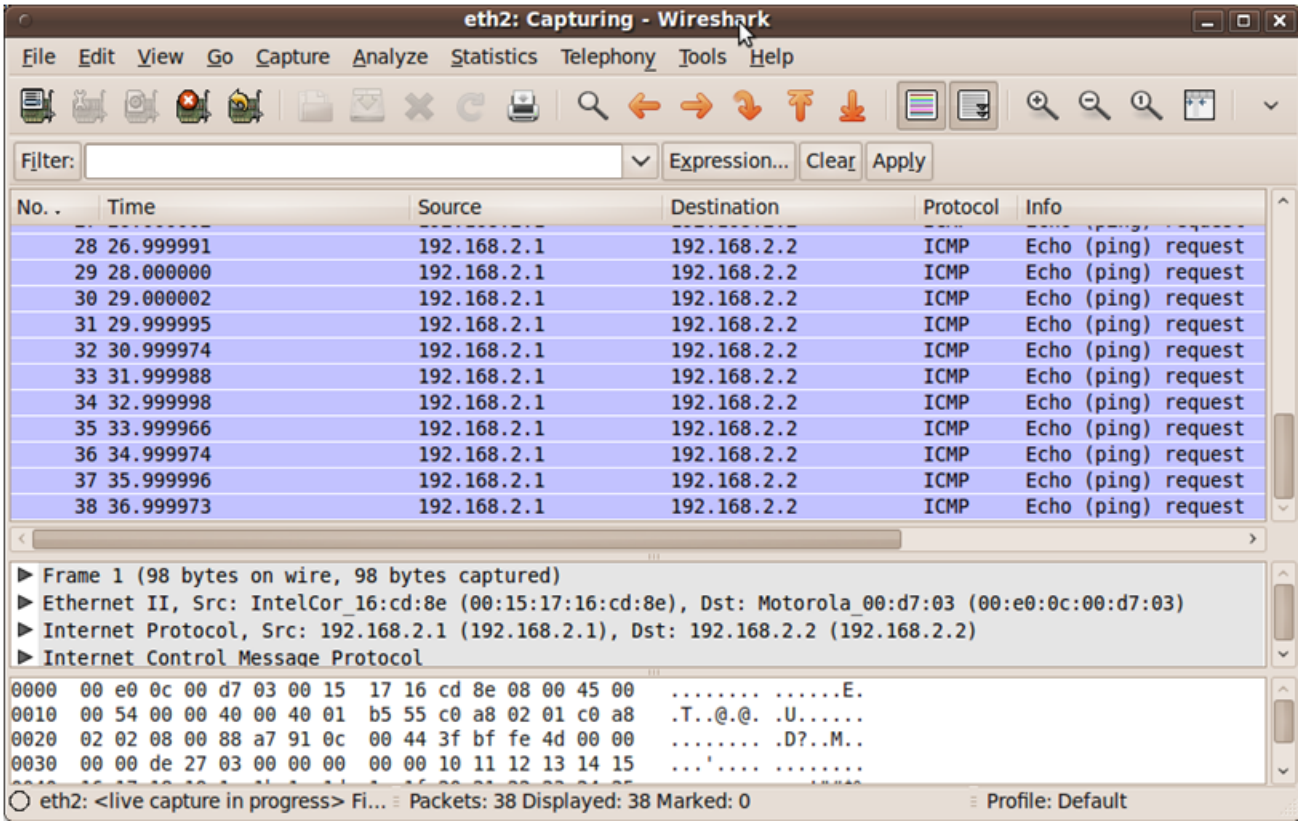
p3041ds login: root
root@p3041ds:~#
```

Configure FMan PCD using fmc with the XML files in /usr/etc:

```
[root@p3041 root]# cd /usr/etc
[root@p3041 root]# fmc -c usdpaa_config_p3_p5_serdes_0x33.xml -p usdpaa_policy_hash_ipv4.xml -a
```

Run the FRA application:

```
root@p3041ds:~# fra
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 47, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@3, Tx Channel = 44, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@4, Tx Channel = 45, FMAN = 0, Port ID = 4
Found /fsl,dpaa/ethernet@5, Tx Channel = 40, FMAN = 0, Port ID = 0
Configuring for 3 network interfaces
fra: BPOOL: Release 8192 bufs to BPID 9
fra: BPOOL: Release 256 bufs to BPID 10
fra: BPOOL: Release 8192 bufs to BPID 11
fra: BPOOL: Release 8192 bufs to BPID 12
fra: RMan inbound block0 initialized
fra: RMan inbound block1 initialized
fra: RMan inbound block2 initialized
fra: RMan inbound block3 initialized
fra: can not find fman port dtsec0
fra: can not find fman port dtsec1
```

```
fra: can not find fman port dtsec2
Start dist(rman_to_fman0_10gec)
Start dist(fman_to_rman_10gec)
Start dist(rman_to_fman0_dtsec4)
Start dist(fman_to_rman_dtsec4)
Start dist(rman_to_fman0_dtsec3)
Start dist(fman_to_rman_dtsec3)
Thread uid:0 alive (on cpu 1)
fra>
```

Additional FRA threads can be started on other CPUs by entering commands at the FRA command prompt. Running threads can also be queried and the application can also be shutdown.

• add a FRA thread on a single cpu (e.g. cpu 2)

```
fra> add 2
```

• add FRA threads on a range of cpus

```
fra> add 2..3
```

• list the cpus currently enabled (by querying them, i.e. this also verifies that they aren't blocked)

```
fra> list
```

Thread alive on cpu 1

Thread alive on cpu 2

Thread alive on cpu 3

• display FRA configuration and status

```
fra> status
RMan configuration:
Create inbound message descriptor: yes
The algorithmic frame queue bits info:
data streaming:2 mailbox:2
BPID info:
data streaming:11 mailbox:11 doorbell:10 sg:12
Use SRIO port 0: using lane 0
Create 3 RX sockets and 12 frame queues
Create 3 TX sockets and 96 frame queues
distribution order-1:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_10gec
rio port:0 - 10gec
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-10gec type:Data-streaming
base FQID:0x1500 count:4 configured to IBCU 0

distribution order-2:
distribution-1-FMAN_TO_RMAN: fman_to_rman_10gec
FMan:10gec - rio port 0
did:1
rio_tran:dstr-10gec type:Data-streaming

distribution order-3:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_dtsec4
rio port:0 - dtsec4
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-dtsec4 type:Data-streaming
```

```
base FQID:0x1400 count:4 configured to IBCU 1

distribution order-4:
distribution-1-FMAN_TO_RMAN: fman_to_rman_dtsec4
FMan:dtsec4 - rio port 0
did:1
rio_tran:dstr-dtsec4 type:Data-streaming

distribution order-5:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_dtsec3
rio port:0 - dtsec3
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-dtsec3 type:Data-streaming
base FQID:0x1300 count:4 configured to IBCU 2

distribution order-6:
distribution-1-FMAN_TO_RMAN: fman_to_rman_dtsec3
FMan:dtsec3 - rio port 0
did:1
rio_tran:dstr-dtsec3 type:Data-streaming
fra>
```

• Enable or disable debug information output

Note: this function is effective only when compiling with ENABLE_FRA_DEBUG definition

```
fra> debug [on/off]
```

• If you want to quit FRA , should use the following command, FRA will release the related resource.

```
fra> q
```

9.9.15.4.8 Testing FRA

In order to test FRA, we need to send IP packets to host board's Ethernet interface, since FRA does not support ARP protocol, we should do IP and MAC binding on host PC. Following are the test FRA steps on P3041DS boards.

1. Setup the hardware as Fig9 and run the FRA application on both host and agent as described in previous section.
2. Find the host board's Ethernet interface Mac address

   This case, FRA will use the port 3 to receive the IP packets; on host board console Linux startup log has the MAC information:

   ```
   cpu0: fsl_mac: FSL FMan MAC API based driver ()
   cpu0: fsl_mac: ffe4e6000.ethernet: FMan dTSEC version: 0x08240101
   cpu0: fsl_mac: ffe4e6000.ethernet: FMan MAC address: 00:e0:0c:00:d7:03
   cpu0: fsl_mac: ffe4e8000.ethernet: FMan dTSEC version: 0x08240101
   cpu0: fsl_mac: ffe4e8000.ethernet: FMan MAC address: 00:e0:0c:00:d7:04
   cpu0/0: Applying 10G tx-ecc error workaround (10GMAC-A004) ...
   cpu0/0: done.
   cpu0: fsl_mac: ffe4f0000.ethernet: FMan XGEC version: 0x00010330
   cpu0: fsl_mac: ffe4f0000.ethernet: FMan MAC address: 00:e0:0c:00:d1:05
   ```

   So we find DTSEC3's Mac address is 00:e0:0c:00:d7:03.
3. If we has configure the Ethernet card IP address 192.168.2.1, so on host machine create a ARP entry in subnet range of the host pc network port connected to host board's DTSEC3, by using the following command:

   ```
   $ sudo arp -s 192.168.2.2 00:e0:0c:00:d7:03
   ```

4. On host machine ping host board's DTSEC3

```
$ ping 192.168.2.2
```

5. Run Wireshark on agent machine and to capture the network port connected to the agent board's DTSEC3 network port.

```
$ sudo wireshark
```

As the figure below shows, we should see the ping packets sent from host machine.



Figure 283. Wireshark Capturing Packets

9.9.15.4.9 Debugging FRA

We may need to debug FRA with detailed output log. FRA provides debug command to enable or disable those debug information output. In order to support debug mode, we should insert the following sentence to fra.h file line 46 and then re-compile.

```
#define ENABLE_FRA_DEBUG
```

Use the following command to enable or disable the debug information output.

```
fra> debug [on/off]
```

9.9.15.4.10 Testing Port Write

A Type8 port-write is intended as an error reporting mechanism from an end point-less device to a control processor or other system host. The RMan also supports generation of outbound port-writes for testability and debug. Inbound port-writes have dedicated hardware for guaranteed delivery.

Run the FRA application with Port-Write test configuration file

```
root@t4240qds:~# fra -r -f fra_config_port_write_test.xml
    fra: BPOOL: Release 8192 bufs to BPID 9
    fra: BPOOL: Release 256 bufs to BPID 10
    fra: BPOOL: Release 8192 bufs to BPID 11
    fra: BPOOL: Release 8192 bufs to BPID 12
    Start dist(pw_to_peer)
    fra: RMan inbound block0 is initialized
    Start dist(pw_from_peer)
    Thread uid:0 alive (on cpu 1)
    fra>
```

Send Port-Write message using "pw" command

```
fra> pw 1234
```

Receive the Port-Write message on the peer

```
 fra> Port Write: get 4 bytes data:1234
```

• Note: RapidIO supports a maintenance port-write type that contains a data payload 4 8 16 32 or 64 bytes.

9.9.15.5 Running FRA with flow control

The chapter is intended to introduce how to run FRA with flow control.

The operations of installation and testing are similar to Chapter 3, So those sections please refer to above description.

9.9.15.5.1 Booting Linux

Set following environment settings on board's bank4, Setup networking variables for tftp in bank4:

```
=> setenv ethact <active Ethernet interface for u-boot use>
=> setenv ethaddr <MAC address 0>
=> setenv ethNaddr <MAC address N=1 to 5>
...
=> setenv ipaddr <your ip address>
=> setenv netmask <your netmask>
=> setenv gatewayip <your default gateway ip address >
=> setenv serverip <the address of your TFTP server>
=> sav
```

Setup srio device ID on left board.

```
=> mm 0xffe0d0100
fe0d0100: 00000000 ? 808b0000
fe0d0104: 00000000 ? q
=>
```

Setup srio device ID on right board.

```
=> mm 0xffe0d0100
fe0d0100: 00000000 ? 808d0000
fe0d0104: 00000000 ? q
=>
```

Setup and save other uboot variables to tftp the USDPAA device tree, root file system and Linux image and boot Linux.

```
=>setenv bootargs "root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=0x4000000"
=>setenv fraboot "tftp 1000000 uImage-t4240qds.bin; tftp 2000000 fsl-image-core-
t4240qds.ext2.gz.u-boot;  tftp c00000 uImage-t4240qds-usdpaa.dtb; bootm 1000000 2000000 c00000"
=>run fraboot
```

Following are some booting information points:

u-boot boot log:

```
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 1024 KB enabled
SRIO1: enabled
SRIO2: disabled
NAND:  1024 MiB
MMC:  FSL_ESDHC: 0
```

Linux kernel boot log:

```
fsl-of-srio ffe0c0000.rapidio: Rapidio UIO driver initialized
...
fsl-of-rman ffe1e0000.rman: Of-device full name /soc@ffe000000/rman@1e0000 initialized
fsl-of-rman ffe1e0000.rman: RMan inbound block0 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block1 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block2 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block3 initialized.
```

9.9.15.5.2 Compilation FRA with flow control

Please follow the SDK build guide to build the software. This will perform a default build of all files needed to boot Linux and run the FRA software. These files include dtb, FMan ucode image, u-boot image, kernel image, and rootfs image.

When build kernel, make sure these options are selected.

```
Device Drivers  --->
          <*> Userspace I/O drivers  --->
                    <*>   Freescale Serial RapidIO support
          [*] Staging drivers  --->
                    [*]     Freescale RapidIO Message Manager support
```

When build rootfs, make sure to define FRA_FC in fra_cfg.h.

```
#define FRA_FC
```

9.9.15.5.3 Running FRA

Logging into Linux

At the Linux prompt, login as "root".

```
Yocto (Built by Poky 6.0) 1.1 t4240qds ttyS0

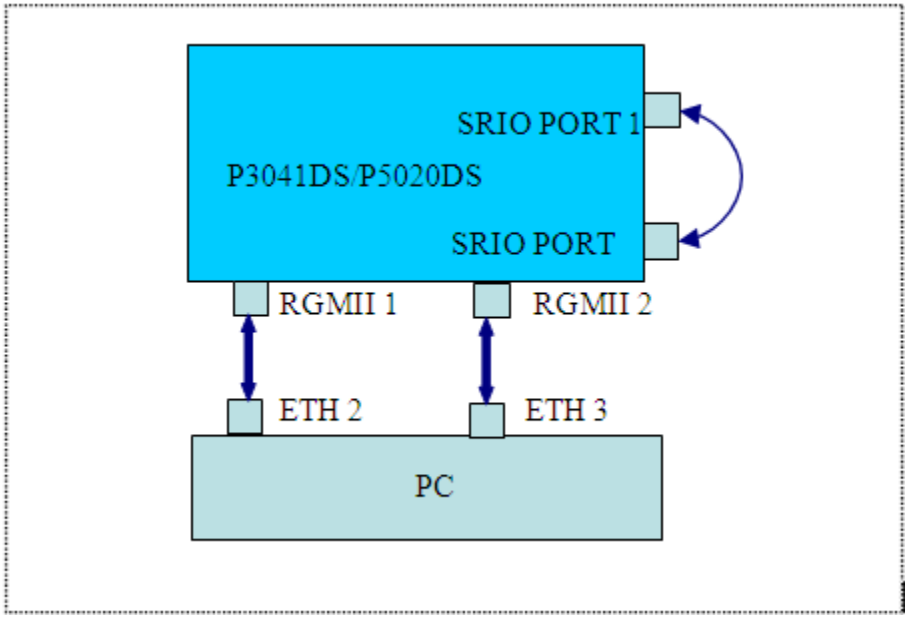
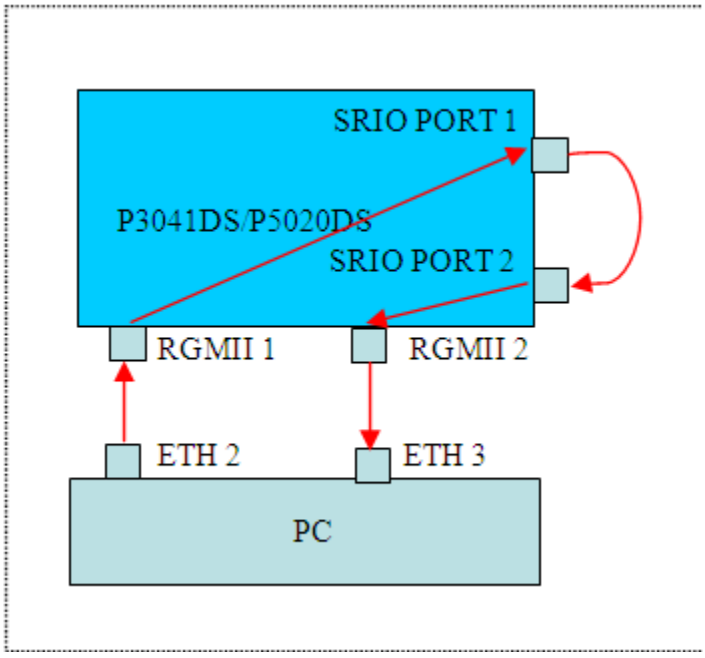
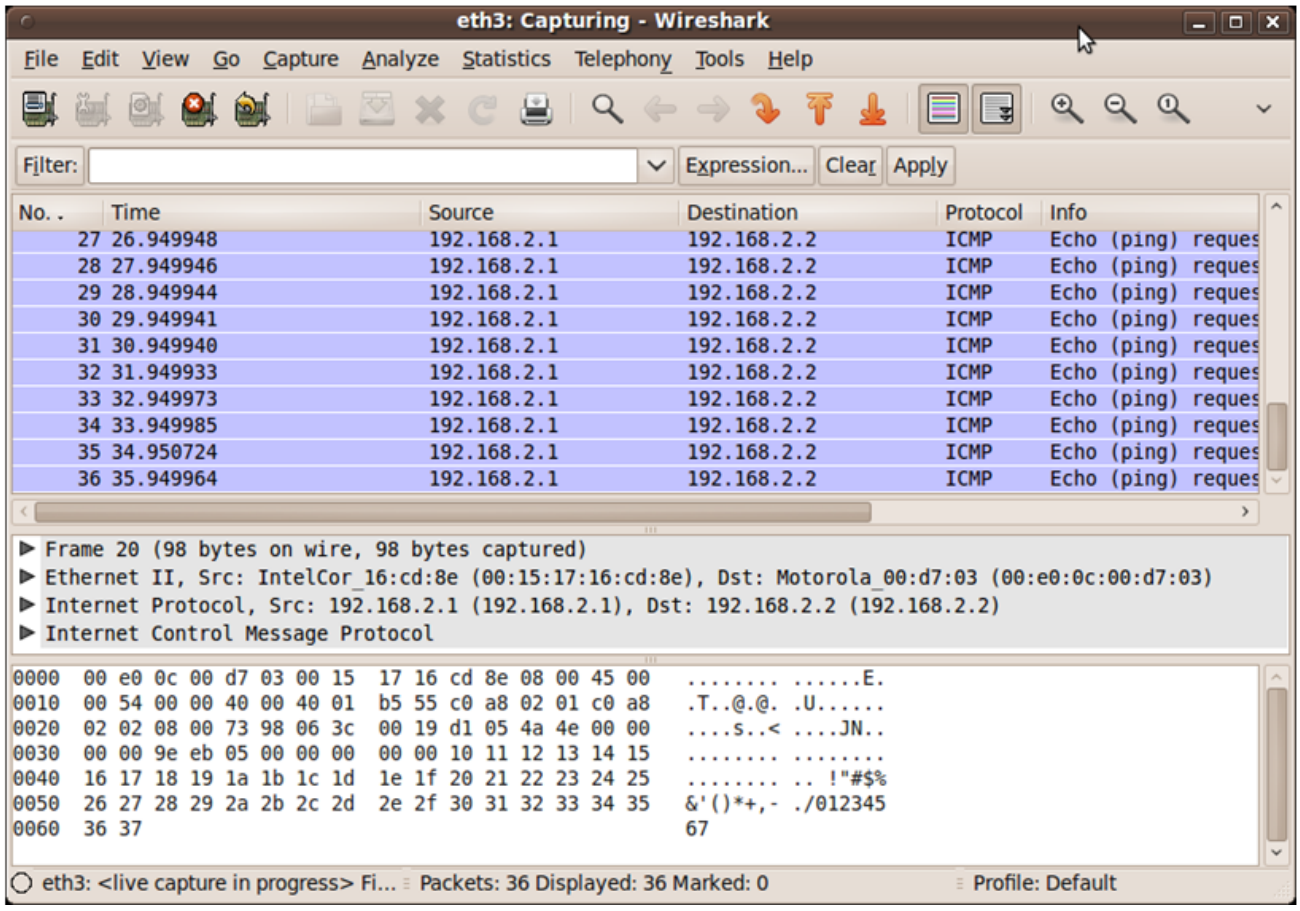
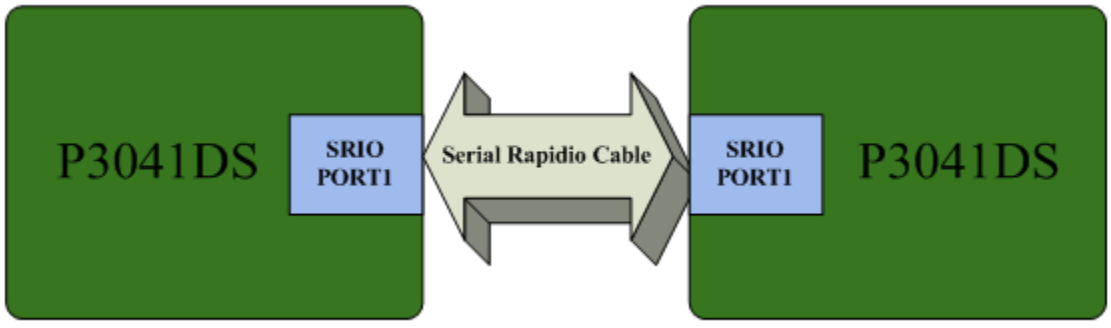
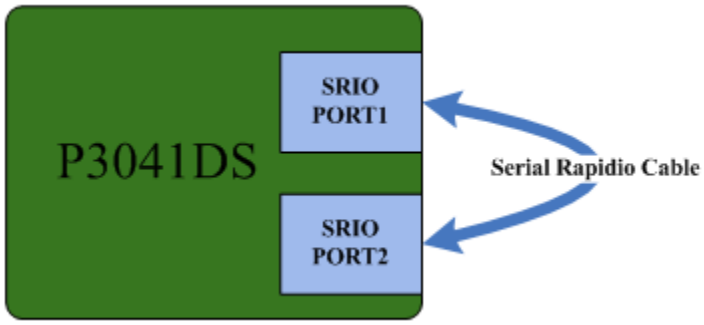
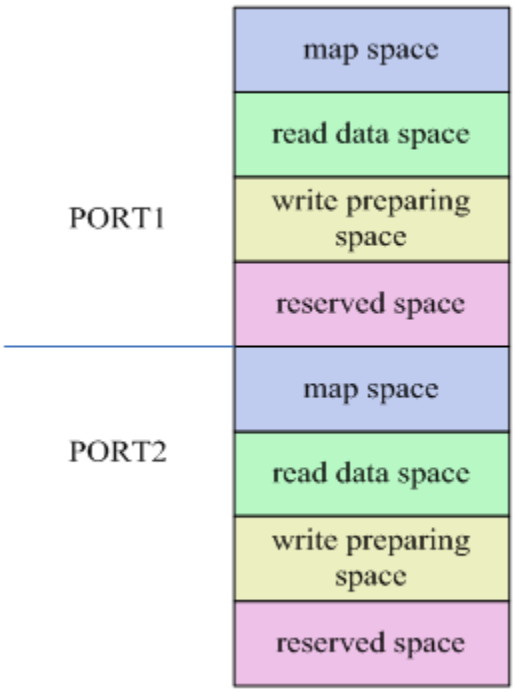
t4240qds login: root
root@t4240qds:~#
```

Configure FMan PCD using fmc with the XML files in /usr/etc:

```
[root@t4240 root]# cd /usr/etc
[root@t4240 root]# fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p fra_dstr_fc_policy.xml -a
```

Run the FRA application on left board:

```
root@t4240qds:~# fra -c usdpaa_config_t4_serdes_1_1_6_6.xml -p fra_dstr_fc_policy.xml -f
fra_config_dstr_fc_processing1_left.xml
```

Run the FRA application on right board:

```
root@t4240qds:~# fra -c usdpaa_config_t4_serdes_1_1_6_6.xml -p fra_dstr_fc_policy.xml -f
fra_config_dstr_fc_processing1_right.xml
```

Additional FRA threads can be started on other CPUs by entering commands at the FRA command prompt. Running threads can also be queried and the application can also be shutdown.

• add a FRA thread on a single cpu (e.g. cpu 2)

```
fra> add 2
```

• add FRA threads on a range of cpus

```
fra> add 2..3
```

• list the cpus currently enabled (by querying them, i.e. this also verifies that they aren't blocked)

```
fra> list
Thread alive on cpu 1
Thread alive on cpu 2
Thread alive on cpu 3
```

• display FRA congestion group status

```
fra> cgr
```

• display FRA configuration and status

```
fra> status
RMan configuration:
Create inbound message descriptor: yes
The algorithmic frame queue bits info:
data streaming:2 mailbox:2
BPID info:
data streaming:11 mailbox:11 doorbell:10 sg:12
Use SRIO port 0: using lane 0
Create 3 RX sockets and 12 frame queues
Create 3 TX sockets and 96 frame queues
distribution order-1:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_10gec
rio port:0 - 10gec
```

```
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-10gec type:Data-streaming
base FQID:0x1500 count:4 configured to IBCU 0

distribution order-2:
distribution-1-FMAN_TO_RMAN: fman_to_rman_10gec
FMan:10gec - rio port 0
did:1
rio_tran:dstr-10gec type:Data-streaming

distribution order-3:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_dtsec4
rio port:0 - dtsec4
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-dtsec4 type:Data-streaming
base FQID:0x1400 count:4 configured to IBCU 1

distribution order-4:
distribution-1-FMAN_TO_RMAN: fman_to_rman_dtsec4
FMan:dtsec4 - rio port 0
did:1
rio_tran:dstr-dtsec4 type:Data-streaming

distribution order-5:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_dtsec3
rio port:0 - dtsec3
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-dtsec3 type:Data-streaming
base FQID:0x1300 count:4 configured to IBCU 2

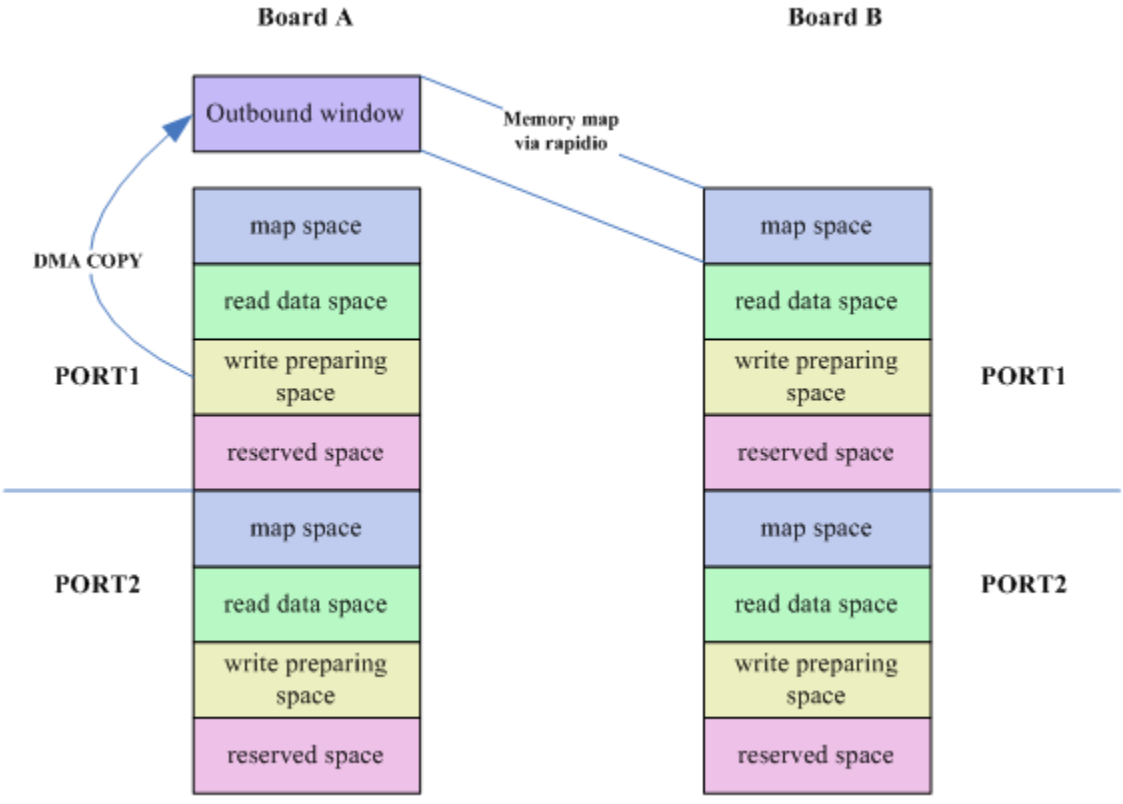
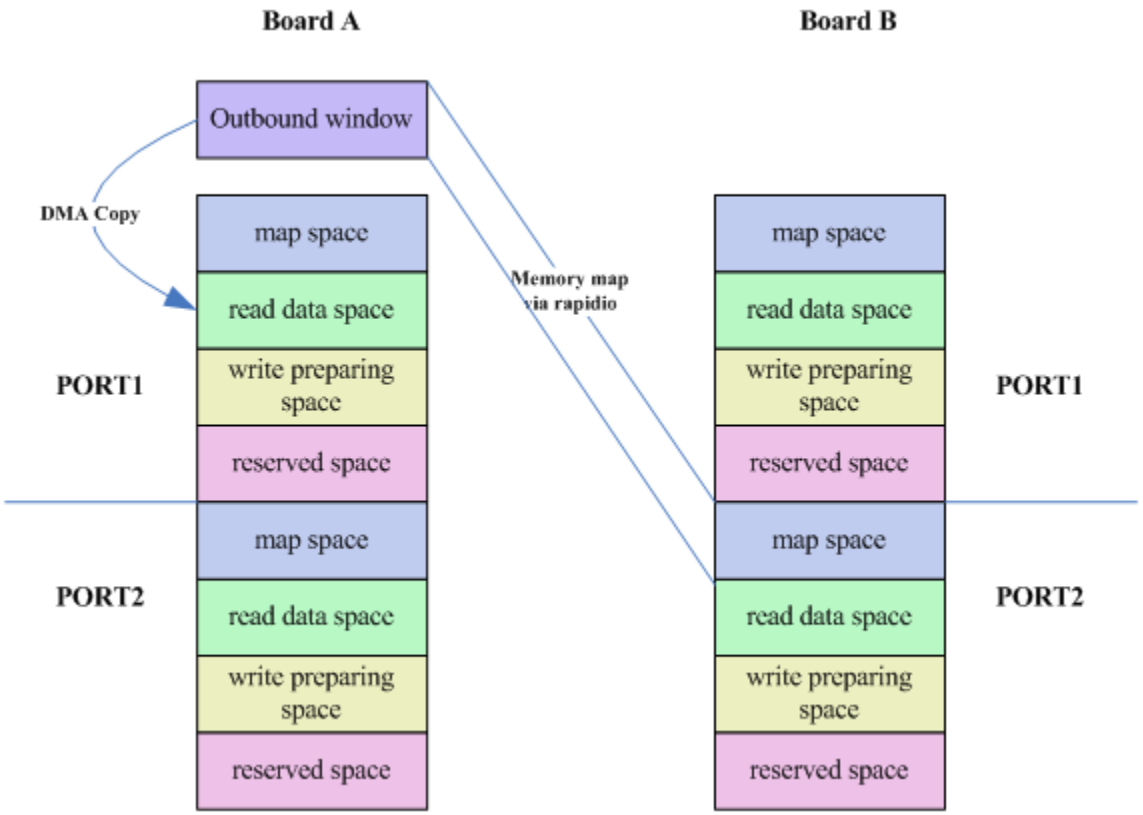
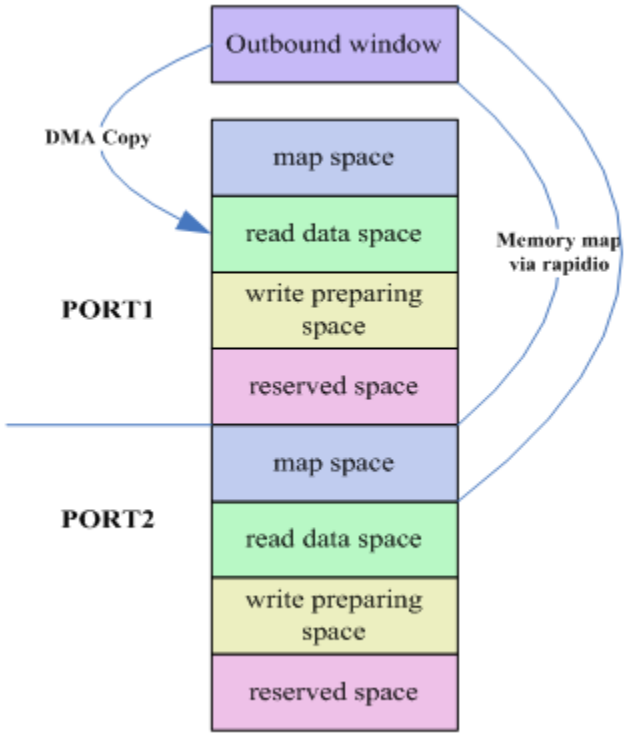
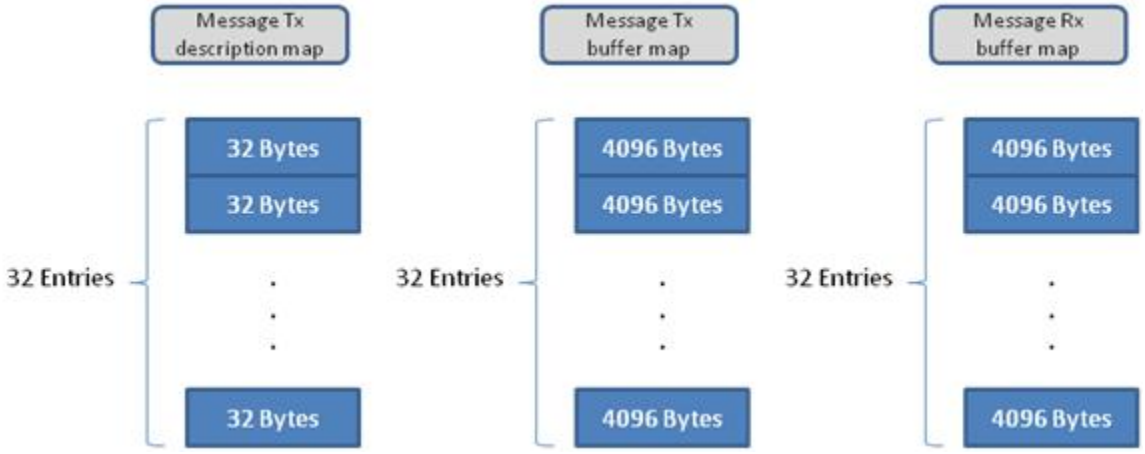
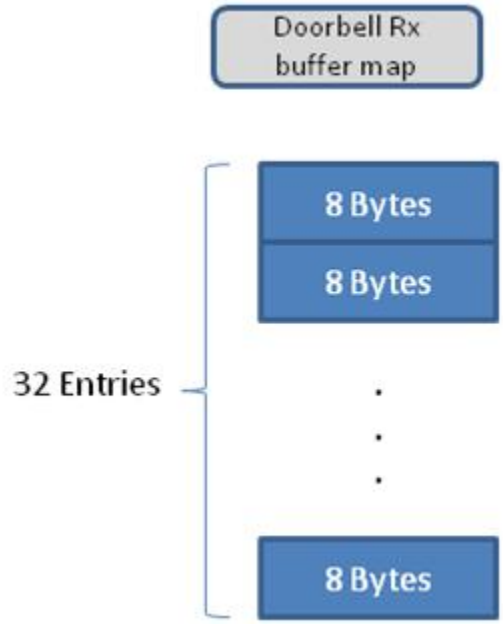
distribution order-6:
distribution-1-FMAN_TO_RMAN: fman_to_rman_dtsec3
FMan:dtsec3 - rio port 0
did:1
rio_tran:dstr-dtsec3 type:Data-streaming
fra>
```

• Enable or disable debug information output

Note: this function is effective only when compiling with ENABLE_FRA_DEBUG definition

```
fra> debug [on/off]
```

• If you want to quit FRA , should use the following command, FRA will release the related resource.

```
fra> q
```

9.9.15.6 Running FRA on One Board

The chapter is intended to introduce how to run FRA on one board. The operations of installation and compilation are similar to Chapter 3, So those sections please refer to above description.

9.9.15.6.1 Prepare the Hardware (one board)

FRA can run on one board P3041DS or P5020DS. As the Fig10 shown, when FRA application run on one board, needs two PC or one PC with two Ethernet interfaces. Two pc's Ethernet interfaces are connected with two RGMI interface of the board respectively. The board should been inserted two RapidIO card to slot6 and slot7. Check board's switch, sw2 is 0b10100101(slot7->SRIO2, slot6->SRIO1, slot3->PEX2, slot2->SGMI), Sw5 is 0b00010100(Serdes reference clock for bank1 is 100MHz, bank2 is 125MHz bank3 is 125MHz).



Figure 284. Serdes 0x04 protocol hardware setup

9.9.15.6.2 Configuring FRA (one board)

In this case FRA default configuration file is fra_config_dstr_port1_port2_loopback.xml. Under This configuration the packet flow is shown as Fig5. Board receives the IP packets from RGMII 1, adds RMan message descriptor to the headroom of each packet according to configuration, and then directly sends them from SRIO port 1; the packets are looped back to SRIO port2. RMan receives those packets, and transfers them to RGMI2. So PC can receive packets which it sends out.

The other configuration files are also valid. The difference with the default configuration is that board will use the same network port for receiving and sending.



Figure 285. packet flow under port1-port2 loop back mode

When we run FRA on one board, may use the following command:

```
# fra -f /usr/etc/fra_config_dstr_port1_port2_loopback.xml
```

9.9.15.6.3 Managing RCW

In order to enable board's two SRIO ports, we should select the appropriate RCW setting, As the example this section provides a valid RCW of Serdes 0x04 type for P3041Ds and P5020DS .

RCW for P3041DS is :

```
00000000: AA55 AA55 010E 0100 1260 0000 0000 0000
00000010: 241C 0000 0000 0000 1081 6400 2400 2000
00000020: FE80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 2F4C A9FA
```

RCW for P5020DS is :

```
00000000: AA55 AA55 010E 0100 0C54 0000 0000 0000
00000010: 1E12 0000 0000 0000 1081 6400 2400 2000
00000020: FE80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 FA0A 674D
```

Save the above RCW data to rcw-0x04.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x04.xxd > rcw-0x04.bin
```

9.9.15.6.4 Running FRA

According to Chapter 3 description, apply the related patches; build the software and burn images to board.

Following are some u-boot boot log:

```
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 1024 KB enabled
SRIO1: enabled
SRIO2: enabled
NAND:  1024 MiB
MMC:  FSL_ESDHC: 0
```

The above log has printed out SRIO status, if SRIO1 or SRIO2 is not enabled, we should check RCW image and uboot image.

The other output log is same with Chapter 3.

Configure FMan PCD using fmc with the XML files located in /usr/etc:

```
[root@p3041 root]# cd /usr/etc
[root@p3041 root]# fmc -c usdpaa_config_p3_p5_serdes_0x04.xml -p
usdpaa_policy_hash_ipv4.xml -a
```

Run the FRA application:

```
[root@p3041 root]# fra -f /usr/etc/fra_config_dstr_port1_port2_loopback.xml
```

We can use status command to check the FRA configuration:

9.9.15.6.5 Testing FRA (one board)

We can configure ETH2's IP address is 192.168.2.1, ETH3's IP address is 192.168.3.1. In order to test FRA, we need to send IP packets to a RGMII Ethernet interface, since FRA does not support ARP protocol, we should do IP and MAC binding on PC. On board console Linux startup log has the MAC information.

```
cpu0: fsl_mac: FSL FMan MAC API based driver ()
cpu0: fsl_mac: ffe4e2000.ethernet: FMan dTSEC version: 0x08240101
cpu0: fsl_mac: ffe4e2000.ethernet: FMan MAC address: 00:e0:0c:00:d7:01
cpu0: fsl_mac: ffe4e4000.ethernet: FMan dTSEC version: 0x08240101
cpu0: fsl_mac: ffe4e4000.ethernet: FMan MAC address: 00:e0:0c:00:d7:02
cpu0: fsl_mac: ffe4e6000.ethernet: FMan dTSEC version: 0x08240101
cpu0: fsl_mac: ffe4e6000.ethernet: FMan MAC address: 00:e0:0c:00:d7:03
cpu0: fsl_mac: ffe4e8000.ethernet: FMan dTSEC version: 0x08240101
cpu0: fsl_mac: ffe4e8000.ethernet: FMan MAC address: 00:e0:0c:00:d7:04
cpu0/0: Applying 10G tx-ecc error workaround (10GMAC-A004) ...
cpu0/0: done.
cpu0: fsl_mac: ffe4f0000.ethernet: FMan XGEC version: 0x00010330
cpu0: fsl_mac: ffe4f0000.ethernet: FMan MAC address: 00:e0:0c:00:d7:05
```

For 0x04 RCW, RGMII1 corresponds to DTSEC3, RGMII2 corresponds to DTSEC4. So we find the RGMII1 port Mac address is 00:e0:0c:00:d7:03, RGMII2 port Mac address is 00:e0:0c:00:d7:04.

Create an ARP entry on pc by using the following command:

```
$ sudo arp -s 192.168.2.2 00:e0:0c:00:d7:03
$ sudo arp -s 192.168.3.2 00:e0:0c:00:d7:04
```

On PC ping host board's DTSEC3

```
$ ping 192.168.2.2
```

Run Wireshark on PC to capture the ETH3 which connected to the board's RGMII2 port.

```
$ sudo wireshark
```

As the figure below shows, we should see the ping packets sent from PC.



Figure 286. Wireshark Capturing Packets

9.9.15.7 Revision History

Table 302. Revision History

Version	Updates
1.1	Add chapter: Overview of FRA Update FRA log
1.0	Creation of FRA UM for sdk 1.1 release. - Basic application flow - Commands to use - Testing FRA application

9.9.16 NXP USDPAA SRA User Manual

9.9.16.1 Serial RapidIO application

This User Manual describes the NXP P3041DS/P4080DS/P5020DS/T4240QDS/B4860QDS Serial RapidIO linux user space driver. This user manual provides you with instructions about how to use SRIO (Serial RapidIO) user space driver.

9.9.16.1.1 Overview

NXP Serial RapidIO user space driver is based on NXP Linux SDK. It is composed of Linux UIO driver in Linux kernel space and SRIO driver/application in Linux user space. There are dma and srio UIO drivers which map dma/srio registers and SRIO window to linux user space for user space driver usage. User space dma and srio driver provide driver interfaces for application, and timer driver provide means to measure the performance. Application can use driver interface to accomplish the final srio function. Currently, the srio driver support two rapidio ports, and the dma driver supports two controllers and eight channels based on basic direct mode and basic chain mode . The demo sra application can implement SRIO SWRITE, NWRITE, NWRITE_R, NREAD, ATOMIC_INC, ATOMIC_DEC, ATOMIC_CLR, ATOMIC_SET type protocols based on window/segment/subsegment, and give user srio performance data to evaluate NXP rapidio IP block. Currently, it can run on P3041DS/P4080DS/P5020DS/T4240QDS/B4860QDS board.

9.9.16.1.2 SRA environment setup

9.9.16.1.2.1 Hardware environment

To run SRIO application demo, have two P3041DS/P4080DS/P5020DS/T4240QDS/B4860QDS boards connected with each other via SRIO cable through their RapidIO ports, as shown in Figure 1 (SRIO riser card located in slot6 on P3041DS/ P5020DS, and located in slot3 on P4080DS, in slot6 on T4240QDS, in additional AMC2PEX-2S card on B4860QDS). Or using one board, connecting its port1 with port2 via SRIO cable, as shown in Figure 2 (SRIO riser card located in slot6 and slot7 on single P3041DS/ P5020DS). Then burn the updated rcw, u-boot, ucode for Fman to flash to setup board environment.

For a two SRIO port operation on single board, burn rcw-0x04 file (support 4 lanes for each rapidio port) on P3041DS/ P5020DS. For two board SRIO connection, burn rcw-0x33 file on P3041DS/ P5020DS (support 4 lanes for SRIO1), and rcw-0x16 on P4080DS (support 4 lanes for each rapidio port, but only SRIO1 can be connected between two P4080DS with 4 lanes). These RCW can be found in the SDK. P4080DS can also use rcw-0x1d to accomplish two board connection via two RapidIO ports with 1 lane for each port. Since this RCW is not included in SDK, generate it following the later RCW generation guide. For T4240QDS and B4860QDS platforms, you should create the new rcw files using QCS based on your specific environment.

For P3041DS/P5020DS, when using rcw-0x33 for two board connection via SRIO, check the board's switch: sw2 is "0b00100001"(slot7->PEX1, slot6->SRIO, slot4->PEX3, slot2->XAUI), sw5 is "0b00010100" (Serdes reference clock for bank1 is 100MHz, bank2 is 125MHz bank3 is 125MHz). For P3041DS/P5020DS, when using rcw-0x04 for two SRIO port connections on one board, board's switch should be: sw2 is "0b10101001"(slot7->SRIO2, slot6->SRIO1, slot3->PEX2, slot2->SGMI), Sw5 is "0b00010100" (Serdes reference clock for bank1 is 100MHz, bank2 is 125MHz bank3 is 125MHz). For P4080DS, when using rcw-0x16 for two SRIO port connections between two boards, board's switch should be: sw3 is "0b01001100" (Serdes reference clock for bank1 is 125MHz, bank2 is 125MHz, bank3 is 125MHz).



Figure 287. SRIO Hardware Connection between Two Boards



Figure 288. SRIO Hardware Connection between two Ports on One Board

9.9.16.1.2.2 SDK Installation

SRA is based on NXP Linux SDK. So to use SRA, make sure NXP Linux SDK is well installed.

9.9.16.1.2.3 RCW Generation

For P3041DS and P5020DS platforms, user can use NXP PBL tools to create SerDes 0x04 RCW file (support srio two ports) or SerDes 0x33 RCW file (support srio one port) and then convert to rcw image.

SerDes 0x33 RCW of P3041DS is :

```
0000000: aa55 aa55 010e 0100 1260 0000 0000 0000
0000010: 241c 0000 0000 0000 cc98 4a00 0300 2000
0000020: fe80 0000 4100 0000 0000 0000 0000 0000
0000030: 0000 0000 1007 0000 0000 0000 0000 0000
0000040: 0000 0000 0000 0000 0813 8040 cf8e 2f88
```

SerDes 0x33 RCW of P5020DS is :

```
0000000: aa55 aa55 010e 0100 0c54 0000 0000 0000
0000010: 1e12 0000 0000 0000 cc98 4a00 0300 2000
0000020: fe80 0000 4100 0000 0000 0000 0000 0000
0000030: 0000 0000 1007 0000 0000 0000 0000 0000
0000040: 0000 0000 0000 0000 0813 8040 1ac8 e13f
```

Save the above RCW data to rcw-0x33.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x33.xxd  > rcw-0x33.bin
```

SerDes 0x04 RCW for P3041DS is :

```
00000000: aa55 aa55 010e 0100 1260 0000 0000 0000
00000010: 241c 0000 0000 0000 1081 6400 2400 2000
00000020: fe80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 2f4c a9fa
```

SerDes 0x04 RCW for P5020DS is :

```
00000000: aa55 aa55 010e 0100 0c54 0000 0000 0000
00000010: 1e12 0000 0000 0000 1081 6400 2400 2000
00000020: fe80 0000 4100 0000 0000 0000 0000 0000
```

```
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 fa0a 674d
```

Save the above RCW data to rcw-0x04.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x04.xxd > rcw-0x04.bin
```

For P4080DS platform, SerDes 0x16 RCW file supports srio two X4 ports; SerDes 0x1d RCW file supports srio two X1 ports.

SerDes 0x16 RCW for P4080DS is :

```
00000000: aa55 aa55 010e 0100 105a 0000 0000 0000
00000010: 1e1e 181e 0000 cccc 5840 0000 3c3c 2000
00000020: fe80 0000 e100 0000 0000 0000 0000 0000
00000030: 0000 0000 008b 6000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 532a bb17
```

Save the above RCW data to rcw-0x16.xxd file and then convert to RCW image

```
$ xxd –r rcw-0x16.xxd > rcw-0x16.bin
```

SerDes 0x1d RCW for P4080DS is :

```
0000000: aa55 aa55 010e 0100 0c58 0000 0000 0000
0000010: 1818 1818 0000 8888 7440 4000 0000 2000
0000020: fe80 0000 0100 0000 0000 0000 0000 0000
0000030: 0000 0000 0083 0000 0000 0000 0000 0000
0000040: 0000 0000 0000 0000 0813 8040 0129 56be
```

Save the above RCW data to rcw-0x1d.xxd file and then convert to RCW image

```
$ xxd –r rcw-0x1d.xxd > rcw-0x1d.bin
```

For P2041RDB platform, SerDes 0x02 RCW file supports srio one X4 ports. RapidIO card should be inserted to slot1. and the following commands should be ran first:

```
cpld lane_mux 6 0;
cpld lane_mux a 0;
cpld lane_mux c 0;
cpld lane_mux d 0;
cpld reset altbank;
```

SerDes 0x02 RCW for P2041RDB is :

```
00000000: AA55 AA55 010E 0100 1260 0000 0000 0000
00000010: 241C 0000 0000 0000 0899 30C0 C7C0 2000
00000020: FE80 0000 4000 0000 0000 0000 0000 0000
00000030: 0000 0000 D003 0F07 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 57B0 F7D9
```

Save the above RCW data to rcw-0x02.xxd file and then convert to RCW image

```
$ xxd –r rcw-0x02.xxd > rcw-0x02.bin
```

For T4240QDS and B4860QDS platforms, please select the right RCW files from the released SDK.

u-boot/kernel/sra generation:

Please follow the SDK build guide.

9.9.16.1.2.4 Build Configuration

When build kernel, make sure these options are selected.

```
Device Drivers  --->
          <*> Userspace I/O drivers  --->
                    <*>     Freescale Serial RapidIO support
                    <*>     Freescale DMA support
```

9.9.16.1.3 Boot

To run the USDPAA software, one must first boot Linux with the correct files. These files are including u-boot.bin, uImage, p3041ds/p4080ds/p5020ds/t4240qds/b4860qds-usdpaa.dtb and initramfs.cpio.gz.uboot. Please use the ucode and rcw file in the SDK package.

Following are some booting information points:

u-boot boot log (rcw-0x04):

```
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 1024 KB enabled
SRIO1: enabled
SRIO2: enabled
NAND:  1024 MiB
```

Linux kernel boot log:

```
cpu2: fsl_oh: FSL FMan Offline Parsing port driver ()
cpu2: fsl_oh: dpa-fman0-oh.32: Found OH node handle compatible with fsl,dpa-oh.
cpu2: fsl_oh: dpa-fman0-oh.32: OH port /soc@ffe000000/fman@400000/port@82000 enabled.
fsl-of-srio ffe0c0000.rapidio: Rapidio UIO driver initialized
fsl-of-dma ffe100300.dma: dma channel dma-uio0-0 initialized
fsl-of-dma ffe100300.dma: dma channel dma-uio0-1 initialized
fsl-of-dma ffe100300.dma: dma channel dma-uio0-2 initialized
fsl-of-dma ffe100300.dma: dma channel dma-uio0-3 initialized
fsl-of-dma ffe101300.dma: dma channel dma-uio1-0 initialized
fsl-of-dma ffe101300.dma: dma channel dma-uio1-1 initialized
fsl-of-dma ffe101300.dma: dma channel dma-uio1-2 initialized
fsl-of-dma ffe101300.dma: dma channel dma-uio1-3 initialized
ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
```

9.9.16.1.4 SRA Demo

SRA is a demo application to implement SRIO functions, via dma and srio driver interface. It supports SRIO SWRITE, NWRITE, NWRITE_R, NREAD, ATOMIC_INC, ATOMIC_DEC, ATOMIC_CLR, ATOMIC_SET type protocols based on window/segment/subsegment, using command input. It can read data from the other board memory, and write data to the other board memory, then display the memory. It can also evaluate the srio performance, and print out the performance result. These functions are done by dma direct mode operation or core. It also supports dma basic chain mode test and basic direct mode performance test. Currently, it supports two rapidio ports. And you can assign a port to implement rapidio performance test. It will measure the srio performance under different transmission protocol type, with different transmission data size, and with different dma BWC (bandwith control).

As shown in Figure 3, SRA defines two large areas for two rapidio ports in dma pool. Each large area is 8M bytes. And the large area is divided into four small areas, each of which is 2M bytes. They are:

map space - match data written from other rapidio ports come into this area.

read data space - when read data from other ports, the data is saved in this area.

writing preparing space - this area is preparing data for writing to other rapidio ports.

reserved space - unused.



Figure 289. SRA DMA Pool Partition

9.9.16.1.5 SRA Command Description

There are sra command formats as followings:

```
sra -attr [port_id] [fun] [fun_id]([write_attr] [read_attr])

sra -attr port1/2 device_id [id]
sra -attr port1/2 target_id [id]
sra -attr port1/2 seg_num [num]
sra -attr port1/2 subseg_num [num]
sra -attr port1/2 subseg_tdid [seg_id] [tdid]
sra -attr port1/2 accept_all [id]
sra -attr port1/2 irq [id]
sra -attr port1/2 win_attr [id] [write_attr] [read_attr]
sra -attr port1/2 seg_attr [id] [write_attr] [read_attr]
```

Notes: This is sra attribute command serial, which set rapidio attribute.

-attr: It indicates this is an attribute setting command.

port_id: It indicates which port is set currently. It can be these characters: port1, port2.

fun: It indicates which function this command sets. The function names are listed above.

fun_id: It indicates which operation this function uses. In most case, it is a number.

write_attr: This is writing attribute when sra do writing operation. It can be these characters: swrite, nwrite, nwrite, nwrite_r which are consistent with rapidio standard protocol.

read_attr: This is reading attribute when sra do reading operation. It can be these characters: nread, atomic_inc, atomic_dec, atomic_clr, atomic_set, which are consistent with rapidio standard protocol.

```
sra -op [port_id] [win_id] [seg_id] [subseg_id] [operation] [data_len]

sra -op port1/2 [win_id] [seg_id] [subseg_id] w/r/s/p [data_len]
```

-op: It indicates this is an operation command.

port_id: It indicates which port is set currently. It can be these characters: port1, port2.

win_id: It indicates which window this operation works on. Currently, application uses window 1 as the only window for operation.

seg_id: It indicates which segment this operation works on.

subseg_id: It indicates which subsegment this operation works on.

operation: It indicates what operation should do.

s(set): It sets 0/1/2/3/4/5/6/7 in different area to local memory and ignore the data_len value.

p(print): It prints definite parts of local memory and ignore the data_len value.

w(write): It sends data in current port write preparing space to the other board map space in memory according to data_len.

r(read): It reads the other board port map space data, and saves them in its read data space in local memory according to data_len.

data_len: This is the data transmission length. It can be any number which is smaller than 2M. And for ATOMIC operation, the data_len should be 1/2/4. Other number using for ATOMIC reading operation will cause transmission fault. The data_len will be ignored for the s(set) and p(print) commands, but when run the s(set) and p(print) commands you also should give the data_len parameter, any value will be ok.

```
sra -test [case] [port] [task] [srio] [number]
 sra -test srio port1/2 dma/core [srio_type] payload_size
        [srio_type] should be swrite/nwrite/nwrite_r/nread
        payload_size should be less than 2M bytes
 sra -test dma_chain
 sra -test show_perf port1/2 times
 sra -test show_task
 sra -test free_task port1/2
```

-test: It indicates sra will do test or create a srio transmission task.

case: The type of the test, can be srio, dma_chain, show_perf, show_task, or free_task. There should be different following parameters for different test case. And also there are some dependent relationships between the cases.

port: Indicates which SRIO port will be used for the test, can be port1 or port2.

task: Determines the mode of the test, can be dma or core. If you select the core mode, the transmission will be implemented with memcpy function, so you'd better very clear about the effect of the cache. The testing memory space for the srio was configured to be cached by default, so the results of the core test may don't make any sense for the normal requirements.

srio: Indicates the type of the srio transmission, can be swrite, nwrite, nwrite_r, or nread.

number: The payload size of the srio transmission task or the times for a performance calculation.

9.9.16.1.6 SRA command Usage

- SRA Help

```
#sra
sra> sra
```

Then sra help will display. Error command will cause sra to print help information. Followings are the help information:

```
-----------------SRIO APP CMD FORMAT-----------------
Set window attribute
sra -attr [port_id] [fun] [fun_id]([write_attr] [read_attr])
sra -attr port1/2 device_id [id]
sra -attr port1/2 target_id [id]
sra -attr port1/2 seg_num [num]
sra -attr port1/2 subseg_num [num]
sra -attr port1/2 subseg_tdid [seg_id] [tdid]
sra -attr port1/2 accept_all [id]
sra -attr port1/2 irq [id]
sra -attr port1/2 win_attr [id] [write_attr] [read_attr]
sra -attr port1/2 seg_attr [id] [write_attr] [read_attr]

Notes:
        [id] for command accept_all: 0 - disable; 1 - enable
        [id] for irq: 0 - disable; 1 - enable
                [write_attr]    : swrite/nwrite/nwrite_r
        [read_attr]     :
 nread/atomic_inc/atomic_dec/atomic_set/atomic_clr

Do sra operation
sra -op [port_id] [win_id] [seg_id] [subseg_id] [operation] [data_len]
sra -op port1/2 [win_id] [seg_id] [subseg_id] w/r/s/p [data_len]
        [data_len]      : should be less than the window/segment/subsegment's size, max size is 2M
                        data_len should be 1/2/4 for ATOMIC operation

Do SRIO test and print performance result
sra -test [case] [port] [task] [srio] [number]
 sra -test srio port1/2 dma/core [srio_type] payload_size
        [srio_type] should be swrite/nwrite/nwrite_r/nread
       payload_size should be less than 2M bytes
 sra -test dma_chain
 sra -test show_perf port1/2 times
 sra -test show_task
 sra -test free_task port1/2
-----------------------------------------------------
```

- **SRA Set SRIO Port Device Id**

This command sets SRIO port device id. Any package from other srio device will be accepted only when the sender's target id matches the accepter's device id.(When "accept all" functions is closed)

```
sra> sra -attr port1 device_id 0x55
```

- **SRA Set SRIO Port Target Id**

This command sets SRIO port1 target id.

```
sra> sra -attr port1 target_id 0x33
```

- **SRA Set SRIO Port Segment Number**

This command sets the segment number in a window of a srio port. The number can be 1/2/4.

```
sra> sra -attr port1 seg_num 4
```

- **SRA Set SRIO Port Sub Segment Number**

This command sets the sub segment number in a segment of a srio port. The number can be 1/2/4/8. This command needs the segment to be set firstly.

```
sra> sra -attr port1 subseg_num 4
```

•SRA Set SRIO Port Sub Segment base target id

This command sets the sub segment base target id for subsegment serial of a srio port. This command needs srio segment/subsegment number be set firstly. [seg_id]: It indicates the segment which is used for subsegment setting. [tdid]: subsegment base target id. It should be aligned with the subsegment number.

```
sra> sra -attr port1 subseg_tdid 1 0x40
```

•SRA Set SRIO Port enable or disable accept all function

This command can enable or disable srio port accept all package feature. [id]: 0 - disable the feature; 1 - enable the feature.

```
sra> sra -attr port1 accept_all 1
```

•SRA Set SRIO enable or disable irq

This command can enable or disable srio irq. [id]: 0 - disable irq; 1 - enable irq.

```
sra> sra -attr port1 irq 1
```

•SRA Set SRIO Port Window Attributes

This command sets SRIO port outbound window attributes. The outbound window attribute for write can be nwrite, swrite, nwrite_r, and for read can be nread, atomic_inc, atomic_dec, atomic_set, atomic_clr. [id]: indicates window id. Currently, this parameter should 1, indicating sra setting window 1.

```
sra> sra -attr port1 1 nwrite nread
```

•SRA Set SRIO Port Window1 Segment Attributes

This command sets SRIO port1/2 window1 segment attribute. The outbound window segment attribute for write can be nwrite, swrite, nwrite_r, and for read can be nread, atomic_inc, atomic_dec, atomic_set, atomic_clr. [id]: indicates window1 segment id. Currently, this parameter should be compatible with segment number. Ex. if segment number is 4, the segment id can be 1/2/3/4.

•SRA Setting and Printing Operation

This command sets 0/1/2/3/4/5/6/7 to the eight areas in local memory. [win_id] indicates the window id. [seg_id] indicates the segment id. [subseg_id] indicates the subsegment id.

```
sra> sra -op port1 1 0 0 s 0x100000
```

This command prints parts of local memory data.

```
sra> sra -op port1 1 0 0 p 0x100000
```

•SRA Writing Operation

This command sends 1M bytes data in port writing preparing space to other board memory, using port attribute set in attribute command.

```
sra> sra -op port1 1 0 0 w 0x100000
```

•SRA Reading Operation

This command uses port attribute to read 1M byte data from other port, and stores the data in port read data space in local memory.

Notes: ATOMIC operation requires data length should be 1/2/4.

```
sra> sra -op port2 1 0 0 r 0x100000
```

•SRA Test Operation

This command uses srio port1 or port2 to implement rapidio performance test, or do dma basic chain mode test based on the [case] parameter input. In rapidio performance test, if you just give the parameters [case], [port], [task], it will measure the srio performance under different transmission protocol type, with different transmission data size, and with different dma BWC (bandwith control), and print out the performance result. If you give all the parameters [case], [port], [task], [srio], [number], it will create a task for the specific srio transmission performance test. When you want to get the performance result from the task, you can start the calculation by the other command. In dma basic chain mode test, sra will copy lower region data to the upper region.

•SRA Test under dma mode

It will measure the srio port1 performance using dma under different transmission protocol type with different transmission data size, and with different dma BWC (bandwith control). It will print out the performance result finally.

```
sra> sra -test srio port1 dma
```

•SRA Test under core mode

It will measure the srio port1 performance using core under different transmission protocol type with different transmission data size. It will print out the performance result finally.

Note: The cache will impact the results of the performance.

```
sra> sra -test srio port1 core
```

•SRA Test to create a transmission task with dma mode

It will create a srio transmission task with the port1, dma mode, the swrite protocol type, and the 4096 bytes payload size. It can print out the performance result for the specific transmission when start the performance calculation with the other command.

Note: The payload size should be less then 2M bytes.

```
sra> sra -test srio port1 dma swrite 4096
```

•SRA Test to create a transmission task with core mode

It will create a srio transmission task with the port2, core mode, the nread protocol type, and the 2048 bytes payload size. It can print out the performance result for the specific transmission when start the performance calculation with the other command.

Note: The payload size should be less then 2M bytes, and the cache will impact the results of the performance.

```
sra> sra -test srio port2 core nread 2048
```

•SRA Test for dma chain

The sra will implement the dma chain mode test.

```
sra> sra -test dma_chain
```

•SRA Test to start the task's performance calculation

When there is a srio task on port1, this command can start the performance calculation with the 10000 transmission times. And it will print out the performance result finally.

```
sra> sra -test show_perf port1 10000
```

•SRA Test to show the existing tasks

When there are srio tasks, this command can show the all tasks on port1 and port2, and print out the task's parameters.

```
sra> sra -test show_task
```

•SRA Test to free the task

When there is a srio task on port1, this command can free the task.

```
sra> sra -test free_task port1
```

9.9.16.1.7 Run SRA Demo

Example 1: Board A sends 1M byte data to board B memory via rapidio NWRITE protocol. Two boards are connected via board A port1 and board B port1.

The figure below describes the two board memory map. Board A copies data from its port1 write preparing space to outbound window via dma. And board A outbound window is mapped to board B port1 map space via rapidio system. So writing to board A outbound window just like writing to board B ddr. Following is the command operation steps.



Figure 290. SRIO NWRITE Operation with Two Boards

```
Step1:      Boot up two boards.
Step2:
```

```
Board B:
sra> sra -attr port1 win_attr 1 nwrite nread (set board B port1 window 1 attribute to nwrite,
nread)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see the memory data)
Step3:
Board A:
sra> sra -attr port1 win_attr 1 nwrite nread (set board A port1 outbound window with nwrite,
nread attribute)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether memory is set)
sra> sra -op port1 1 0 0 w 0x100000 (send 1M data in writing preparing space to board B map space)
Step4:
Board B:
sra> sra -op port1 1 0 0 p 0x100000 (To see whether board A write preparing data are written to
board B map space)
```

Example 2: Board A NREAD 1M byte data from Board B memory. Two boards are connected via board A port1 and board B port2.

The figure below describes the two board memory maps. Board A copies data from its port1 outbound window to its read data space via dma. And board A port1 outbound window is mapped to board B port2 map space via rapidio system. So reading from board A outbound window just like reading from board B ddr. Following is the command operation steps.



Figure 291. SRIO NREAD Operation with Two Boards

```
Step1:      Boot up the two boards.
Step2:
Board B:
sra> sra -attr port2 win_attr 1 nwrite nread (set board B port2 outbound window attribute to
```

```
nwrite, nread)
sra> sra -op port2 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port2 1 0 0 p 0x100000 (to see the memory data)
Step3:
Board A:
sra> sra -attr port1 win_attr 1 nwrite nread (set board A port1 outbound window attribute to
nwrite, nread)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether memory is set)
sra> sra -op port1 1 0 0 r 0x100000 (read 1M data from board B map space, and save them in its
read data space)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether board B map space data is saved in its port1
read data space)
```

Example 3: Board A ATOMIC_INC read 4 byte from Board B memory. Two boards are connected via board A port1 and board B port1.

```
Step1:      Boot up the two boards.
Step2:
Board B:
sra> sra -attr port1 win_attr 1 nwrite nread (set board B port1 outbound window attribute to
nwrite, nread)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see the memory data)
Step3:
Board A:
sra> sra -attr port1 win_attr 1 nwrite atomic_inc (set board A port1 outbound window attribute to
nwrite, atomic_inc)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether the data is set)
sra> sra -op port1 1 0 0 r 0x4 (read 4 byte data from board B map space, and save them in its
read data space)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether read data is correct)
Step4:
Board B:
sra> sra -op port1 1 0 0 p 0x100000 (to see whether Board B data is added by 1)
```

Example 4: As shown in the figure below, on one board, using NREAD protocol to read 1M byte data from its port2 map space, and then store data into its port1 read data space. In this scenario, user can test srio two port functions on one board. It needs connecting two srio ports on P3041 board via srio cable.



Figure 292. SRIO NREAD Operation with Two Ports on One Board

```
Step1:      Boot up the board.
Step2:
sra> sra -attr port1 win_attr 1 nwrite nread (set port1 outbound window attribute to nwrite,
nread)
sra> sra -attr port2 win_attr 1 nwrite nread (set port2 outbound window attribute to nwrite,
nread)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see the memory data)
sra> sra -op port1 1 0 0 r 0x100000 (port1 reads 1M byte data from port2 map space, and save them
in port1's read data space)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether read data is correct)
```

NOTE

Currently, SRA is used to demonstrate IO related transaction functionalities, not include functionalities of Message unit and RMan, and not support Doorbell Mailbox Data-streaming transactions.

9.9.16.2 Revision History

Document revision history.

Table 303. Revision History

Version	Author	Description

9.9.17 USDPAA RMU User Manual

9.9.17.1 RapidIO Message Unit Application

This User Manual describes the NXP RapidIO Message Unit Linux user space driver, provides you with instructions about how to use RMU user space application.This user manual provides you with instructions about how to use RMU user space application.

9.9.17.2 Overview

NXP RapidIO Message Unit user space driver is implemented based on NXP Linux SDK. It is composed of Linux UIO driver in Linux kernel space and RMU driver/application in Linux user space. There is RMU UIO driver which maps RMU registers to Linux user space for user space driver. User space RMU driver provides interfaces for application, and application accomplishes the final RMU function. Currently, the RMU driver support two message units and one doorbell unit. The demo RMU application can implement RapidIO message and doorbell type protocols, and give performance data to evaluate NXP RMU IP block.

9.9.17.3 RMU Environment Setup

To run the RMU application, user needs to build an appropriate environment based on the requirements of hardware, software, RCW and other configurations.

9.9.17.3.1 Hardware Environment

To run RMU application demo, user needs two P4080DS boards connected with each other via SRIO cable as shown in Figure 1 (SRIO card locates in slot3 on P4080DS). Then update the RCW, U-Boot, Ucode to setup board environment.

For the RCW, user should burn RCW-0x16 on P4080DS (support 4 lanes for each RapidIO port, but only SRIO1 can be connected between two P4080DS boards with 4 lanes). The RCW can be found in SDK. P4080DS can also use RCW-0x1d to accomplish two boards connection via two rapidio 1x ports. This RCW is not included in SDK, user needs to generate it following the later RCW generation guide.

For P4080DS, when using RCW-0x16 for two SRIO ports connection between two boards, board's switch should be: sw3 is "0b01001100" (Serdes reference clock for bank1 is 125MHz, bank2 is 125MHz, bank3 is 125MHz).

Figure 293. SRIO Hardware Connection between Two Boards

9.9.17.3.2 SDK Installation

RMU is based on NXP Linux SDK. So to use RMU, make sure NXP Linux SDK is well installed.

9.9.17.3.3 RCW Generation

User can use the released RCW binaries in SDK or to create the specific RCW by NXP PBL tool.

serdes 0x16 RCW for P4080DS is :

```
00000000: aa55 aa55 010e 0100 105a 0000 0000 0000
00000010: 1e1e 181e 0000 cccc 5840 0000 3c3c 2000
00000020: fe80 0000 e100 0000 0000 0000 0000 0000
00000030: 0000 0000 008b 6000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 532a bb17
```

Save the above RCW data to rcw-0x16.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x16.xxd > rcw-0x16.bin
```

serdes 0x1d RCW for P4080DS is :

```
0000000: aa55 aa55 010e 0100 0c58 0000 0000 0000
0000010: 1818 1818 0000 8888 7440 4000 0000 2000
0000020: fe80 0000 0100 0000 0000 0000 0000 0000
0000030: 0000 0000 0083 0000 0000 0000 0000 0000
0000040: 0000 0000 0000 0000 0813 8040 0129 56be
```

Save the above RCW data to rcw-0x1d.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x1d.xxd > rcw-0x1d.bin
```

9.9.17.3.4 Kernel Building Configuration

When building kernel, make sure these options are selected.

```
Device Drivers  --->
          <*> Userspace I/O drivers  --->
                    <*> Freescale Rapidio Message Unit support
```

9.9.17.4 Boot

To run the RMU application, one must first boot Linux with the correct files. They include u-boot.bin, uImage, p4080ds-usdpaa.dtb and initramfs.cpio.gz.uboot. And please use the ucode and RCW file in the SDK package.

Following are some booting information points:

u-boot boot log:

```
POST memory PASSED
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 2048 KB enabled
SRIO1: enabled
SRIO2: enabled
MMC:  FSL_SDHC: 0
```

Linux kernel boot log:

```
fsl-of-rmu ffe0d3000.rmu: rmu unit rmu-uio-msg0 initialized
fsl-of-rmu ffe0d3000.rmu: rmu unit rmu-uio-msg1 initialized
fsl-of-rmu ffe0d3000.rmu: rmu unit rmu-uio-doorbell initialized
```

9.9.17.5 RMU Demo

RMU is a demo application to implement NXP RapidIO message and doorbell functions. It can send message or doorbell package to the partner board, then the partner can display the received package. It can also evaluate the message and doorbell unit performance, and print out the result. Currently, it supports two message units and one doorbell unit, and uses message unit 0 and SRIO port 0 to implement performance tests. It will measure the message unit performance with different transmission data size.

As shown in Figure 1, RMU defines three areas for each message unit in DMA pool. They are:

Message Tx description map – all 32 entries for the message unit Tx description ring.

Message Tx buffer map – each buffer of the 32 entries loads the Tx data of message package.

Message Rx buffer map – every received message package will be saved to each of the 32 entries buffer.



Figure 294. RMU each message unit DMA Pool Partition

The doorbell unit just needs one area in DMA pool for the received package, as shown in Figure 2.



Figure 295. RMU doorbell unit DMA Pool Partition

9.9.17.6 RMU Commands

There are RMU command formats as followings. It will also display when you give the command " rmu help" or give a wrong command.

```
-----------------RMU APP CMD FORMAT-----------------
--Set Rmu Attribute--
rmu -attr [unit_name] [fun] [slot]
rmu -attr msg0/msg1/dbell  txdesc [slot]
rmu -attr msg0/msg1/dbell  txbuff [slot]
rmu -attr msg0/msg1/dbell  rxbuff [slot]
        -[unit_name]: msg0/msg1/dbell
        -[slot]: the slot of the buffer in the ring

--Do Rmu Operation--
rmu -op [unit_name] [operation] [port_id] [dest_id] [dest_mbox] [priority] [data] [data_len]
rmu -op msg0/msg1/dbell t/r/ar/s/p [port_id] [dest_id] [dest_mbox] [priority] [data] [data_len]
        -[operation]: t---tx a message or doorbell
                r---rx a message or doorbell and print
                ar---create a rx thread for the unit and print
                s---set next tx message buffer
                p---print next tx local buffer
        -[data]: for message should be u8 size
                for dbell should be u16 size
        -[data_len]: for message should be 8~4096 bytes
                for dbell should be equal 2 bytes

--Do Rmu Test and Print Performance Result--
rmu -test [case_name]
        -[case_name]: msg/dbell
----------------------------------------------------
```

As shown in the above print information, RMU commands consist of three formats:

-attr: Attribute printing serial. Print the buffers of the message or doorbell unit.

```
rmu -attr [unit_name] [fun] [slot]

    unit_name: which message or doorbell unit will be operated
               valid characters: msg0, msg1 or dbell

    fun: which buffer will be printed
         valid characters: txdesc, txbuff, rxbuff
         Note: The doorbell unit just has rxbuff.

    slot: which entry of the buffer in the 32 entries ring will be printed
          valid value: 0 ~ 31. Over 31 will print all the 32 entries

Example:

● Print the message unit 0 Tx description slot 0.

Command:
rmu> rmu -attr msg0 txdesc 0

Display:
ATTR: Msg0 Tx Desc ......
        cell size:32bytes; entries:32
        base virt addr:1000000; base phy addr:27000000
```

```
        desc 0; virt addr:1000000; phy addr:27000000
        01000000: 00000000 00000000 00000000 00000000
        01000010: 00000000 00000000 00000000 00000000
```

-op: Operate command serial. Accomplish sending, receiving, setting and other operations.

```
rmu -op [unit_name] [operation] [port_id] [dest_id] [dest_mbox] [priority] [data] [data_len]

    unit_name: which message or doorbell unit will be operated
               valid characters: msg0, msg1 or dbell

    operation: which operation will be implemented
               valid characters:
                     t – send a message or doorbell package
                     r – print the valid received package
                     ar – create a receive thread, it can receive and print the packages
continually
                     s – set next local Tx message buffer with a specific value
                     p – print next local Tx message buffer to verify the sending value
               Note: For each of the message units and doorbell unit, it has a 32
                     entries Rx buffer ring for all the received packages. Once
                     received a package, it will occupy a Rx buffer in the ring, the
                     buffer will be released and can be re-used after the reading
                     command. If no “r” or “ar” commands to release the used Rx
                     buffers, the message or doorbell unit can just receive 32
                     packages because of the 32 entries of the Rx buffer ring.

    port_id: which SRIO port the package will be sent through
             valid value: 0, 1
             Note: The used port must be enabled in RCW.

    dest_id: the device ID of the partner. Generally set it to 0

    dest_mbox: which mailbox of the partner will receive this package
               Valid value: 0, 1

    priority: the priority of the sending package
              valid value: 0, 1, 2
              Note: 0 is the lowest and the 2 is the highest.

    data: the value should be set or sent
          valid value: for message should be u8 size, for dbell should be u16 size

    data_len: how many bytes will be set or sent for the message package
              Note: Doorbell operations don't need this parameter.

Note: Different operation command needs different number of parameters! Followings
      are all the “-op” commands based on the msg0 and dbell.

Example:

● Send a message with msg0 to the partner’s mailbox.

Command:
rmu> rmu -op msg0 t 0 0 0 0 128

    -op: operation command
    msg0: [unit_name] - operate message unit 0
    t: [operation] - send a message package
```

```
    0: [port_id] - through the local SRIO port 0
    0: [dest_id] - the destination device ID is 0
    0: [dest_mbox] - send this package to the partner's mailbox 0
    0: [priority] - set the sending package's priority to 0
    128: [data_len] - The package payloads 128 bytes. It must be 8 ~ 4096 size.

Note: This command will send the data in the next msg0 Tx buffer. So if want to send
      a specific value, you need to accomplish "s" command first to set the Tx
      buffer.

● Print the next valid received message package for the msg0.

Command:
rmu> rmu -op msg0 r

    -op: operation command
    msg0: [unit_name] - operate message unit 0
    r: [operation] - Find the next available received message package and print

Note: If there is no available package in the Rx buffer ring, following information
      will display:

OP: next available rx message for msg0:
        message fetch failed!

● Create a receive thread for the msg0.

Command:
rmu> rmu -op msg0 ar 1

    -op: operation command
    msg0: [unit_name] - operate message unit 0
    ar: [operation] - Create a receive thread for the msg0
    1: [data] - 1: create the receive thread
                0: release the thread

Note: If create the receive thread successfully, you can see the following
      information:

OP: msg0 rx thread ...create success!

● Set next local Tx buffer for the msg0.

Command:
rmu> rmu -op msg0 s 0x5a 4096

    -op: operation command
    msg0: [unit_name] - operate message unit 0
    s: [operation] - Set a specific value to the next local Tx buffer of the msg0
    0x5a: [data] - the value to be set
    4096: [data_len] - set 4096 bytes

Note: For the message Tx buffer, [data] must be one byte size. The size of the Tx
      buffer is 4096 bytes, so the max length you can set is 4096. There will be the
      following information after the command:

OP: msg0 NEXT tx buffer SET, slot:0
        txbuff 0; virt addr:1020000; phy addr:27020000
        Slot 0 tx buff set down.
```

```
● Print the next local Tx buffer of the msg0 to verify the sending value.

Command:
rmu> rmu -op msg0 p 64

    -op: operation command
    msg0: [unit_name] - operate message unit 0
    p: [operation] - print the next local Tx buffer with specific printing size.
    64: [data_len] - just print 64 bytes of the Tx buffer.

Note: The max size of the Tx buffer is 4096, so the [data_len] should be equal or
      smaller than 4096. The following information will display after the command:

OP: msg0 NEXT tx buffer PRINT, slot:1
        txbuff 1; virt addr:1021000; phy addr:27021000
        01021000: 00000000 00000000 00000000 00000000
        01021010: 00000000 00000000 00000000 00000000
        01021020: 00000000 00000000 00000000 00000000
        01021030: 00000000 00000000 00000000 00000000

● Send a doorbell package to the partner.

Command:
rmu> rmu -op dbell t 0 0 0 0x5a5a

    -op: operation command
    dbell: [unit_name] - operate doorbell unit
    t: [operation] - send a doorbell package
    0: [port_id] - through the local SRIO port 0
    0: [dest_id] - the destination device ID is 0
    0: [priority] - set the sending package's priority to 0
    0x5a5a: [data] - the value to be doorbell package.

Note: The payload of the doorbell package must be 2 bytes.

● Print the next valid received doorbell package for the dbell.

Command:
rmu> rmu -op dbell r

● Create a receive thread for the dbell.

Command:
rmu> rmu -op dbell ar 1

Note: Dbell has no the "s" and "p" commands because it does not need the Tx buffers.
```

-test: Message unit or doorbell unit performance test commands.

```
rmu -test [case_name]

    case_name: which performance test will be done
               valid characters: msg, dbell

Note: It will use the msg0 for the message unit test. In order to avoid running out
      of the partner's Rx buffer, it's recommended to open the Rx thread in
      partner's board using the "ar" command.

Example:
```

```
● Message unit performance test.

Command:
rmu> rmu -test msg

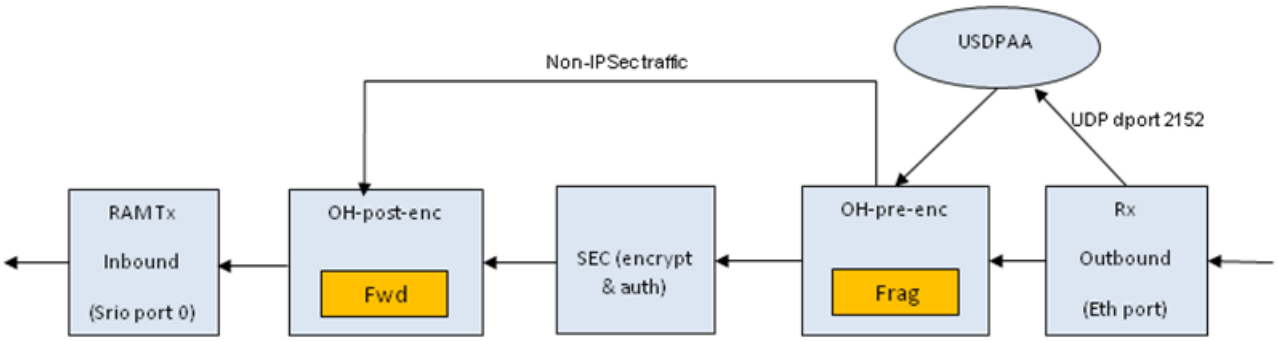
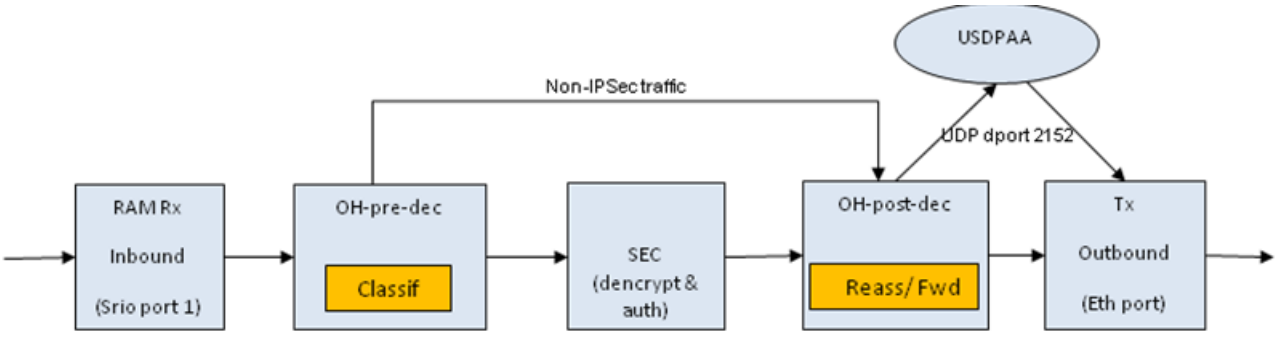
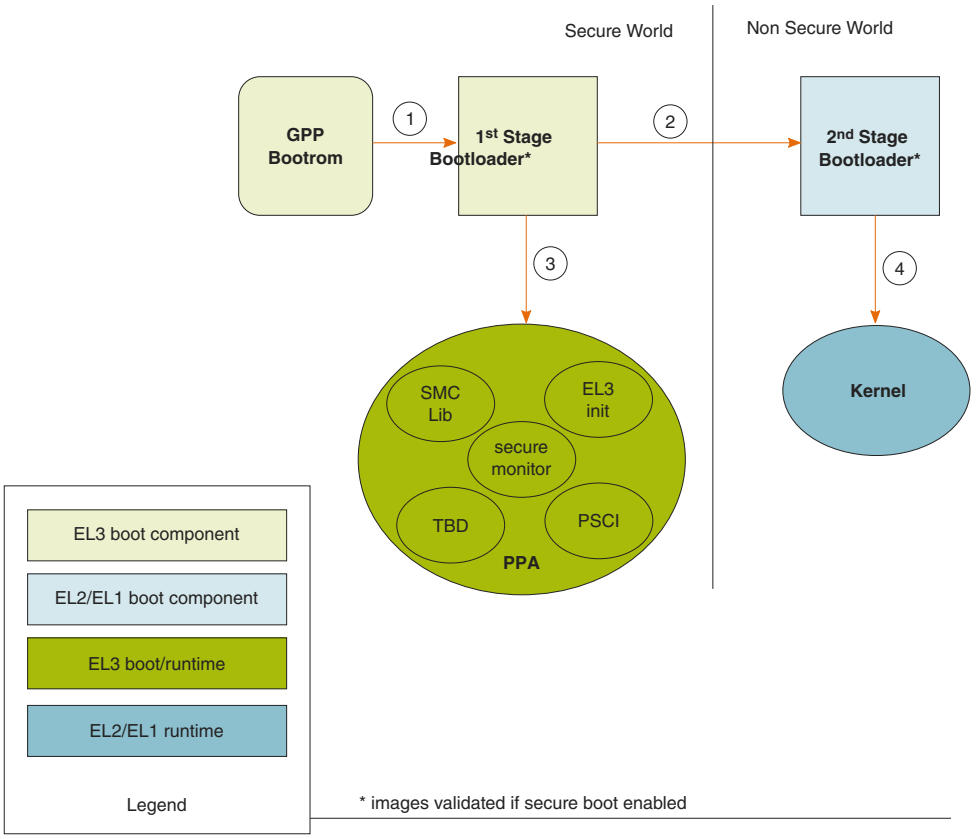
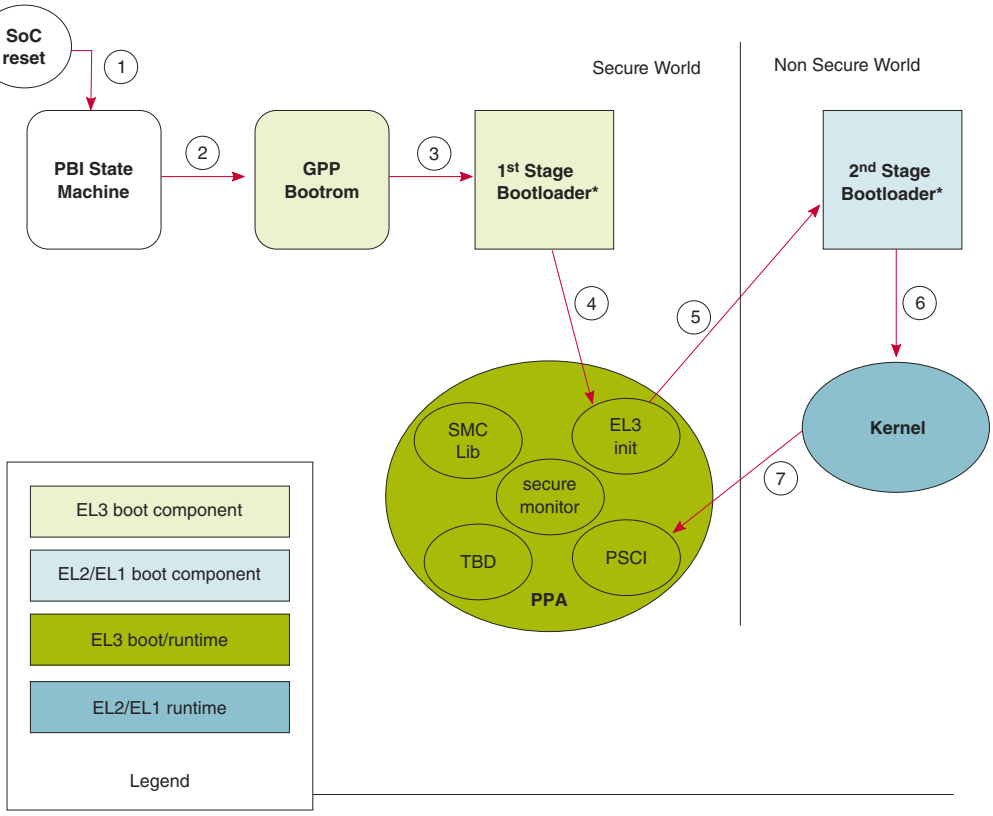
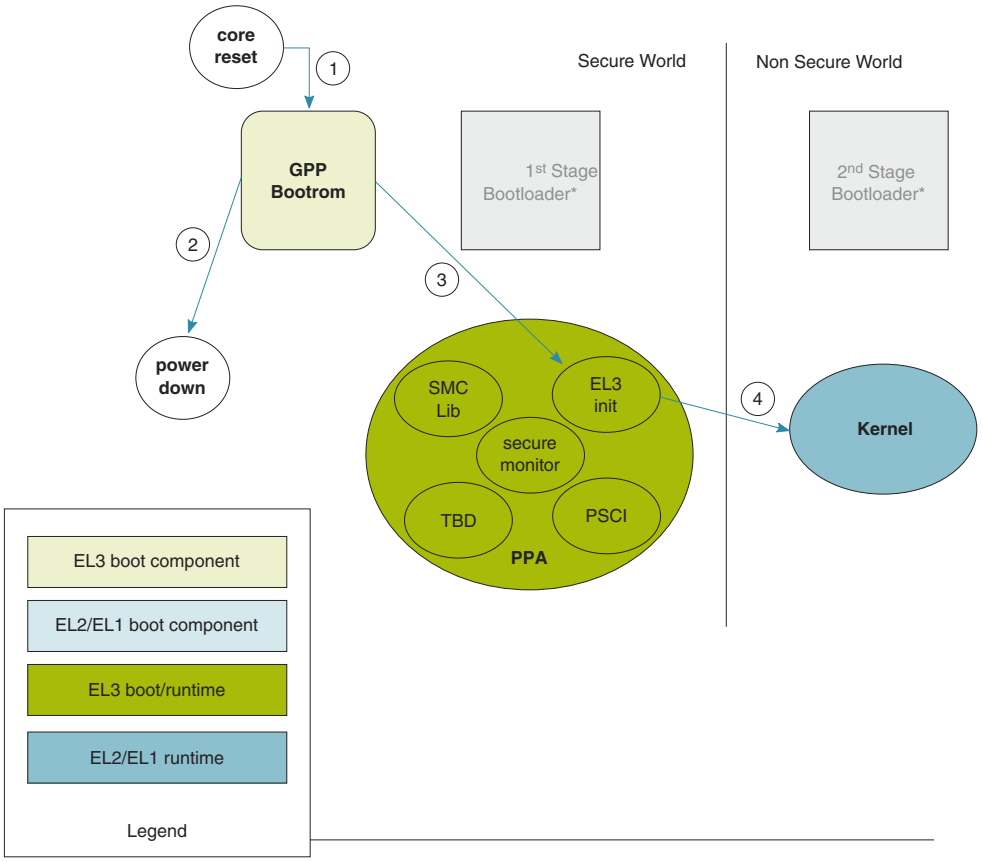
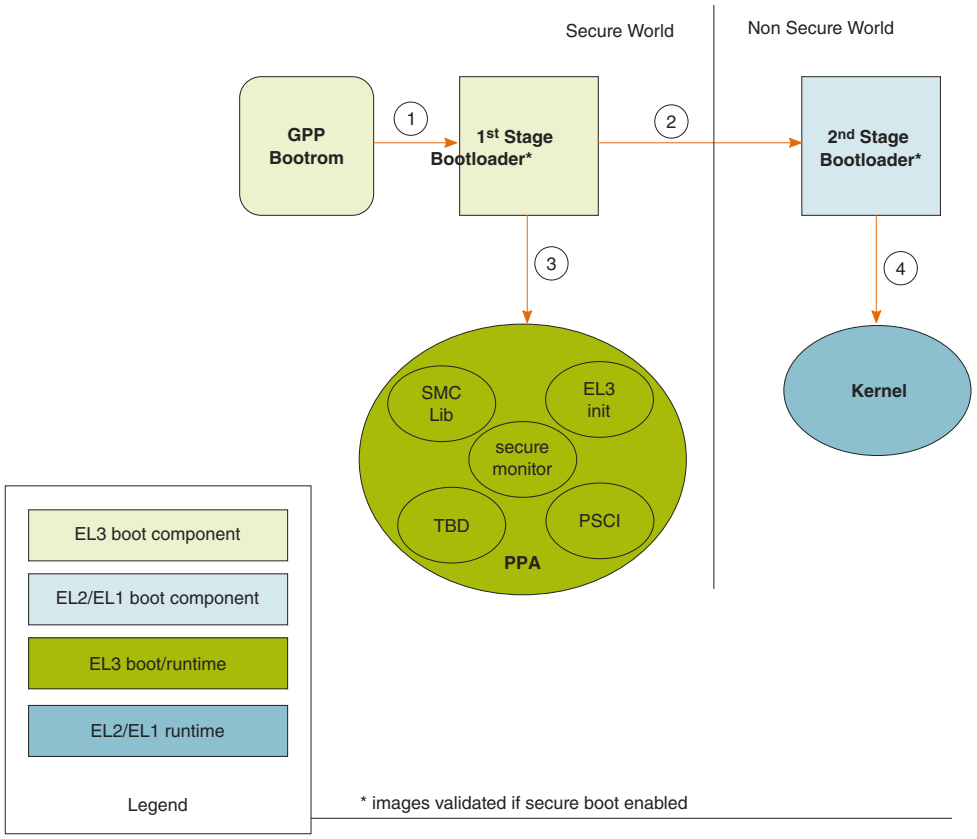
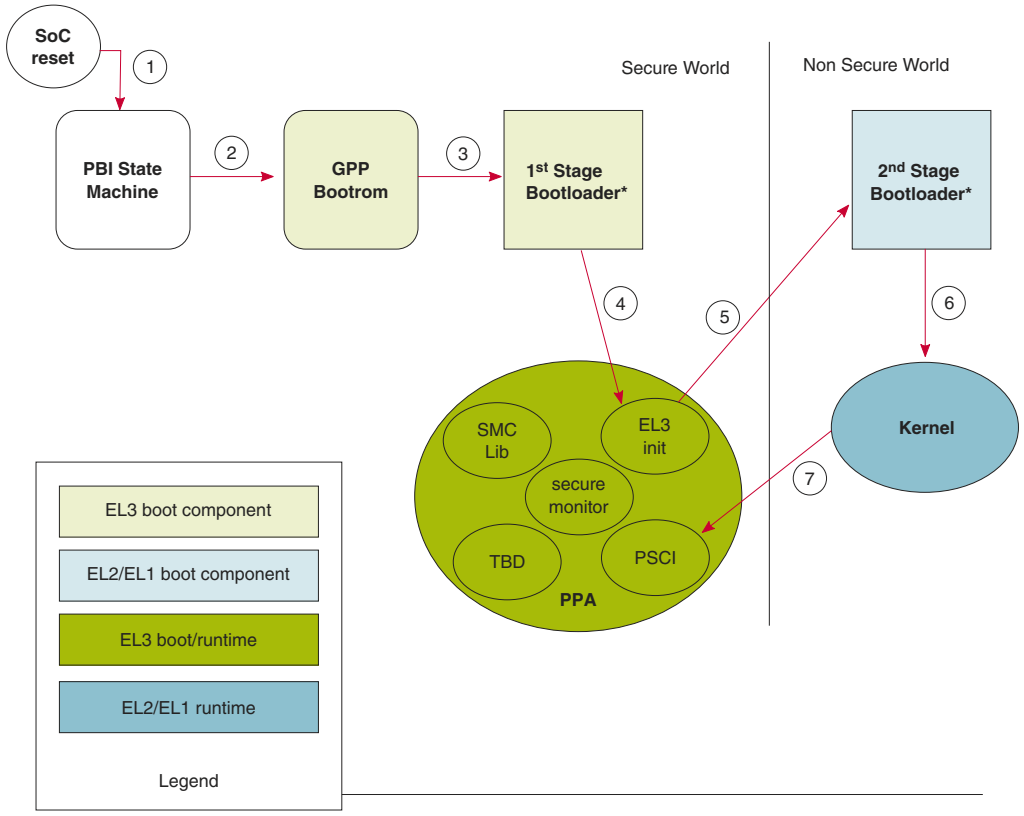
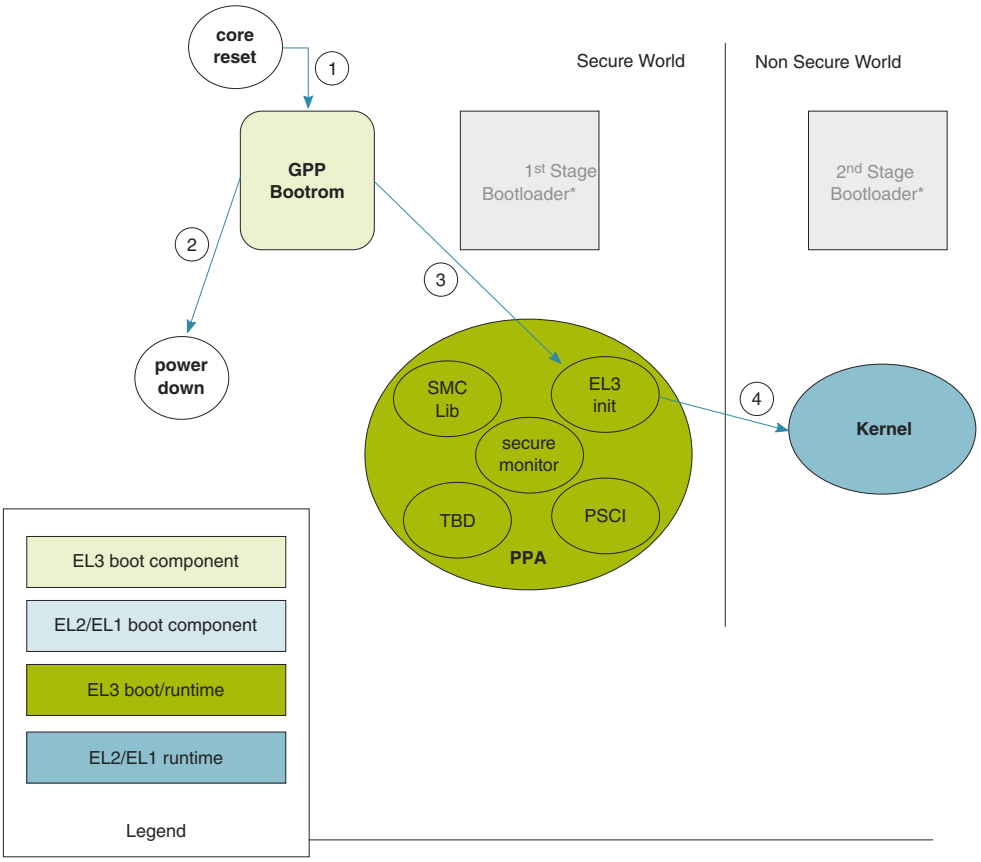
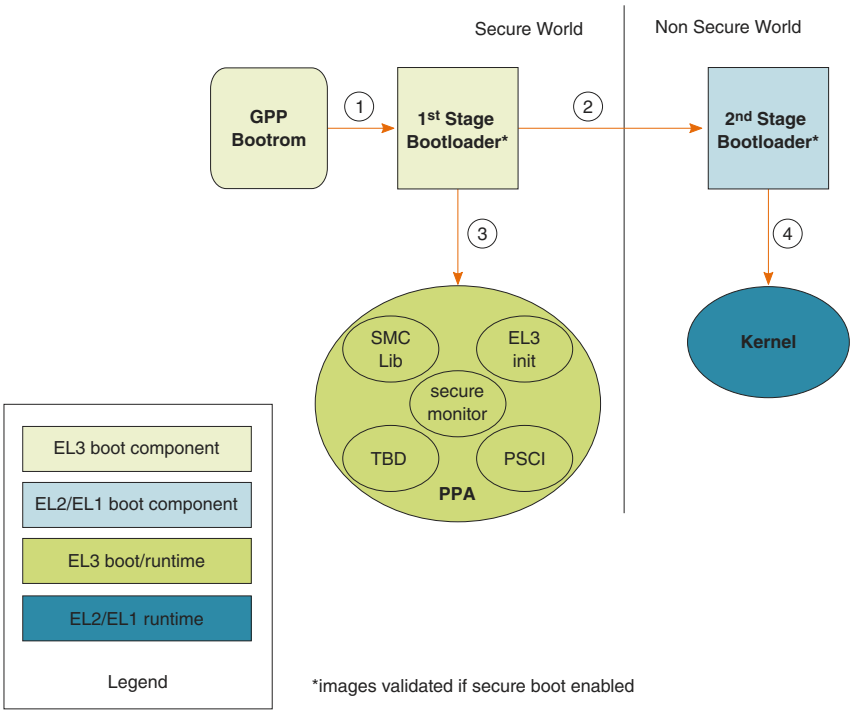
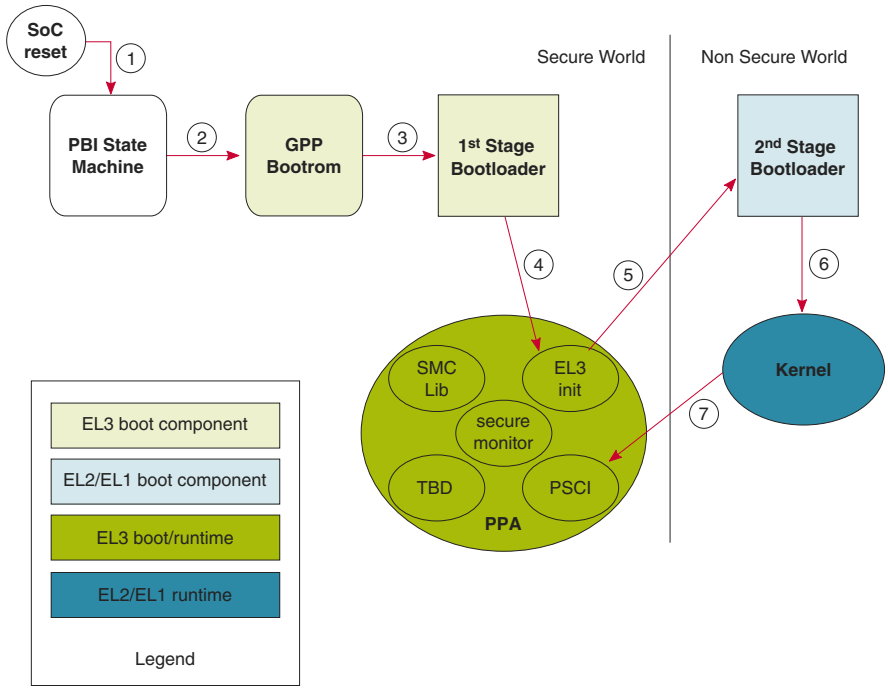
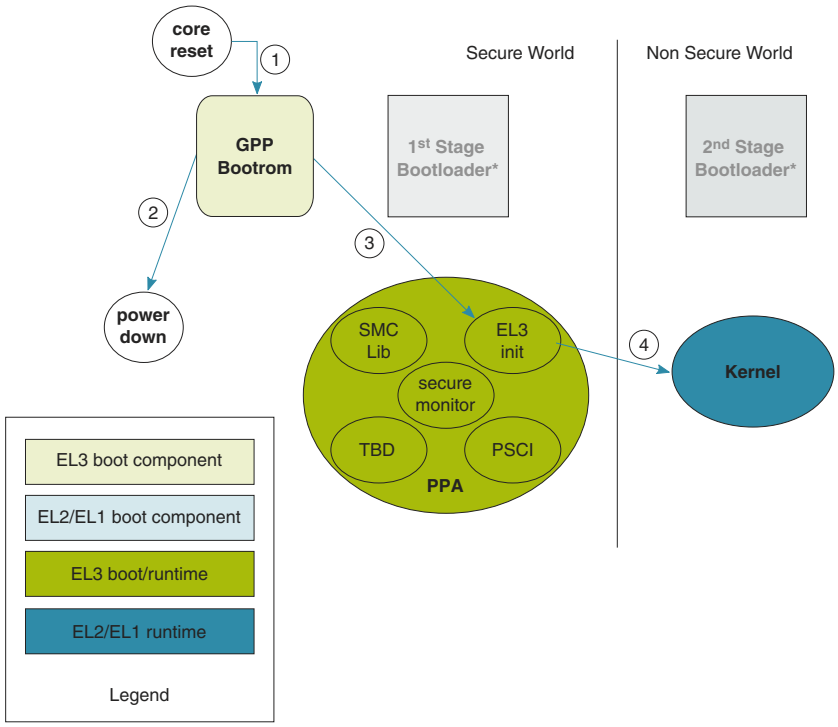
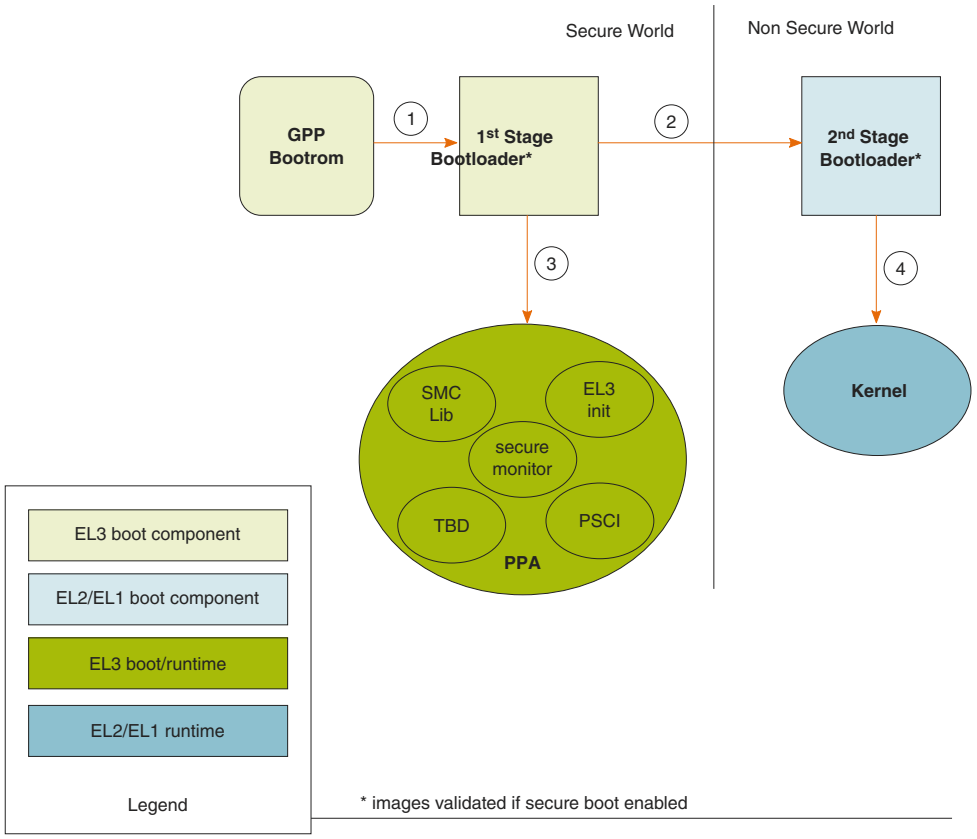
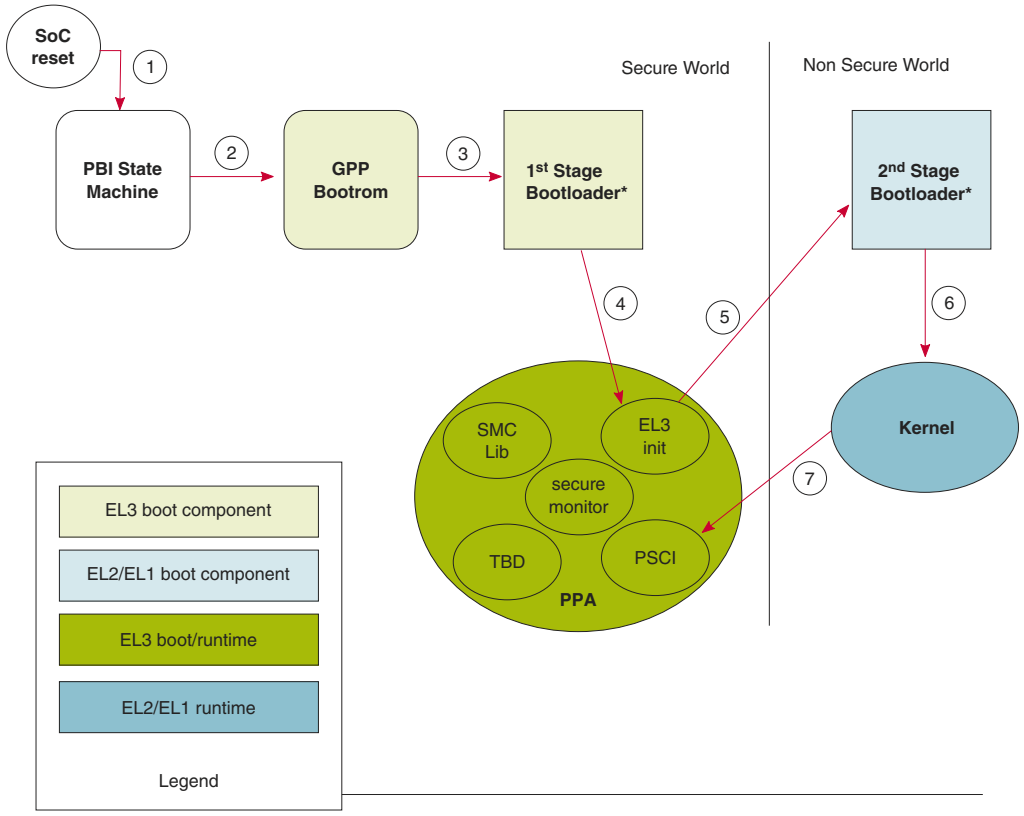
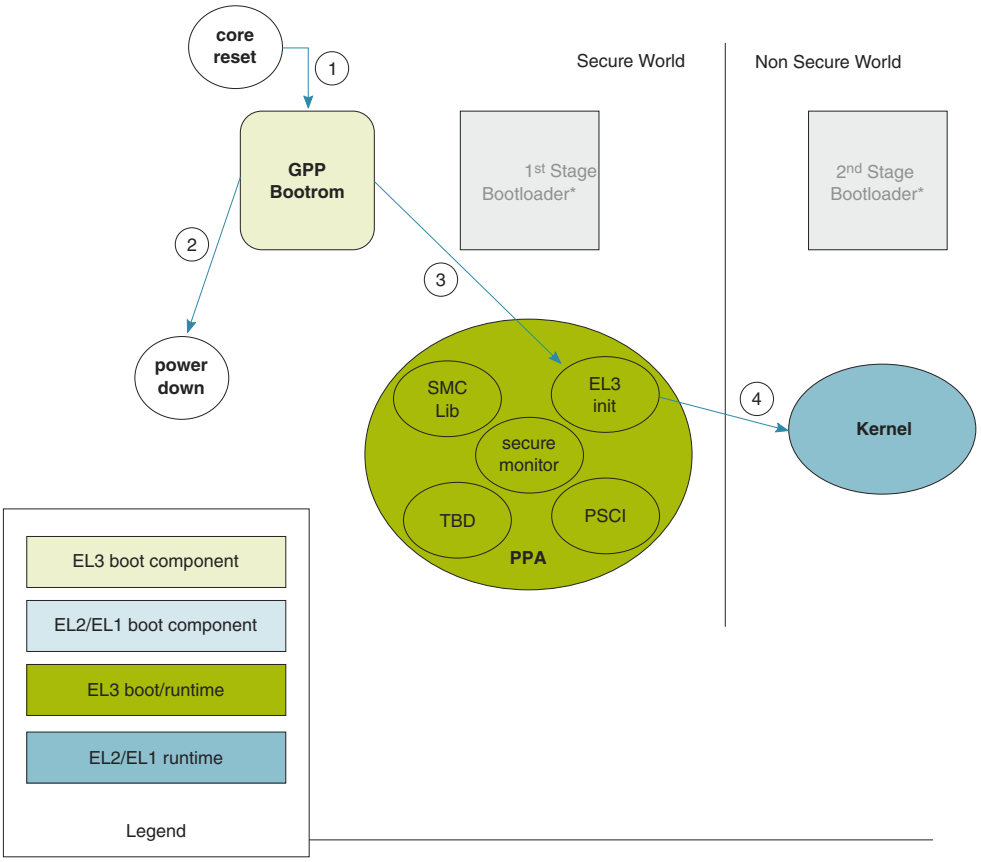
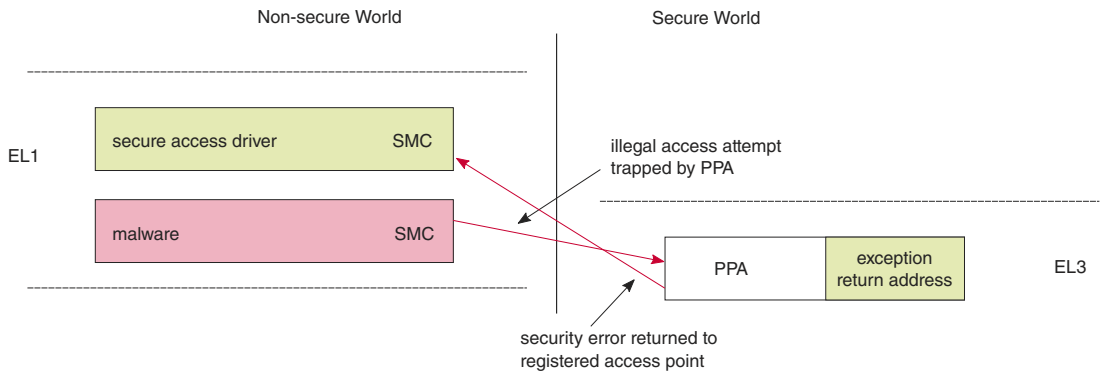
The following information will display:

TEST: msg0 performance test ......
        txbuff 0; virt addr:1020000; phy addr:27020000
Slot 0 tx buff set down.
 length(byte): 8               time(us): 2.034187       avg Gb/s: 0.031462        max Gb/s:
0.032282
length(byte): 16              time(us): 1.994464       avg Gb/s: 0.064178        max Gb/s:
0.064267
length(byte): 32              time(us): 2.004464       avg Gb/s: 0.127715        max Gb/s:
0.127892
length(byte): 64              time(us): 2.305023       avg Gb/s: 0.222124        max Gb/s:
0.222445
length(byte): 128             time(us): 2.610026       avg Gb/s: 0.392333        max Gb/s:
0.392584
length(byte): 256             time(us): 3.219476       avg Gb/s: 0.636128        max Gb/s:
0.636348
length(byte): 512             time(us): 4.136708       avg Gb/s: 0.990159        max Gb/s:
0.990558
length(byte): 1024            time(us): 6.567287       avg Gb/s: 1.247395        max Gb/s:
1.247817
length(byte): 2048            time(us): 10.818163      avg Gb/s: 1.514490        max Gb/s:
1.514918
length(byte): 4096            time(us): 19.619639      avg Gb/s: 1.670163        max Gb/s:
1.670400

● Doorbell unit performance test.

Command:
rmu> rmu -test dbell

The following information will display:

TEST: dbell performance test ......
length(byte): 2               time(us): 1.361402       avg Gb/s: 0.011753        max Gb/s:
0.012175
```

9.9.17.7 Run RMU Demo

To run the RMU application, user needs to construct the environment based on the above information.

After the board get the RMU application binary, user should first run the RMU to initialize the two message units and one doorbell unit. If success, the following information will be printed:

```
root@p4080ds:~# ./rmu
RMU: msg0 uio initialized.
     -msg0desc ...
        cell size:32bytes; entries:32
        base virt addr:1000000; base phy addr:27000000
     -msg0txbuff ...
        cell size:4096bytes; entries:32
        base virt addr:1020000; base phy addr:27020000
     -msg0rxbuff ...
```

```
        cell size:4096bytes; entries:32
        base virt addr:1040000; base phy addr:27040000
RMU: msg1 uio initialized.
     -msg1desc ...
        cell size:32bytes; entries:32
        base virt addr:1000400; base phy addr:27000400
     -msg1txbuff ...
        cell size:4096bytes; entries:32
        base virt addr:1060000; base phy addr:27060000
     -msg1rxbuff ...
        cell size:4096bytes; entries:32
        base virt addr:1080000; base phy addr:27080000
RMU: doorbell uio initialized.
     -dbellrxbuff ...
        cell size:8bytes; entries:32
        base virt addr:1000800; base phy addr:27000800
rmu>
```

Then you can implement the RMU application commands under the “rmu>” prompt.

9.9.18 USDPAA SRIO IPSec Offload User Manual

9.9.18.1 Introduction

This document describes the usage of "srio_ipsec_offload" application which is built on USDPAA PPAC architecture. User can experience the functionality of using SRIO (Serial RapidIO) and RMAN with this application. In this application, it shows how to use DPAA technology to implement high performance autonomous ipsec with sRIO-Ethernet connection.

The concept of RMAN is beyond scope of this document. The user should refer to RMAN user manual for this part.

9.9.18.2 Overview of srio_ipsec_offlaod demo

This application is based on ipsec_offlaod(eth-eth) to support srio-eth connection. In this way, it shows how to use DPAA technology to implement high performance autonomous ipsec between srio and ethernet. It mainly supports below configuration:

•IPv4 autonomous /non-autonomous

•ESP tunnel mode

•Fragmentation & reassembly

•SRIO Msg Type9

9.9.18.2.1 Srio_IPSec_offload outbound flows



Figure 1 Outbound processing

Figure 296. Outbound processing

As srio_ipsec_offload application is based on ipsec_offlaod, regarding outbound processing, Please prefer to IPSec_offload outbound flows. The main difference in srio_ipsec_offload is that encrypted packets from post-encryption OH port will be directly sent to RMAN which adds RMan message descriptor to the headroom of each packet according to configuration (Message Type 9), and then directly sends them to SRIO port 0.

9.9.18.2.2 Srio_IPSec_offload inbound flows



Figure 2 Inbound processing

Figure 297. Inbound processing

IPsec traffic (IPv4/IPv6 ESP and UDP-encap ESP) received on the inbound port srio 1 through RMAN is directly sent to pre-decryption OH port where the traffic is classified according to offloaded security associations. Frames for which a security association is found are sent to the appropriate SEC engine input queues for decryption/decapsulation; traffic for which a security association is not found is dropped. Non-IPsec traffic is directly sent to post-decryption OH port. Decrypted and non-IPsec traffic is reassembled if fragmented, and further a post IPsec classification which sends frames matching UDP destination port 2152 to USDPAA frame processing threads. Frames not matching UDP destination port 2152 have their Ethernet source and destination MAC addresses updated with Tx port MAC address and next hop MAC address respectively.

9.9.18.2.3 Limitations

The traffic load in outbound direction is not higher than 750Mbps.

This application doesn't support for restarting functionality.

9.9.18.3 Running srio_ipsec_offload

Before running this demo, please make sure some kernel options are enabled and the device tree used for this application is compiled. For more details, please refer to “Compiling the device tree and enabling kernel options”.

9.9.18.3.1 Application environment specifications

To simplify the test setup, it uses the similar environment as ipsec_offload, please refer to Application environment specifications in ipsec_offload. The application forwards traffic between one Ethernet port and two sRio ports. The one Ethernet is used for receiving clear packets, the other two srio ports process encrypted packets. Here two srio ports are connected with external line. So one srio port(srio 0) works as TX port, the other one(srio 1) as RX port. Like ipsec_offload , one port is configured as the protected interface-BP(eth port), the others as unprotected-BH(sRio port).Now it only supports on B4860QDS Platform.

9.9.18.3.2 Running srio_ipsec_offload

Srio_ipsec_offload only supports b4860qds platform at present. The rcw named N_RRSS_0x2A_0x7A is dedicated to using for this application. Before running this application, need to update this rcw in uboot. Use the steps described in Compiling the device tree for B4860 to generate a device tree binary file. Boot the board with the compiled kernel (arch/powerpc/boot/uImage) and the DTB file. To reserve memory for USDPAA and enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments: setenv othbootargs “fsl_fm_max_frm=9600" For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH="/usr/etc/srio_ipsec_offload_config_b4860.xml"
export DEF_PCD_PATH="/usr/etc/srio_ipsec_offload_pcd_b4.xml"
export DEF_SWP_PATH="/usr/etc/srio_ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
Start the application with the following commands:
/usr/bin/srio_ipsec_offload  \
-c /usr/etc/srio_ipsec_offload_config_b4860.xml\
-p /usr/etc/srio_ipsec_offload_policy.xml \
--vif macless0  --vof eth2   --vipsec eth3 -y -z --mtu-pre-enc 500
```

Notes: Interfaces vif, vof and vipsec may change due to different hardware configuration.

To disable ECN tunneling in outbound and inbound direction, following parameters are used:

```
-y Disable inbound ECN tunneling
```

```
-z Disable outbound ECN tunneling
```

The following arguments specify two Linux virtual interfaces associated with inbound and outbound Ethernet ports. These interfaces are used for adding neighboring entries used for Ethernet header update before transmission:

```
--vif virtual inbound interface index
```

```
--vof virtual outbound interface index
```

The following argument configures MTU for pre encryption fragmentation. Frames larger than this value will be fragmented prior encryption:

```
--mtu-pre-enc pre encryption MTU
```

9.9.18.3.3 Application configuration for IPsec

IPSec can be configured for offloading using setkey tool. To add an SA and two corresponding SPs add the following lines in a setkey.conf file:

```
flush;
spdflush;
add 192.168.100.1 192.168.200.1 esp 0x201
-E 3des-cbc "abcdefghipqrstuvwxyabcde"
-A hmac-sha1 "abcdefghipqrstuvwxya";
spdadd 172.16.0.1/32[any] 172.17.0.1/32[any] udp
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[any] 172.16.0.1/32[any] udp
-P in ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
Now run the command:
setkey –f setkey.conf
```

These commands create an ESP tunnel with the following endpoints: IP src 192.168.100.1 – IP dst 192.168.200.1, SPI 0x201, 3DES-CBC encryption and HMAC-SHA1 authentication. The UDP traffic coming from 172.16.0.1 going to 172.17.0.1 will be tunneled using the defined tunnel.

To add neighboring entries for Ethernet header update run the following commands: ip neigh add 192.168.0.200 lladdr <inbound port MAC address> dev macless0 ip neigh add 172.17.0.1 lladdr <next hop MAC address> dev eth2

9.9.18.3.4 Running Traffic

Assuming that the traffic is generated with a directly connected Linux box, you need to configure the Linux box to output frames for 172.16.0.1 on the connected interface (ethX):ifconfig <ethX> 172.16.0.2 netmask 255.255.0.0 up

You can use the hping tool to generate UDP packets and tcpdump to capture traffic on interface ethX. Each sent frame should be returned by the application. hping -2 -s 2150 -p 2150 -c 1 172.17.0.1

At any point you can use the following commands to show packet and byte statistics

- sa_stats <sa-id>

This command shows the statistics for a particular SA

- ipsec_stats 0

This command shows the global IPSec statistics. This includes the miss counters for the inbound and outbound direction.

9.9.18.4 Compiling the device tree and enabling kernel options

9.9.18.4.1 Compiling the device tree for B4860

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
arch/powerpc/boot/dts/fsl
cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa-srio-eth-shared-interfaces.dts
arch/powerpc/boot/dts
scripts/dtc/dtc –f –b 0 –p 1024 –I dts –O dtb –o b4860qds-usdpaa-srio_eth_shared-mac.dtb
arch/powerpc/boot/dts/b4860qds-usdpaa-srio-eth-shared-interfaces.dts
```

The DTB file b4860qds-usdpaa-srio_eth_shared-mac.dtb will be built in the current directory.

9.9.18.4.2 Enabling DPA Offloading and RMAN Drivers in the Linux Kernel

Regarding srio_ipsec_offload, it need two srio ports, some options related to them should be selected as below.

```
Device Drivers  --->
         <*> Userspace I/O drivers  --->
                    <*>   Freescale Serial RapidIO support
          [*] Staging drivers  --->
                    [*]   Freescale RapidIO Message Manager support
                    [*]   Freescale Datapath Offloading Driver
```

Chapter 10
Boot Loaders

10.1 Primary Protected Application (PPA) User's Guide

10.1.1 Introduction

10.1.1.1 Rationale and Scope

This document is the Specification and Users Guide for a loadable secure services firmware component running in TrustZone. This component, called the Primary Protected Application (PPA), has the following characteristics:

- Is loaded into the secure side of an ARM core early in the boot process
- Remains resident after boot
- Provides secure services for boot software and runtime software
- Contains a Secure Monitor, which controls access to/from the secure world
- Implements services behind a std abstract interface (published by ARM)
- Has the secure world exception vectors and handlers
- Is the focal point for implementing the Platform Security Policy

Each of these will be discussed in detail in the sections that follow. Although secure boot and other boot components such as bootloaders and bootrom will be mentioned here, this specification is not intended to exhaustively cover secure boot, bootloaders (such as UEFI and U-boot), or bootrom code.

The phrase “secure world’, used heavily in this doc, refers to ARM TrustZone, *and vice-versa*. The phrase “secure monitor” refers to the ARM v8 definition of a software entity that runs at EL3 and controls access to/from the secure world. Do not confuse “secure monitor” in this context with “security monitor”, which is a hardware component of the QorIq Trust Architecture.

There are a number of compelling reasons for having a resident secure services layer:

1. The secure services layer is first-and-foremost a focal point for implementation of a Platform Security Policy.
2. ARM cores come out of reset executing in the secure world.
3. The non-secure world needs an agent to perform tasks in the secure world
4. The PSCI interface, which is ARM’s abstract power mgmt interface, runs in the secure world.
5. A resident secure firmware can streamline bootloaders, making it easier to support multiple bootloaders.
6. The PPA is the foundation upon which a deeper TrustZone software stack can be built.

10.1.1.2 References

[1] LS1043A SoC Architecture Specification v0.3.0

[2] SMC Calling Convention, ARM Ltd, 2013

[3] ARM Architecture Reference Manual, Armv8 Edition, beta

[4] LS1043 PRL, Revision 0.7

[5] QorIQ Chassis Architecture Specification, Generation 2.1, Revision 0.8

[6] LS1 Trust Architecture, Chapter 10

[7] Layerscape Chassis Architecture Specification, Generation 3, v0.9

[8] ARM Trusted Firmware Design

[10] LS2 Boot Interfaces Programmers Guide, v1.5

[11] Power State Coordination Interface, ARM Ltd, 2012-2013

[12] QorIQ LS1046A Reference Manual, Rev C, 06/2016

10.1.1.3 Definitions

AP – Application Processor, same as GPP

ATF – ARM Trusted Firmware

Bootloader – FW that loads the OS kernel (uboot, uefi)

ESBC – External Secure Boot Code, image validation code in the bootloader

GPP – General Purpose Processor

ISBC – Internal Secure Boot Code, image validation code in the bootrom

OCRAM – On-Chip RAM

PPA – Primary Protected Application, the secure monitor and associated functions that comprise the base EL3 sw foundation

Protected – Higher-privilege sw, such as a hypervisor

PSCI – Power State Coordination Interface, an ARM std interface

Secure – SW or components that are isolated by the TrustZone architecture

Secure Monitor – the SW running at EL3 that controls the gateway from the non-secure world to the secure world

Security Monitor – a HW feature of the QorIQ Trust Architecture

SCP

SMC – an ARM instruction which generates an exception, *and* an ARM std interface based on that excepton call

SP – Service Processor, an auxiliary core that performs initial boot functions on the SoC

TPM – Trusted Platform Module, a specification of the Trusted Computing Group

Trusted Architecture (TA) – a security architecture found in the QorIq family of SoCs, including the ARM-based QorIq products

TrustZone (TZ) – an isolation context provided as part of the ARM architecture; an infrastructure for building secure subsystems

10.1.2 Boot Flow Architecture

10.1.2.1 LS1043A PPA Boot Flow

Component Load Sequence

1. GPP Bootrom loads/validates* 1st stage bootloader
2. 1st stage bootloader loads/validates* 2nd stage bootloader
3. 1st stage bootloader loads/validates* PPA
4. 2nd stage bootloader loads/validates* kernel



Figure 298. Component Load Sequence

Boot execution Order

1. Execution begins in the PBI State Machine when the SoC comes out of reset
2. After PBI, execution starts with bootcore in GPP bootrom
3. Bootcore branches to 1st stage bootloader running in EL3
4. Bootcore in 1st stage bootloader branches to EL3 init code in PPA
5. When bootcore completes EL3 init, it branches to 2nd stage bootloader in EL2
6. Bootcore in 2nd stage bootloader branches to Linux kernel in EL1
7. Kernel calls PSCI (cpu_on) to release secondary cores (LS1043A only)



Figure 299. Boot Execution Order

Secondary core execution path

1. Execution starts in the GPP bootrom when secondary core released from reset
2. If core is marked to be disabled, core enters power-down sequence in bootrom
3. Cores not disabled branch to EL3 init code in PPA
4. Upon completion of EL3 init, cores branch to start address at EL2 in kernel



Figure 300. Secondary Core Execution Path

10.1.2.2 LS1043A/LS1012A Boot Flow

Component Load Sequence

1. GPP Bootrom loads/validates* 1st stage bootloader
2. 1st stage bootloader loads/validates* 2nd stage bootloader
3. 1st stage bootloader loads/validates* PPA
4. 2nd stage bootloader loads/validates* kernel



Figure 301. Component Load Sequence

Boot execution Order

1. Execution begins in the PBI State Machine when the SoC comes out of reset
2. After PBI, execution starts with bootcore in GPP bootrom
3. Bootcore branches to 1st stage bootloader running in EL3
4. Bootcore in 1st stage bootloader branches to EL3 init code in PPA
5. When bootcore completes EL3 init, it branches to 2nd stage bootloader in EL2
6. Bootcore in 2nd stage bootloader branches to Linux kernel in EL1
7. Kernel calls PSCI (cpu_on) to release secondary cores (LS1043A only)



Figure 302. Boot Execution Order

Secondary core execution path

1. Execution starts in the GPP bootrom when secondary core released from reset
2. If core is marked to be disabled, core enters power-down sequence in bootrom
3. Cores not disabled branch to EL3 init code in PPA
4. Upon completion of EL3 init, cores branch to start address at EL2 in kernel

NOTE

LS1012A does not have secondary cores



Figure 303. Secondary Core Execution Path

10.1.2.3 LS2088A Boot Flow

Component Load Sequence

1. SP Bootrom loads/validates* 1st stage bootloader
2. 1st stage bootloader loads/validates* 2nd stage bootloader
3. 1st stage bootloader loads/validates* PPA
4. 2nd stage bootloader loads/validates* kernel



Figure 304. Component Load Sequence

Boot Execution Order

1. Execution begins in the SP bootrom when the SoC comes out of reset
2. SP bootrom completes execution, and execution starts with bootcore in GPP bootrom
3. Bootcore branches to 1st stage bootloader running in EL3
4. Bootcore in 1st stage bootloader branches to EL3 init code in PPA
5. When bootcore completes EL3 init, it branches to 2nd stage bootloader in EL2
6. Bootcore in 2nd stage bootloader branches to Linux kernel in EL1
7. Kernel calls PSCI (cpu_on) to release secondary cores



Figure 305. Boot Execution Order

Secondary Core Execution Path

1. Execution starts in the GPP bootrom when secondary core released from reset
2. If core is marked to be disabled, core enters power-down sequence in bootrom
3. Cores not disabled branch to EL3 init code in PPA
4. Upon completion of EL3 init, cores branch to start address at EL2 in kernel



Figure 306. Secondary Core Execution Path

10.1.2.4 LS1046A Boot Flow

Component Load Sequence

1. GPP Bootrom loads/validates* 1st stage bootloader
2. 1st stage bootloader loads/validates* 2nd stage bootloader
3. 1st stage bootloader loads/validates* PPA
4. 2nd stage bootloader loads/validates* kernel



Figure 307. Component Load Sequence

Boot execution Order

1. Execution begins in the PBI State Machine when the SoC comes out of reset
2. After PBI, execution starts with bootcore in GPP bootrom
3. Bootcore branches to 1st stage bootloader running in EL3
4. Bootcore in 1st stage bootloader branches to EL3 init code in PPA
5. When bootcore completes EL3 init, it branches to 2nd stage bootloader in EL2
6. Bootcore in 2nd stage bootloader branches to Linux kernel in EL1



Figure 308. Boot Execution Order

Secondary core execution path

1. Execution starts in the GPP bootrom when secondary core released from reset
2. If core is marked to be disabled, core enters power-down sequence in bootrom
3. Cores not disabled branch to EL3 init code in PPA
4. Upon completion of EL3 init, cores branch to start address at EL2 in kernel



Figure 309. Secondary Core Execution Path

10.1.3 Loading and Initializing the PPA

- The PPA must be loaded to a 64Kb boundary
- Copy the binary image to the load address – the component installing the PPA MUST be executing at EL3
- PPA should be loaded to an address in secure memory - recommend a 2MB secure region in DDR (but PPA can be tested in non-secure DDR)
- After copying the image file to DDR, clean the data cache by VA (all virtual address ranges affected by PPA load) , and invalidate the instruction cache
- The PPA initialization runs at EL3 – but the PPA transfers control back to the address loaded in BOOTLOCPTR **at EL2** – you must write the start address of the EL2 portion of your bootloader into BOOTLOCPTR *before* initializing the PPA
- After writing the EL2 start address in BOOTLOCPTR, initialize the PPA by branching to its start address

10.1.4 How to Call SMC/PSCI functions

- SMC functions obey the ARM ABI
- SMC functions treat registers x0-x12 as volatile, all others are preserved
- To call an SMC/PSCI function, load the registers according to the table below, then execute an “SMC 0x0” instruction
- If the function specifies a return value, it will be found in register x0

- Return values, including error returns, are returned in x0
- The “SMC 0x0” instruction generates an exception – after the exception is processed in the secure monitor (when the SMC function has completed), control will return to the instruction following the “SMC 0x0”
- Please refer to [2] for more details of the SMC calling convention
- Note: currently, only SMC “fast calls” are implemented (including PSCI). This means that while the calling core is in the secure world, interrupts to the core are masked.

SMC input parameters:

Register	Meaning
X0	*Function ID* (see Section 5, 6)
X1	*First optional parameter*
X2	*Second optional parameter*
X3	*Third optional parameter*

SMC return values:

Register	Meaning
X0	*Return value, Error return code*

10.1.5 PSCI Function List

Please see [11] for details of the PSCI function interface. Keep in mind that the PSCI interface is a subset of the SMC Calling Convention [2].

10.1.5.1 PSCI_VERSION

Get the Version number of this PSCI implementation.

Function ID: 0x8400_0000

Input parameters:

Register	Meaning
X0	*Function ID*
No other inputs	

Return values:

Register	Meaning
X0 bits[31:16]	*Major Version Number*
X0 bits [15:0]	*Minor Version Number*

Currently, the PSCI v0.2 specification is implemented. Thus, the *Major Version Number* returned is 0x0, and the *Minor Version Number* returned is 0x2.

10.1.5.2 CPU_ON

Release a secondary core from reset, or from the CPU_OFF state.

Function ID: 0xC400_0000

Input Parameters:

Register	**Meaning**
X0	*Function ID*
X1	*Target CPU, in MPIDR format (see [11])*
X2	*Start address (Physical)*
X3	*Context ID*

Return Values:

Register	**Return Code** (see 5.8)
X0	SUCCESS
“	INVALID_PARAMETERS
“	ALREADY_ON
“	ON_PENDING
“	INTERNAL_FAILURE

NOTE

When cores are delivered to the *Start Address*, they will be executing at EL2.

10.1.5.3 CPU_OFF

Power down the calling core.

Function ID: 0x8400_0002

Input Parameters:

Register	**Meaning**
X0	*Function ID*

Return Values:

Register	**Return Code** (see 5.8)
Function does not return if successful	
X0	DENIED

Note that this function is called on the core that is to be powered down. There is no mechanism to power down one core from another core. By definition, a power-down state is a state without retention, so the caller must save whatever state is needed when the core resumes execution.

The only way to restart a core after a call to *CPU_OFF* is with a call to *CPU_ON*.

If successful, this function does not return.

10.1.5.4 CPU_SUSPEND

Put the calling core/cluster/system into a low-power state.

Function ID: 0xC400_0001

Input Parameters (see [11]):

Register	**Meaning**
X0	*Function ID*
X1	*Power_State*
X2	*Start_Address (Physical)*
X3	*Context_Id*

Return Values:

Register	**Return Code** (see 5.8)
X0	SUCCESS
“	INVALID_PARAMETERS

Note that this function is called on the core/cluster/system that is to be suspended. There is no mechanism to suspend one core from another core. There are two available power states, *Standby* and *Power-Down* (see [11]).

For cluster low-power states, all cores of the cluster except this final core must already be in the requested power state. The function checks to see if this is the “last core standing” of the cluster – if it is the last core, the core is suspended along with the cluster. If this function is called when there is more than one active core in the cluster, the function will return with the INVALID_PARAMETERS value in x0.

Likewise for system power-down, the function checks to see if this is the last active gpp core in the SoC. If it is the last active core, then the core is suspended along with the SoC. If it is not the last active core, then the function returns with the error value INVALID_PARAMETERS in x0.

The input parameters to this function describe a complex interface. Please see [11] for details of how to use this function call.

If successful, this function does not return.

10.1.5.5 AFFINITY_INFO

Get information about a specific affinity level.

Function ID: 0xC400_0004

Input Parameters (see [11]):

Register	**Meaning**
X0	*Function ID*
X1	*Target_Affinity*
X2	*Lowest_Affinity*

Return Values:

Register	Return Codes (see 5.8, 5.8.1)
X0	ON_PENDING
“	OFF
“	ON
“	INVALID_PARAMETERS
“	NOT_PRESENT
“	DISABLED

10.1.5.6 SYSTEM_OFF

Power down the entire system.

Function ID: 0x8400_0008

Input Parameters:

Register	Meaning
X0	*Function ID*

Return Values:

The function does not return.

10.1.5.7 SYSTEM_RESET

Perform a hard reset on the entire system.

Function ID: 0x8400_0009

Input Parameters:

Register	Meaning
X0	*Function ID*

Return Values:

The function does not return.

10.1.5.8 PSCI Return Code Values

Mnemonic	Value
SUCCESS	0
NOT_SUPPORTED	-1
INVALID_PARAMETERS	-2
DENIED	-3

Table continues on the next page...

Table continued from the previous page...

ALREADY_ON	-4
ON_PENDING	-5
INTERNAL_FAILURE	-6
NOT_PRESENT	-7
DISABLED	-8

PSCI Return Codes Specific to Affinity_Info

Mnemonic	Value
ON	0
OFF	1
ON_PENDING	2

10.1.5.9 PSCI Functions Implemented, by SoC

	LS1012A	LS1043A	LS1046A	LS2088A
CPU_ON	N/A	✓	✓	✓
CPU_OFF	N/A	✓	✓	WIP
AFFINITY_INFO	✓	✓	✓	✓
CPU_SUSPEND	WIP	✓	✓	WIP
PSCI_VERSION	✓	✓	✓	✓
SYSTEM_RESET	✓	✓	✓	WIP
SYSTEM_OFF	X	X	X	X

10.1.6 SMC Function List

Please see [2] for details of the SMC function interface.

10.1.6.1 Function Count - SMC64

Return the number of functions implemented by the smc64 interface, including this function and PSCI functions using this interface.

Uses smc64 interface.

Function ID: 0xC200_FF00

Input Parameters:

Register	Meaning
X0	*Function ID*

Return Values:

Register	Value
X0	*Smc64 function count*

10.1.6.2 Function Count - SMC32

Return the number of functions implemented by the smc32 interface, including *this* function and PSCI functions using this interface.

Uses smc32 interface.

Function ID: 0x8200_FF00

Input Parameters:

Register	Meaning
X0	*Function ID*

Return Values:

Register	Value
X0	*Smc32 function count*

10.1.6.3 Get UUID

Return the 128-bit UUID uniquely identifying this SMC implementation.

Uses the smc32 interface.

Function ID: 0x8200_FF01

Input Parameters:

Register	Meaning
X0	*Function ID*

Return Values:

Register	Value
X0	Bytes [3:0] of UUID
X1	Bytes [7:4] of UUID
X2	Bytes [11:8] of UUID
X3	Bytes [15:12] of UUID

10.1.6.4 Get Revision

Return the major and minor revision numbers of the SIP portion of this SMC implementation.

Uses smc32 interface.

Function ID: 0x8200_FF03

Input Parameters:

Register	Meaning
X0	*Function ID*

Return Values:

Register	Value
X0	Major revision number of sip-smc
X1	Minor revision number of sip-smc

10.1.7 Building the PPA

In the SDK, the PPA image will be built as part of the Yocto recipe.

10.1.8 System Considerations When Calling SMC & PSCI Functions

There is always the possibility that malware will attempt to execute an SMC. The affects of this may be mitigated with three methods:

1. Insure that calling an smc/psci function can *never* corrupt the machine
2. Insure that calling an smc/psci function can *never* lower the security stance of the machine
3. Provide *only* one place in the system, a kernel driver, from which *all* smc calls are made

Items (1) & (2) mean that even if malware successfully makes an smc call, the effects of that call will not open the system up to an attack. Take careful note of item (2) – this means that any EL3 configuration that lowers the security of the system must be set in **the PPA initialization phase**, and *not* as a dynamic smc function.

Item (3) allows the PPA to determine if an unauthorized access has been attempted. Since the smc call generates an exception, the register ELR_EL3 is loaded, by hw, with the return address of the caller. If there is only one place in the system where smc calls are made from, then the PPA can be configured to register the return address. Malware, attempting to make an smc call, will have a different return address. The secure monitor in the PPA can detect this different return address, and reject the request. In addition, the secure monitor can return to the authorized return address with a security violation error. To enable this capability in later revisions of the PPA, the kernel developer should implement a kernel driver where all smc calls are made from.



Figure 310. System Considerations When Calling SMC and PSCI Functions

10.2 U-Boot

10.2.1 eSPI/SD/NAND Boot

Description

The Corenet Soc integrates a pre-boot-loader(PBL) which performs configuration registers read and write to initialize external memory devices such as I2c, eLBC FCM (NAND flash), eSDHC, or SPI interface, loads RCW and/or pre-boot initialization commands from those devices before the local cores are permitted to boot.

Specifications

Target board:	NXP CoreNet board
CPU:	NXP P2041E/P3041E/P4080E/P5020E/P5040E/T1024/T1040/T4240/T2080/B4860
Software:	U-boot 2014.01+

Introduction

The QorIQ SoC is capable of booting from eSPI/SD/NAND besides NOR boot. The bootup process can be divided into two stages:

1. PBL loads boot Image from eSPI/SD/NAND, the boot Image contains three parts, the first is RCW, the second is PBI commands, the third is the u-boot image. During the load process, PBL will do:

```
load the RCW first;
```

```
load the PBI commands to write configuration registers, e.g. to initialize SPI interface, and
configure CPC as SRAM;
```

```
load U-boot image to the CPC.
```

2. U-boot will run in the CPC and then boot up to U-Boot prompt.

Note: for T4240/T2080(with 512KB CPC) and T1024/T1040(with 256KB CPC), CPC are not big enough to store the U-boot since the U-boot size is 768KB, a two-stage boot is introduced which produces two U-boot images, the first stage u-boot is loaded to CPC, and run in CPC, which will init DDR and load the second stage u-boot to DDR, then jump to DDR to boot up to U-boot command line.

Test Procedure

The creation of such a PBL image requires:

1. a known working RCW, taken from the QorIQ SDK release.
2. the QorIQ Configuration Suite (QCS) PBL tool.

```
QCS can be downloaded from  http://www.nxp.com/QCVS
```

```
Install QCS and review its documentation before proceeding
```

3. a U-boot image for booting from eSPI flash, SD Card or NAND flash, built from the QorIQ SDK release.

Building the U-boot images

In the following example procedure the SoC is a P2041 with the RCW taken from the QorIQ SDK release.

In the SDK installation folder, go to the build folder for the chosen target machine:

```
$ cd build_p241rdb_release
$ source ./SOURCE_THIS
$ bitbake -c cleansstate u-boot
$ bitbake -f u-boot
```

This will produce all supported U-boot images for the target machine in build_p2041rdb_release/tmp/work/p2041rdb-fsl-linux/u-boot-git-r<xx>/image/boot:

```
$ ls -gG | sort
lrwxrwxrwx 1 37 Mar 18 15:41 u-boot-nand.bin -> u-boot-nand-P2041RDB_NAND-git-r27.bin
lrwxrwxrwx 1 37 Mar 18 15:41 u-boot-sd.bin -> u-boot-sd-P2041RDB_SDCARD-git-r27.bin
lrwxrwxrwx 1 39 Mar 18 15:41 u-boot.bin -> u-boot-P2041RDB_SECURE_BOOT-git-r27.bin
lrwxrwxrwx 1 40 Mar 18 15:41 u-boot-spi.bin -> u-boot-spi-P2041RDB_SPIFLASH-git-r27.bin
-rwxr-xr-x 1 524288 Mar 18 15:41 u-boot-nand-P2041RDB_NAND-git-r27.bin
-rwxr-xr-x 1 524288 Mar 18 15:41 u-boot-P2041RDB-git-r27.bin
-rwxr-xr-x 1 524288 Mar 18 15:41 u-boot-P2041RDB_SECURE_BOOT-git-r27.bin
-rwxr-xr-x 1 524288 Mar 18 15:41 u-boot-sd-P2041RDB_SDCARD-git-r27.bin
-rwxr-xr-x 1 524288 Mar 18 15:41 u-boot-spi-P2041RDB_SPIFLASH-git-r27.bin
```

Note: for T4240/T2080/T1040/T1042, the two stage boot produces two U-boot images, one is u-boot.bin, another is u-boot-spl.bin under directory spl.

```
$ ls u-boot.bin spl/u-boot-spl.bin
spl/u-boot-spl.bin  u-boot.bin
```

Generating the PBL image

Create a new QorIQ Configuration Project for SoC. Example procedure the SoC is a P2041

- In the wizard choose:

```
P2041 SoC
```

```
Silicon revision 1.0
```

```
PBL - Preboot Loader RCW Configuration Component
```

```
select the proposed RCW Hard-coded configuration
```

- Once the project is created, open a Component Inspector window for the **PBL1:PBL** component
- In the **Import** tab:

```
Select the Binary input format
```

```
Hit the Browse button to choose a QorIQ SDK release pre-build RCW
```

```
Hit the Import button
```

```
The message "Successfully Imported" appears, if the RCW contains any PBI data, it will also be
imported.
```

- In the **Properties** tab change the following properties:

```
Reset Configuration Word (RCW)->RCW Source
```

```
Boot Configuration->PBI_SRC = 0b0110 - SD/MMC (change PBI_SRC are per requirmenet, For SPI 24-
bit it is 0b0101)
```

```
Boot Configuration->BOOT_LOC = 0b10000 - Memory Complex 1
```

Note:

For T208xRDB:

PBI_SRC need to set to 0b1110 - IFC.

BOOT_LOC need to set to 0b11000 - IFC.

IFC_MODE need to set to 0b100000100 - 8-bit NAND Flash.

SD/SPI/NAND boot use the same settings.

For T2080QDS:

PBI_SRC need to set to 0b1110 - IFC.

BOOT_LOC need to set to 0b10000 - Memory Complex.

IFC_MODE need to set to 0b100000100 - 8-bit NAND Flash.

SD/SPI/NAND boot use the same settings.

NOTE

Note: The RCW bits might need to be change as per requirement or SoC. Please refer to Hardware document for details.

- Putting together the PBI data

```
Go to the PBI Data input -> PBI Data input property
```

```
Click on the Value field then on the ellipsis dots [...] button on the right
```

```
The PBI Data Input panel is now shown, if the RCW imported above has any PBI commands, they
will be shown in the panel;
```

Append the common PBI commands below into the text panel:

```
PBI Commands (Lines started with '#' is a comment):

Below are common PBI commands for P-series SoCs which have 1MB CPC.
#Clear CPC1
09010000 00200400
09138000 00000000
091380c0 00000100
#Initialize CPC1 as 1MB SRAM
09010100 00000000
09010104 fff0000b
09010f00 08000000
09010000 80000000
#Configure LAW for CPC1
09000d00 00000000
09000d04 fff00000
09000d08 81000013
#Initialize eSPI controller
09110000 80000403
09110020 2d170008
09110024 00100008
09110028 00100008
0911002c 00100008
#Configure alternate space
09000010 00000000
09000014 ff000000
09000018 81000000
#Flush data
09138000 00000000
091380c0 00000000

For T-/B-series SoCs (with 512 KB CPC SRAM) :
#Initialize CPC1
09010000 00200400
09138000 00000000
091380c0 00000100
#512KB SRAM
09010100 00000000
09010104 fff80009
09010f00 08000000
#enable CPC1
09010000 80000000
#Configure LAW for CPC1
09000d00 00000000
09000d04 fff80000
09000d08 81000012
#Initialize eSPI controller
```

```
09110000 80000403
09110020 2d170008
09110024 00100008
09110028 00100008
0911002c 00100008
#Configure alternate space
09000010 00000000
09000014 ff000000
09000018 81000000
#Flush PBL data
09138000 00000000
091380c0 00000000

For T-series SoCs like T1040 (with 256 KB CPC SRAM)
#Initialize CPC1
09010000 00200400
09138000 00000000
091380c0 00000100
#Configure CPC1 as 256KB SRAM
09010100 00000000
09010104 fffc0007
09010f00 081e000d
09010000 80000000
#Configure LAW for CPC1
09000cd0 00000000
09000cd4 fffc0000
09000cd8 81000011
#Configure alternate space
09000010 00000000
09000014 ff000000
09000018 81000000
#Configure SPI controller
09110000 80000403
09110020 2d170008
09110024 00100008
09110028 00100008
0911002c 00100008
#Flush PBL data
091380c0 000FFFFF
```

Hit the **Apply** button

NOTE

The PBI commands stated above are for reference. Ensure to change only required register bits. The register settings might change as per SoC. Hardware document should be refer for details.

- Insert the u-boot image file:

```
Select ACS File (data from binary file) then file Browse appears.
```

```
For P-series SoCs:
```

```
Change Offset: 0xf40000
```

```
Browse to File: u-boot-sd.bin
```

```
For T4240/T2080/T1040 SoCs:
Change offset: 0xfd000 (will change to 0xfd8000 if U-boot version is 2014.04+)
Browse to File: u-boot-spl.bin

Hit the Add button, then the Apply button
```

- Select and click Add for these 2 PBI commands:

```
Flush
```

```
Wait
```

```
Hit the Apply button
```

NOTE

Note: Some devices like T1040, B4860 does not support FLUSH command for SD-boot, SPI-boot. In that cases FLUSH command should be replaced with WAIT command. Refer to hardware document for details

- In **Properties** tab goto **PBL data->Output Format** property, click value field and select **Binary**.

```
From the QCS menu, click on Project -> Generate Processor Expert Code  button.
```

```
The generated file can be found in the QCS project under : Generated Code -> PBL1.bin
```

Preparing the external memory for booting

Two images must be programmed in the external memory : the PBL image and the FMan microcode image. Copy both images to the TFTP server directory, so they are available for download by U-boot.

Procedure for eSPI flash:

-
```
All operations are done on the target board in the u-boot console.

For P-series SoCs:

1) write PBL1.bin to eSPI from offset 0x0

=>tftp 100000 PBL1.bin
=>sf probe 0
=>sf erase 0 100000
=>sf write 100000 0 $filesize

For T-series SoCs:
need to write two Images, the first is the PBL1.bin produced by QCS tool, the second is the u-
boot.bin.
```

```
1.1) write PBL1.bin at offset 0:

=>tftp 100000 PBL1.bin
=>sf probe 0
=>sf erase 0 100000
=>sf write 100000 0 $filesize

1.2) write u-boot.bin at offset 0x40000 (256KB):

=>tftp 100000 u-boot.bin
=>sf write 100000 40000 $filesize


2.1) write fman ucode to eSPI from offset 0x110000

=>tftp 100000 fsl_fman_ucode_xx.bin
=>sf erase 110000 10000
=>sf write 100000 110000 $filesize

2.2) write CS4315 PHY ucode.txt to eSPI from offset 0x120000 (only for T2080RDB/T4240RDB)

=>tftp 100000 cs4315-ucode.txt
=>sf erase 120000 40000
=>sf write 100000 120000 $filesize

3) power down board
4) Change board switch configuration.

P3041/P4080/P5020/P5040: SW1[1:5] = '00101'.
P2041:                   SW1[1:5] = '10100'.
T2080/T1040:                   SW1[1:8] = '00100010', SW2[1] ='1'.

5) power on board
```

Procedure for SD card:

-

```
All operations are done on the target in the U-boot console

For P-series SoCs:

1) write PBL1.bin to SD card from offset block 8

=>tftp 100000 PBL1.bin
=>mmcinfo
=>mmc write 100000 8 block_number

2) write fman ucode to SD card from offset 1680(0x690)

=>tftp 100000 fsl_fman_ucode_xx.bin
=>mmc write 100000 690 block_number

For T-series SoCs:

need to write two Images, the first is the PBL1.bin produced by QCS tool, the second is the u-
boot.bin.

1.1)write PBL1.bin at offset block 8:
```

```
=>tftp 100000 PBL1.bin
=>mmcinfo
=>mmc write 100000 8 block_number

1.2)write u-boot.bin at offset 0x41000 (260KB), e.g. 520(0x208) blockes (For T4240QDS, the
offset is 0x200, it's changed to 0x208 in 2014.04+).

=>tftp 100000 u-boot.bin
=>mmc write 100000 208 block_number

2.1) write fman ucode to SD card from block 0x820

=>tftp 100000 fsl_fman_ucode_xx.bin
=>mmc write 100000 820 block_number

2.2) write CS4315 ucode to SD from block 0x8a0 (only for T2080RDB/T4240RDB)

=>tftp 100000 CS4315-ucode.txt
=>mmc write 100000 8a0 200

3) power down board
4) Change board switch configuration

P3041/P4080/P5020/P5040: SW1[1:5] = '00110'.
P2041:                   SW1[1:5] = '01100'.
T4240(FPGA V3 or later): SW1[1:8] = '00100000'; SW2[1] = '0'; SW7[4] = '1'.
T2080/T1040:                   SW1[1:8] = '00100000', SW2[1] = '0'.

5) power on board

Note: the block_number is HEX of ($filesize - 1) /512 + 1.
```

Procedure for NAND flash:

- ```
  All operations are done on the target in the U-boot console

  For P-series SoCs:

  1) write PBL1.bin to NAND from offset 0x0.

  =>tftp 100000 PBL1.bin
  =>nand info
  =>nand erase 0 e0000
  =>nand write 100000 0 $filesize

  2) write fman ucode to NAND from offset 0x100000

  =>tftp 100000 fsl_fman_ucode_xx.bin
  =>nand erase 100000 20000
  =>nand write 100000 100000 $filesize

  For T4240QDS/T2080QDS(NAND block size is 128KB):

  1.1) write PBL1.bin to NAND from offset 0x0.

  =>tftp 100000 PBL1.bin
  =>nand info
  =>nand erase 0 40000
  =>nand write 100000 0 $filesize
  ```
  ```

```
1.2) write u-boot.bin to NAND from offset 0x40000 (256KB).
=>tftp 100000 u-boot.bin
=>nand info
=>nand erase 40000 c0000
=>nand write 100000 40000 $filesize

2) write ucode.bin to NAND from offset 0x160000.
=>nand info
=>tftp 100000 ucode.bin
=>nand erase 160000 20000
=>nand write 100000 160000 $filesize

For T2080RDB, T104xD4RDB(NAND block size is 512KB):

1.1) write PBL1.bin to NAND from offset 0x0.

=>tftp 100000 PBL1.bin
=>nand info
=>nand erase 0 100000
=>nand write 100000 0 $filesize

1.2) write u-boot.bin to NAND from offset 0x40000 (256KB).
=>tftp 100000 u-boot.bin
=>nand write 100000 40000 $filesize

2.1) write fman ucode to NAND from offset 0x180000.
=>nand info
=>tftp 100000 fsl_fman_ucode_xx.bin
=>nand erase 180000 80000
=>nand write 100000 180000 $filesize

2.2) write CS4315 PHY ucode to NAND from offset 0x200000 (only for T2080RDB/T4240RDB).
=>nand info
=>tftp 100000 CS4315-ucode.txt
=>nand erase 200000 80000
=>nand write 100000 200000 $filesize

3) power down board
4) Change board switch configuration

P3041/P5020/P5040: SW1[1:5] = '01001'.
                   SW7[1:4] = '1001'.
p2041:             SW1[1:5] = '10010'.
T4240QDS/T2080QDS: SW1[1:8] = '10000010', SW2[1.1] = '0', SW6[1:4] = '1001'.
T2080RDB:          SW1[1:8] = '10000010', SW2[1] = '1', SW3[4] = '1'.
T1040(or personality) RDB boards:         SW1[1:8] = '10001000', SW2[1] = '0', SW3[4] = '1'.

5) power on board
```

Known Bugs, Limitations, or Technical Issues

- Provided steps for building PBL images with some reference settings. Hardware document should be refer for exact setting for particular SoC, board

Supporting Documentation

- N/A.

10.2.2 Corenet DS Boards Boot up from SRIO/PCIE User Manual

Description

For some PowerPC processors with SRIO or PCIE interface, boot location can be configured to SRIO or PCIE by RCW. The processor booting from SRIO or PCIE can do without flash for u-boot image, UCode and ENV. All the images can be fetched from another processor's memory space by SRIO or PCIE link connected between them. This UM describes the processes based on an example implemented on P4080DS platforms and RCW examples with boot from SRIO or PCIE configuration.

Specifications

Target board:	NXP Corenet DS boards (P2041/P3041/P4080/P5020/T2080/T4240/B4860)
CPU:	NXP P2041/P3041E/P4080E/P5020E/T2080/T4240/B4860
Software:	U-boot 2012.10+

Environment

Two Corenet DS boards connected with SRIO/PCIE cable, one is Master and the other is Slave. Only Master has NorFlash for booting, and all the Master's and Slave's U-Boot images, UCodes will be stored in this flash. Slave's NorFlash has been programmed only for its RCW.

Make-Config Options

Option	Description
make P4080DS_config	Compile Master U-Boot image for SRIO or PCIE Boot
make P4080DS_SRIO_PCIE_BOOT_config	Compile Slave U-Boot image for SRIO or PCIE Boot

Slave's RCW for boot from SRIO or PCIE

All cores in holdoff and Slave will boot from SRIO port 1	All cores in holdoff and Slave will boot from PCIE port 1
00000000: aa55 aa55 010e 0100 0c58 0000 0000 0000 00000010: 1818 1818 0000 8888 7440 4000 0000 2000 00000020: f440 0000 0100 0000 0000 0000 0000 0000 00000030: 0000 0000 0083 0000 0000 0000 0000 0000 00000040: 0000 0000 0000 0000 0813 8040 063c 778f	00000000: aa55 aa55 010e 0100 0c58 0000 0000 0000 00000010: 1818 1818 0000 8888 1440 4000 0000 2000 00000020: f040 0000 0100 0000 0020 0000 0000 0000 00000030: 0000 0000 0083 0000 0000 0000 0000 0000 00000040: 0000 0000 0000 0000 0813 8040 547e ffc9

Test Procedure

- **Update RCW for Slave with boot from SRIO or PCIE port 1 and all cores in holdoff configurations.**

  We can update the RCW in Slave's NorFlash for the test, and should ensure the Slave can get RCW from its local NorFlash. For the boards can be booted up from vbank 0, vbank 2 or bank 4, we can update the RCW in the vbank 2 or vbank 4 and implement the SRIO/PCIE boot in vbank 2 or vbank 4.

  Note: Currently just B4860 platform should be implemented from vbank2, and all others should be implemented from vbank4.

- **Update Master's U-Boot image compiled normally.**

  Update this U-Boot image for Master's vbank 2 or vbank 4 boot space.

- **Program Slave's U-Boot image, UCode and ENV parameters into master's NorFlash.**

Slave's U-Boot image should be compiled with the "make P4080DS_SRIO_PCIE_BOOT_config". Based on the configurations in file "include/configs/corenet_ds.h" or "include/configs/B4860QDS.h" or other relative header files, we should write all the Slave's images from these addresses into Master's NorFlash:

Write Slave's U-Boot image from 0xeb240000 (P4080DS) or 0xed240000 (B4860QDS) in Master's NorFlash when Master has been started up in vbank 0 environment.

```
P4080DS:

tftp 1000000 tftpboot/u-boot.bin.slave;erase eb240000 eb2fffff;cp.b 1000000 eb240000 c0000

B4860QDS:

tftp 1000000 tftpboot/u-boot.bin.slave;erase ed240000 ed2fffff;cp.b 1000000 ed080000 c0000
```

Write Slave's UCode from 0xeb100000 (P4080DS) or 0xed100000 (B4860QDS) in Master's NorFlash when Master has been started up in vbank 0 environment.

```
The size of the UCode is 64k bytes.
```

Write Slave's ENV from 0xeb120000 (P4080DS) or 0xed120000 (B4860QDS) in Master's NorFlash when Master has been started up in vbank 0 environment.

```
The size of the ENV is 128k bytes.
```

- **Set environment variable "bootmaster" to "SRIO1" or "PCIE1" and save this variable for master's vbank 2 (B4860QDS) or vbank 4 (P4080DS) boot space.**

```
        setenv bootmaster SRIO1

      or

        setenv bootmaster PCIE1

        saveenv
```

- **Power down the Master and Slave boards after all the configurations described above.**

  When set all cores in holdoff by RCW, we should power on the Slave board first, and it will implement the configurations based on the RCW and then wait to be released by Master.

- **Power on the Slave board and switch to the vbank 4 (P4080DS) or vbank 2 (B4860QDS), the board will wait to be released because of all the cores in holdoff.**

- **Power on the Master board and switch to the vbank 4 (P4080DS) or vbank 2 (B4860QDS), it will finish some necessary configurations and then release Slave's core 0.**

  Master will boot up normally from its NorFlash. Then it will finish necessary configurations for slave's boot from SRIO or PCIE port 1 and release Slave's core 0 by SRIO or PCIE interface. When the core 0 is released, Slave will fetch U-Boot image, UCode and ENV from Master's NorFlash by SRIO or PCIE interface.

  If has opened the debug messages, Master will give some information about the SRIO or PCIE boot when starting up:

```
=> pix altbank

U-Boot 2011.09-01225-g5c484d3-dirty (Dec 31 2011 - 17:04:51)

CPU0:  P4080E, Version: 2.0, (0x82080020)
```

```
Core:  E500MC, Version: 2.0, (0x80230020)

Clock Configuration:

       CPU0:1199.988 MHz, CPU1:1199.988 MHz, CPU2:1199.988 MHz, CPU3:1199.988 MHz,

       CPU4:1199.988 MHz, CPU5:1199.988 MHz, CPU6:1199.988 MHz, CPU7:1199.988 MHz,

       CCB:599.994 MHz,

       DDR:599.994 MHz (1199.988 MT/s data rate) (Asynchronous), LBC:74.999 MHz



.........



SRIO1: enabled

SRIO2: enabled

SRIOBOOT - MASTER: Master port [ 1 ] for srio boot.

SRIOBOOT - MASTER: Inbound window1 for slave's image; Local = 0xfef080000, Srio = 0xfff80000,
Size = 0x80000

SRIOBOOT - MASTER: Inbound window2 for slave's image; Local = 0xfef080000, Srio = 0x3fff80000,
Size = 0x80000

SRIOBOOT - MASTER: Inbound window for slave's ucode; Local = 0xfef020000, Srio = 0x3ffe00000,
Size = 0x10000

SRIOBOOT - MASTER: Inbound window for slave's ENV; Local = 0xfef060000, Siro = 0x3ffe20000,
Size = 0x20000

SRIOBOOT - MASTER: Check the port status and release slave core ...

SRIOBOOT - MASTER: Port [ 1 ] is ready, now release slave's core ...

SRIOBOOT - MASTER: Release slave successfully! Now the slave should start up!

MMC:  FSL_ESDHC: 0



.........
```

Important Note

- For the Master board, do not need to rebuild an image when change the SRIO or PCIE port for boot from SRIO/PCIE, just using the same U-Boot image and set the corresponding value to "bootmaster" based on the SRIO or PCIE port used for the boot. For example, you can set and save the environment variable "bootmaster" with "SRIO1", "SRIO2" or "PCIE1", "PCIE2", "PCIE3". And in addition, do not forget to update the RCW for Master based on your selection.
- For the Slave board, do not need to rebuild an image when change the SRIO or PCIE port for boot from SRIO/PCIE, just using the same U-Boot image and rewrite the new RCW with selected port, then the code will get the port information by reading new RCW and configure the port automatically.

Known Bugs, Limitations, or Technical Issues

- Because the Slave cannot erase, write master's Norflash by SRIO or PCIE interface, so it also cannot modify the ENV parameters stored in Master's Norflash using "saveenv" or other commands.
- It is very important to ensure that all master and slave RapidIO or PCIE ports are correct. If the SRIO link has been initialized and worked well, then reboot master or slave will result in an error state of the partner's RapidIO port. This error is most likely caused by the mismatching of the AckIDs on the both sides of the link, and will result in a link failure.

  In order to make the SRIO link recover and work well:

  1. If master and slave are directly connected, when reboot one, the other must also be rebooted.

  2. If master and slave are indirectly connected, for example switch chip is used in the RapidIO system, when reboot one, another must be responsible for the link back to normal, or reboot the entire system.

  PCIE link has no this issue.
- For the P2041RDB board, the SRIO lanes should be reconfigured to right slot by CPLD commands; But when the slave is in hold-off mode, we cannot implement those reconfigurations. So the P2041RDB cannot support Boot from SRIO.

Supporting Documentation

- doc/README.srio-pcie-boot-corenet

10.3 Secure Boot

Refer to the table below to identify the Secure Boot process that corresponds to your SoC:

Secure Boot Process	SoC		Trust Arch. Version
Hardware PreBoot Loader (PBL) Based Platforms	**PowerPC**	P4080	1.0
		P3041	1.1
		P5020	1.1
		P5040	1.1
		T4240	2.0
		B4860	2.0
		T2080	2.0
		T1040	2.0
		T1023	2.0
	ARM	LS1020A/LS1021A	2.1
		LS1043A	2.1
		LS1046A	2.1
		LS1012A	2.1
Software-PreBoot Loader (PBL) Based Platforms	P1010		1.0
	BSC913x		1.1

Table continues on the next page...

Table continued from the previous page...

	C29x	2.0
Service Processor (SP) Based Platforms	LS2088A	3.1

10.3.1 Hardware PreBoot Loader (PBL) Based Platforms

10.3.1.1 Introduction

This document is intended for end-users to demonstrate the image validation process. The image validation can be split into stages, where each stage performs a specific function and validates the subsequent stage before passing control to that stage. In the example, the ESBC is NXP* provided reference code referred to as ESBC uboot.

Chain of Trust

ESBC uboot performs minimal SoC configuration before validating the Next Executable using the same CSF header format as the ISBC used to validate ESBC Uboot. The CSF Header and signature are added to the Next Executable using the NXP Code Signing Tool.

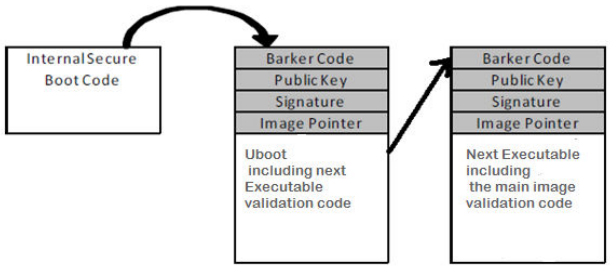


Figure 311. Chain of Trust

Chain of Trust with confidentiality

The validated ESBC uboot image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Cryptographic blob mechanism is used to establish Chain of trust with confidentiality.

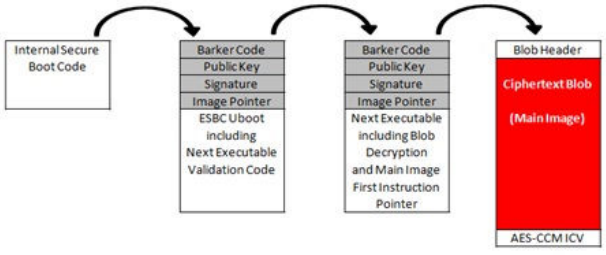


Figure 312. Chain of Trust with confidentiality

This document provides more details on the secure boot flow, ISBC, ESBC and NXP Code signing tool.

NOTE

NXP and Freescale may be used synonymously in this document. Registers, register file names, and Trust Architecture software may not be updated to NXP for an extended period of time.

10.3.1.2 Secure boot Process

Secure boot process uses a digital signature validation routine already present in INTERNAL BOOT ROM. This routine performs validation using HW bound RSA public key to decrypt the signed hash and compare it to a freshly calculated hash over the same system image. If the comparison passes, the image can be considered as authentic.

The complete process can be broken down into following phases:

- Pre Boot Phase
  1. PBL
  2. SFP
- ISBC
- ESBC

The Complete Secure boot Process is shown in the Figure below.

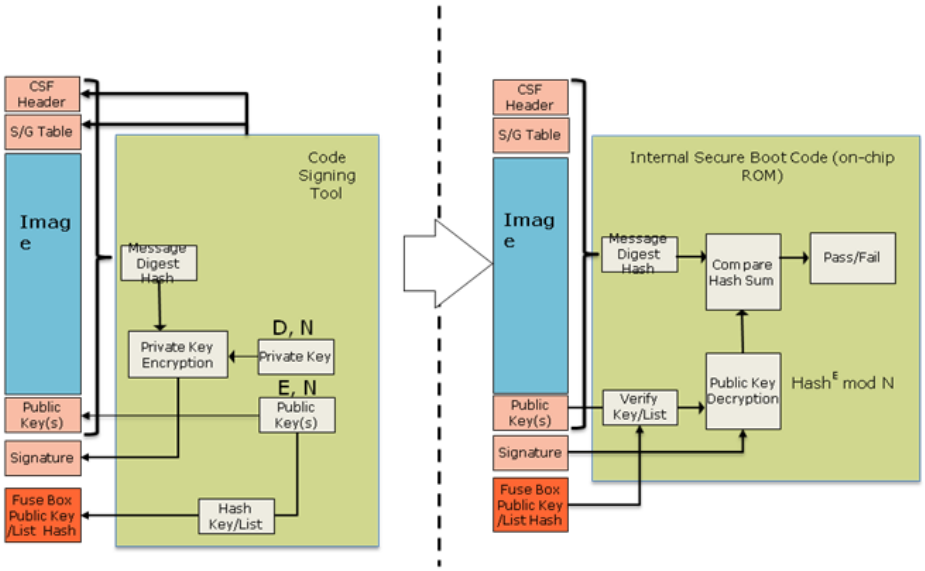


Figure 313. Secure Boot Process

10.3.1.3 Pre-Boot Phase

When the processor is powered on, reset control logic blocks all device activity (including scan and debug activity) until fuse values can be accurately sensed. The most important fuse value at this stage of operation is the 'Intent to Secure' (ITS) bit. When an OEM sets ITS, they intend for the system to operate in a secure and trusted manner.

The two main components involved during this process are :

The security fuse processor (SFP) has two roles. The first is to physically burn fuses during device provisioning. The second is to use these provisioned values to enforce security policy in the pre-boot phase, and to securely pass provisioned keys and other secret values to other hardware blocks when the system is in a trusted/secure state.

PreBoot Loader (PBL) is the micro-sequencer that can simplify system boot by configuring the DDR memory controllers to more optimal settings and copying code and data from low speed memory into DDR. This allows subsequent phases of boot to operate at higher speed. The setting of ITS determines where the PBL is allowed to read and write. The use of the PBL is mandatory when performing secure boot. At a minimum, the PBL must read a command file from a location determined by the Reset Configuration Word (RCW) and perform a store of a value to the ESBC Pointer Register within the SoC. If the PBL doesn't perform this operation (or sets the ESBC pointer to the wrong value), the ISBC will fail to validate the ESBC. Once the PBL has completed any operations defined by its command file, the PBL is disabled until the next Power on Reset and the Boot Phase begins.

Some example PBI commands used in the demo are given below. The commands are embedded in the RCW's mentioned in the SDK Images required for the demo

NOR SECURE BOOT

- **P3/P4/P5**

```
#LAW for ESBC
    09000cd0 00000000
    09138000 00000000 (Flush command)
    09000cd4 c0000000
    09138000 00000000  (Flush command)
    09000cd8 81f0001d
    09138000 00000000(FLUSH command)
# Scratch Register
    090e0200 c0b00000
```

- **T1/T2/T4/B4**

```
#LAW for ESBC
    09000c10 00000000
    09000c14 c0000000
    09000c18 81f0001b
# LAW for CPC/SRAM
    09000d00 00000000
    09000d04 bff00000
    09000d08 81000013
# Scratch Registers
    090e0200 c0b00000
    090e0208 c0c00000
# CPC SRAM
    09010100 00000000
    09010104 bff00009
# CPC Configuration
    09010f00 08000000
    09010000 80000000
```

NAND SECURE BOOT

- **P3/P5**

```
# SCRATCH REGISTER
    090e0200 bff00000
    09138000 00000000 (Flush Command)
# CPC1 SRAM
    09010000 00200400
```

```
    09010100 00000000
    09010104 bff0000b
    09010f00 08000000
    09010000 80000000
    09138000 00000000 (Flush Command)
# LAW for CPC/SRAM
    09000d00 00000000
    09000d04 bff00000
    09000d08 81000013
    09138000 00000000 (Flush Command)
# Alternate Configuration Space Configuration
    09000010 00000000
    09000014 bf000000
    09000018 81000000
    09138000 00000000 (Flush Command)
# CPC2 Cache
    09110000 80000403
    09110020 2d170008
    09110024 00100008
    09110028 00100008
    0911002c 00100008
    09138000 00000000 (Flush Command)

/* hdr_uboot.out and u-boot.bin must also be loaded on NAND
 * ALT_CONFIG_WRITE command must be used for the same.
 * Starting offset for ALT_CONFIG_WRITE command would be
 * hdr_uboot.out - 0xf00000
 * u-boot.bin    - 0xf40000
 */
```

The ISBC is capable of reading from NOR flash connected to the Local Bus, on-chip memory configured as SRAM, or main memory. Unless the ESBC is stored in NOR flash, the developer is required to create a PBL Image that copies the image to be validated from NVRAM to main memory or internal SRAM prior to writing the SCRATCHRW1 Register and executing the ISBC code.

To assist with the creation of PBL Images (for both normal and Trust systems), NXP offers a PBL Image Tool.

Note that it is possible for an attacker to modify the board to direct the PBL to the wrong non-volatile memory interface, or change the PBL Image and CSF Header pointer, however this will result in a secure boot failure and the system remaining in an idle loop indefinitely.

10.3.1.4 ISBC Phase

10.3.1.4.1 Flow in the ISBC Code

With the PBL disabled and all external masters blocked by the PAMUs, CPU 0 is released from boot hold-off and begins executing instructions from a hardwired location within the Internal BootROM. The instructions inside the Internal BootROM are NXP developed code known as the Internal Secure Boot Code (ISBC). The ISBC leads CPU 0 to perform the following actions:

1. **Who am I check?** - CPU 0 reads its Processor ID Register, and if it finds any value besides physical CPU 0, the CPU enters a loop. This insures that only CPU 0 executes the ISBC.
2. **Sec_Mon check** - CPU 0 confirms that the Sec_Mon is in the Check state. If not, it writes a ‘fail’ bit in a Sec_Mon control register, leading to a state transition.
3. **ESBC pointer read** - CPU 0 reads the ESBC Pointer Register, and then reads the word at the indicated address, which is the first word of the Command Sequence File Header which precedes the ESBC itself. If the contents of the word don’t match a hard coded preamble value, the ISBC takes this to mean it has not found a valid CSF and cannot proceed. This leads to a fail, as described in #2 above.

4. **CSF parsing and public key check** - If CPU 0 finds a valid CSF header, it parses the CSF header to locate the public key to be used to validate the code. There can be a single public key or a table of 4 public keys present in the header. The Secure Fuse Processor doesn't actually store a public key, it stores a SHA-256 hash of the public key/table of 4 keys. This is done to allow support for up to 4096b keys without an excessively large fuse block. If the hash of the public key fails to match the stored hash, secure boot fails.
5. **Signature validation** - With the validated public key, CPU 0 decrypts the digital signature stored with the CSF header. The ISBC then uses the ESBC lengths and pointer fields in the CSF header to calculate a hash over the code. The ISBC checks that the CSF header is included in the address range to be hashed. Option flags in the CSF header tell the ISBC whether the NXP Unique ID and the OEM Unique ID (in the Secure Fuse Processor) are included in the hash calculation. Including these IDs allows the image to be bound to a single platform. If the decrypted hash and generated hash don't match, secure boot fails.
6. **ESBC First Instruction Pointer check** - One final check is performed by the ISBC. This check confirms that the First Instruction Pointer in the CSF header falls within the range of the addresses included in the previous hash. If the pointer is valid, the ISBC writes a 'PASS' bit in a Sec_Mon command register, the state machine transitions to 'Trusted', and the OTPMK is made available to the SEC.
7. In case of failure , for Trust v2.0 devices , secondary flag is checked in the CSF header. If set, ISBC reads the CSF header pointer form SCRATCHRW3 location and repeats from step 4.

There are many reasons the ISBC could fail to validate the ESBC. Technicians with debug access can check the SCRATCHRW2 Register to obtain an error code. For a list of error codes refer ISBC Validation Error Codes.

10.3.1.4.2 Super Root keys (SRKs) and signing keys

These are RSA public and private key pairs. Private keys are used to sign the images and public keys are used to validate the image during ISBC and ESBC phase.

Public keys are embedded in the header and the hash of srk table is fused in SRKH register of SFP.

These are Hardware Bound Keys, once the hash is fused the public private key pair can't be modified.

Keys of sizes 1k, 2k and 4k are supported in FSL Secure Boot Process.

It is the OEM's responsibility to tightly control access to the RSA private signature key. If this key is ever exposed, attackers will be able to generate alternate images that will pass secure boot.

If this key is ever lost, the OEM will be unable to update the image.

10.3.1.4.3 Key Revocation

Trust Architecture 2.x introduces support for revoking the RSA public keys used by the ISBC to verify the ESBC. The RSA public keys used for this purpose are called super root keys.

OEM can use either a single key or a list of upto 4 super root keys in the Trust Arch v2.x devices.

In the NXP Code Signing Tool (CST), the OEM defines whether the device uses a single super root key, or offers a list of super root keys. If using a single super root key, a new flag bit in the CSF header will indicate "Key", otherwise the flag will indicate "Key List". Assuming key list, the OEM can populate a list of up to 4 super root keys for trust arch v2.x onwards platforms . And calculates a SHA-256 hash over the list. This hash is written to the SRKH registers in the SFP.

As part of code signing, the OEM defines which key in the key list is to be used for validating the image. This key number is included as a new field in the CSF header.

During secure boot, the ISBC determines whether a key list is in use. If the key list is valid, the ISBC checks the key number indicated in the CSF header against the revocation fuses in the SFP's OEM Security Policy Register (SFP_OSPR). If the key is revoked, the image validation fails.

NOTE

In order to prevent unauthorized revocation of keys, SFP provides a bit (Write Disable). If the bit is set, the Key revocation bits cannot be written to.

In regular operation, the ESBC (early Trusted S/W) needs to set the SFP Write Disable bit. When circumstances call for revoking a key, the OEM will use an ESBC image with “Write Disable” bit not set. So, the SFP will be in a state in which key revocation fuses can be set.

Logically after revoking the required key(s), the OEM would then load a new signed ESBC image with code to set the "Write Disable" bit, with new CSF header indicating which of the remaining non-revoked key to use.

So, only the possessor of a legitimate RSA private key can enable key revocation.

One possible motivation for an OEM to revoke a super root key is the loss of the associated RSA private key to an attacker. If the attacker has gained access to a legitimate RSA private key, and the attacker can turn on power to the fuse programming circuitry, then the attacker could maliciously revoke keys. To prevent this from being used to permanently disable the system, one super root key does not have an associated revocation fuse.

10.3.1.4.4 Alternate Image Support

Trust 2.0 onwards will support a primary and alternate image, where failure to find a valid image at the Primary location will cause the ISBC to check a configured alternate location.

To execute, the alternate image must be validated using a non-revoked public key as defined by its CSF Header. A valid alternate image has same rights and privileges as a valid primary image.

This feature helps to reduce risk of corrupting single valid image during firmware update or as a result of Flash block wear-out.

To enable this feature, create PBI with pointers for both Primary and Alternate Images (HW PBL uses SCRATCHRW1 & SCRATCHRW3).

10.3.1.4.5 ESBC with CSF Header

ESBC is the generic name for the code that the ISBC validates. A few ESBC scenarios are described in later sections.

The figure below provides an example of an ESBC with CSF (Command Sequence File) Header. The CSF Header includes lengths and offset which allow the ISBC to locate the operands used in ESBC image validation, as well as describe the size and location of the ESBC image itself.

Note: CSF Header and ESBC Header may be used synonymously in this and other NXP Trust Architecture documentation.

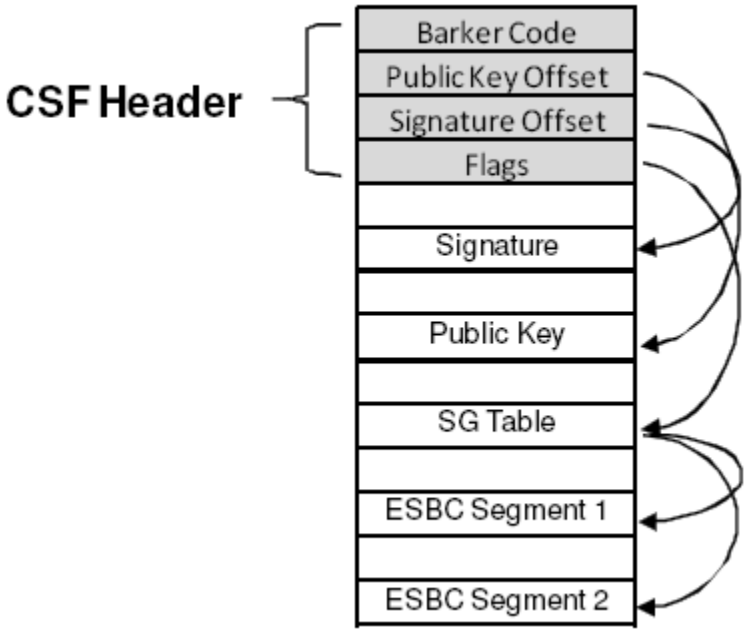


Figure 314. ESBC with CSF Header

10.3.1.5 ESBC Phase

Unlike the ISBC, which is in an internal ROM and therefore unchangeable, the ESBC is NXP-supplied reference code, and can be changed by OEMs. The remainder of this section is the description of a reasonable secure boot chain of trust based on NXP's reference software for secure boot. Depending on the requirement, ESBC can be a monolithic image - including uboot, device trees, boot firmware, drivers along with the OS and applications or can be mini-uboot.

NXP provided ESBC consists of standard u-boot which has been signed using a private key. U-boot reserves a small space for storing environment variables. This space is typically one sector above or below the u-boot and is stored on persistent storage devices like NOR flash if macro CONFIG_ENV_IS_IN_FLASH is used. In case of secure boot, macro CONFIG_ENV_IS_NOWHERE is used and so, environment is compiled in uboot image and is called default environment. This default environment can't be stored on flash devices. User won't be able to edit this environment also as he can't reach to uboot prompt in case of secure boot. There is default boot command for secure boot in this default environment which executes on autoboot.

ESBC validates a file called boot script and on successful validation execute the commands in the boot script.

There are many reasons ESBC could fail to validate Client images or boot script. The error status message along with the code is printed on the u-boot console. For a list of error codes refer ESBC Validation Error Codes.

Users are free to use NXP ESBC as it is provided or to use it as reference to modify their own secure boot system.

NOTE

On Soc's with ARMv8 core (eg:- LS1043, LS1046,LS1012), during ISBC phase in Internal Boot ROM, SMMU (which by default is in by-pass mode) is configured to allow only secure transactions from CAAM.

The security policy w.r.t. SMMU in ESBC phase must be decided by the user/customer. So, currently in ESBC (U-Boot), SMMU is configured back to by-pass mode allowing all transactions (secure as well as non-secure).

10.3.1.5.1 Boot script

Bootscript is a U-Boot script image which contains u-boot commands. ESBC would validate this boot script before executing commands in it.

NOTE

1. Boot script can have any commands which u-boot supports. No checking on the allowed commands in boot script. Since it is validated image, assumption is that commands in boot script would be correct.
2. If some basic scripting error done in boot script like unknown command, missing arguments, the required usage of that command and core is put in infinite loop.
3. After execution of commands in boot script, if control reaches back in u-boot, error message would be printed on u-boot console and core would be put in spin loop by command esbc_halt.
4. Scatter gather images not supported with validate command.
5. If ITS fuse is blown, any error in verification of the image would result in system reset. The error would be printed on console before system goes for a reset.

10.3.1.5.1.1 Where to place the boot script?

NXP's ESBC u-boot expects the boot script to be loaded in flash as specified in address map for different platforms under the topic 'Appendix <platform> Secure Boot Demo'. ESBC u-boot code assumes that the public/private key pair used to sign the boot script is same as that was used while signing the u-boot image. If user used different key pair to sign the image, hash of the N and E component of the key pair should be defined in macro:

CONFIG_BOOTSCRIPT_KEY_HASH.

Note - The hash defined should be hex value, 256 bits long.

Both the above macros can be defined or changed in the configuration file secure_boot.h at the following location in u-boot code:

`u-boot/arch/powerpc//include/asm/fsl_secure_boot.h`

Two new commands called esbc_validate and esbc_halt have been added in NXP ESBC u-boot.

Two more commands are present, 'blob enc' and 'blob dec' for running chain of trust with confideniality.

10.3.1.5.1.2 Chain of Trust

Boot script contains information about the next level of images, e.g. Linux, HV, etc. ESBC validates these images as per their public keys and then executes bootm command to pass-on the control to next image.

Users are free to use NXP ESBC as it is provided or to use it as reference to modify their own secure boot system.

Figure below shows the Chain of trust established for Validation with this ESBC u-boot.

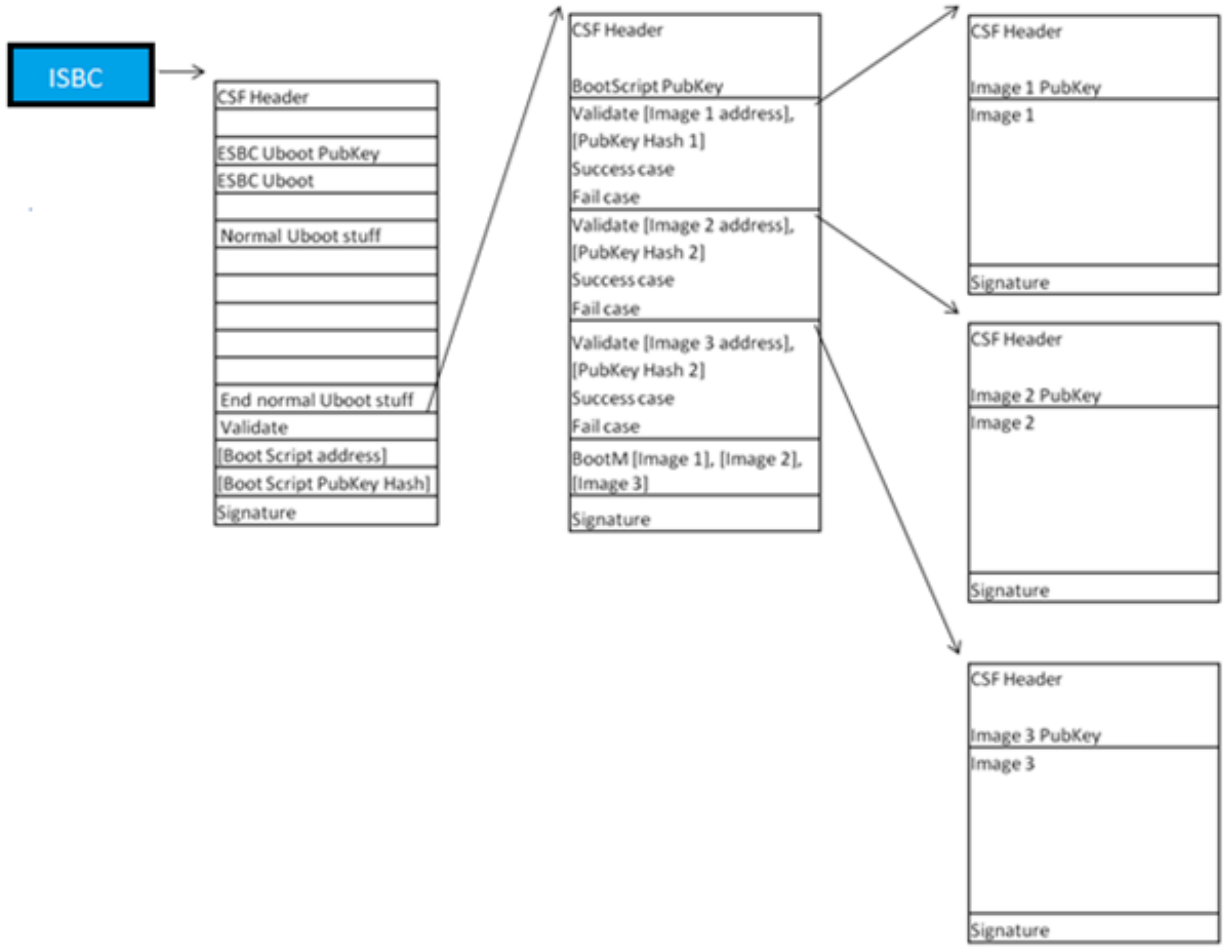


Figure 315. Secure boot flow (Chain of Trust)

10.3.1.5.1.2.1 Sample Boot Script

A sample boot script would look like:

```
...
esbc_validate <Img1 header addr> <pub_key hash>
esbc_validate <Img2 header addr> <pub_key hash>
esbc_validate <Img3 header addr> <pub_key hash>
...
bootm <img1 addr> <img2 addr> <img3 addr>
```

10.3.1.5.1.2.1.1 esbc_validate command

esbc_validate img_hdr [pub_key_hash]

Input arguments:

`img_hdr` - Location of CSF Header of the image to be validated

`pub_key_hash` - hash of the public key used to verify the image. This is optional parameter. If not provided, code makes the assumption that the key pair used to sign the image is same as that used with ISBC. So the hash of the key in the header is checked against the hash available in SRK fuse for verification.

Description:

The command would do the following:

- Perform CSF header validation on the address passed in the image header. During parsing of the header, image address in stored in an environment variable which is later used in source command in default secure boot command.
- Signature checks on the image

10.3.1.5.1.2.1.2 esbc_halt command

esbc_halt (no arguments)

Description:

The command would do the following:

This command puts core in spin loop.

After successful validation of images, bootm command in bootscript should execute and control should never reach back to uboot. If somehow, control reaches back to uboot (eg. bootm not present in bootscript), core should just spin.

10.3.1.5.1.3 Chain of Trust with Confidentiality

To establish chain of trust with confidentiality, cryptgraphic blob mechanism can be used. In this chain of trust, validated image is allowed to use the One Time Programmable Master Key to decrypt system secrets.

Two bootscripts are to be used. First encap bootscripts is used which creates a blob of the LINUX images and saves them. After this the system is booted after replacing the encap bootscript with decap bootscript which decapsulates the blobs and boot the LINUX with the images.

Figures below show the Chain of trust with confidentiality (Encapsulation and Decapsulation).

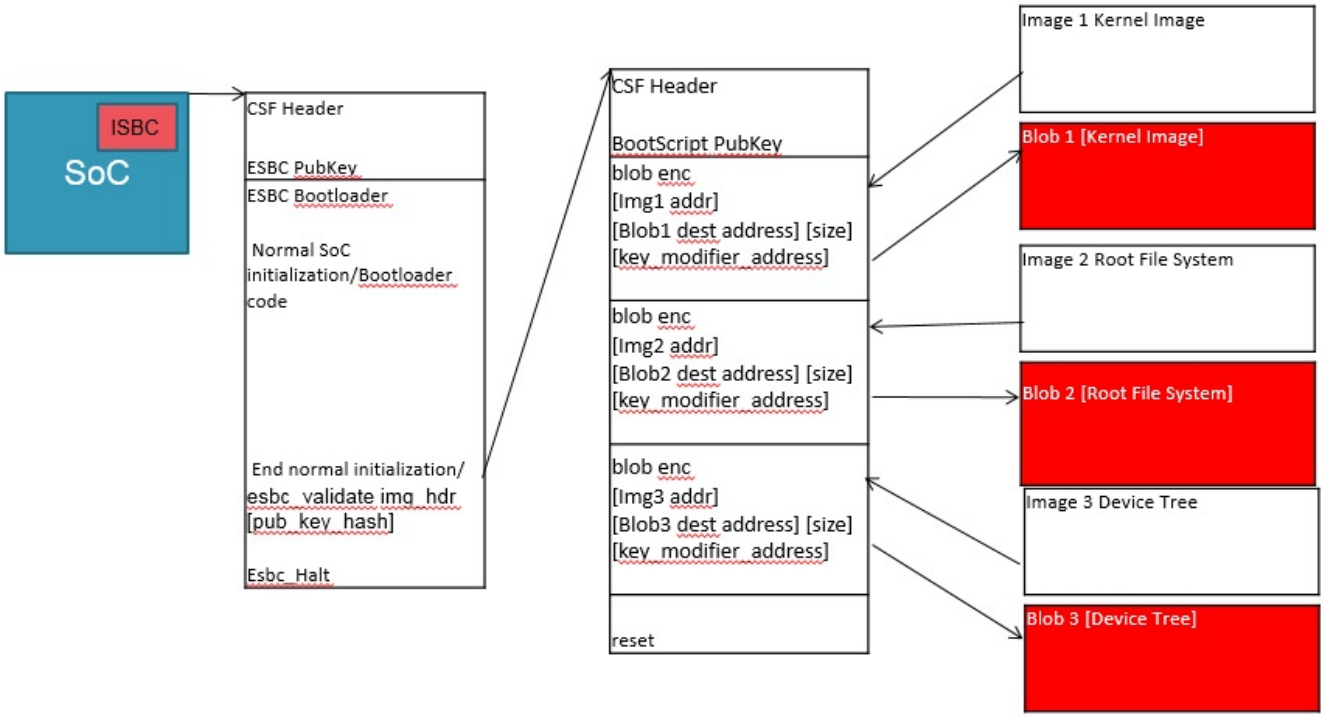


Figure 316. Chain of Trust with Confidentiality (Encapsulation)

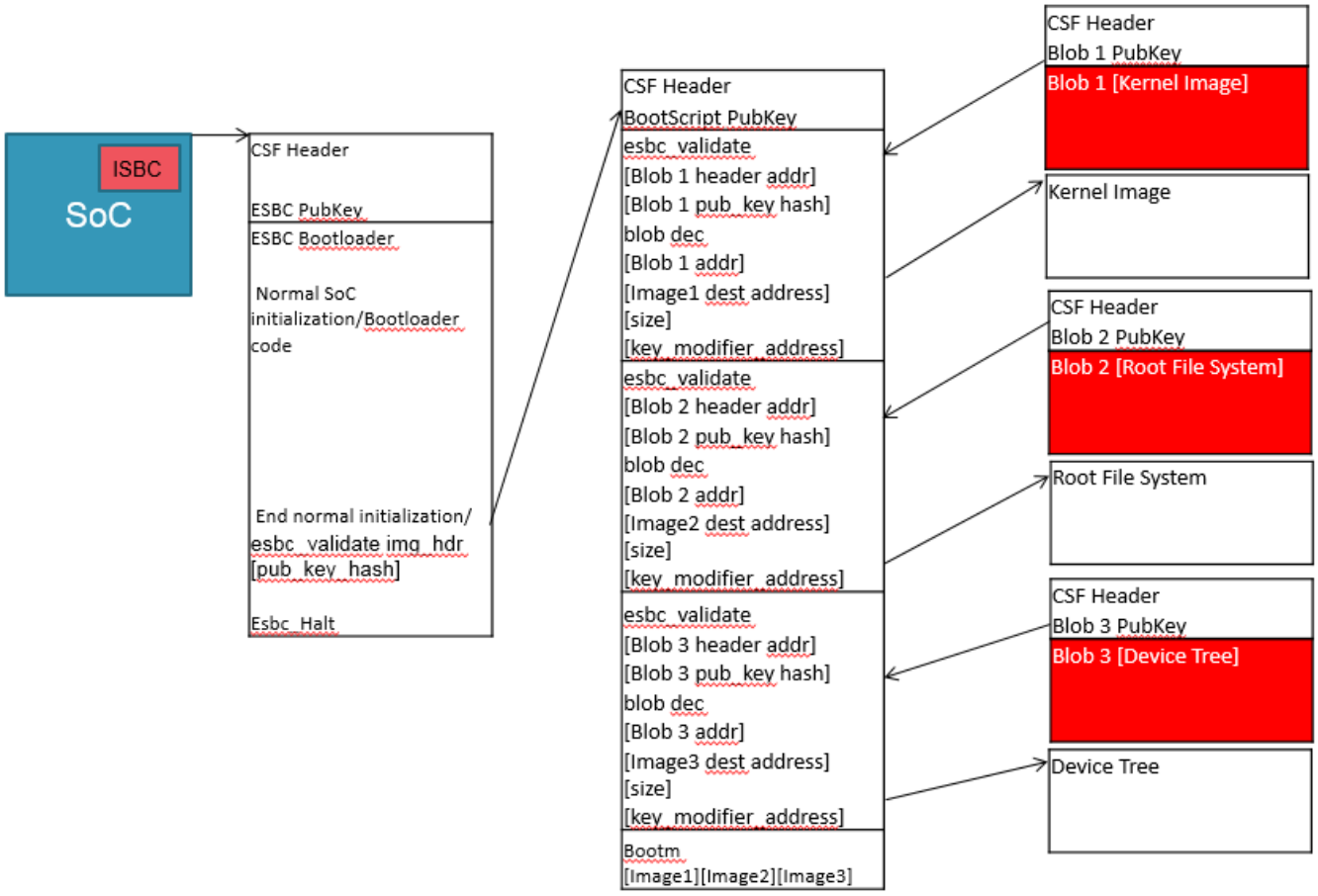


Figure 317. Chain of Trust with Confidentiality (Decapsulation)

10.3.1.5.1.3.1 Sample Encap Boot Script

A sample encap boot script would look like:

```
...
blob enc <Img1 addr> <Img1 dest addr> <Img1 size> <key_modifier address>
erase <encap Img1 addr> +<encap Imag1 size>
cp.b <Img1 dest addr> <encap Img1 addr> <encap Imag1 size>

blob enc <Img2 addr> <Img2 dest addr> <Img2 size> <key_modifier address>
erase <encap Img2 addr> +<encap Imag2 size>
cp.b <Img2 dest addr> <encap Img2 addr> <encap Imag2 size>

blob enc <Img3 addr> <Img3 dest addr> <Img3 size> <key_modifier address>
erase <encap Img3 addr> +<encap Imag3 size>
cp.b <Img3 dest addr> <encap Img3 addr> <encap Img3 size>

...
```

10.3.1.5.1.3.1.1 blob enc command

blob enc <src location> <dst location> <length> <key_modifier address>

Input arguments:

`src location` - Address of the image to be encapsulated

`dst location` - Address where the blob will be created

`length` - Size of the image to be encapsulated

`key_modifier address` - Address where a random number 16 bytes long(key modifier) is placed

Description:

The command would do the following:

- Create a cryptographic blob of the image placed at src location and place the blob at dst location.

10.3.1.5.1.3.2 Sample Decap Boot Script

A sample decap boot script would look like:

```
...
blob dec <Img1 blob addr> <Img1 dest addr> <expected Img1 size> <key_modifier address>
blob dec <Img2 blob addr> <Img2 dest addr> <expected Img2 size> <key_modifier address>
blob dec <Img3 blob addr> <Img3 dest addr> <expected Img3 size> <key_modifier address>
...
bootm <Img1 dest addr> <Img2 dest addr> <Img3 dest addr>
```

10.3.1.5.1.3.2.1 blob dec command

blob dec <src location> <dst location> <length> <key_modifier address>

Input arguments:

`src location` - Address of the image blob to be decapsulated

`dst location` - Address where the decapsulated image will be placed

`length` - Expected Size of the image after decapsulation.

`key_modifier address` - Address where key modifier (Same as that used for Encapsulation) is placed

Description:

The command would do the following:

- Decapsulate the blob placed at src location and place the decapsulated data of expected size at dst location.

10.3.1.6 Next Executable (Linux Phase)

The bootloader (ESBC) finishes the platform initialization and passed control to the Linux image. The boot-chain can be further extended to be able to sign application which would be running on Linux prompt. Further RTIC can be integrated to verify memory regions using Security Engine (SEC) during run time.

10.3.1.7 Product execution

This section presents the steps needed to be followed in order to properly run the software product according to its intended use and functionalities.

10.3.1.7.1 Getting started

The example below demonstrates the secure-boot flow with all the images loaded in NOR Flash.

Steps in the demo would be:

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to validate next level images, i.e rootfs, linux uImage and device tree.

5. Once all the images are validated, bootm command in boot script would be executed which would pass control to linux.

NOTE

For PowerPC SoC's, ISBC expects the code to be validated i.e. ESBC code to be within 0 - 3.5G address map. In the demo we map the flash to address 0xc0000000 for the ISBC code to validate ESBC in NOR Flash using PBI commands. Once the control reaches the ESBC code the earlier mapping of flash is removed and flash is mapped to address 0xe0000000.

For useful commands of U-Boot and CCS, refer to commands for different platforms under the topic 'Appendix <platform> Secure Boot Demo'

10.3.1.7.1.1 Environment for Secure Boot

There are 2 ways in which secure boot can be initiated:

- Set SB_EN bit in RCW to 1.
- Programming the ITS fuse.

In a manufacturing environment, it is recommended that all fuses be programmed at once, including the ITS and OEM Section Write Protect bits. In a prototyping environment, it may be preferable to leave ITS and Write Protect unprogrammed (relying on RCW to initiate secure boot) until the developer has confidence in the secure boot process.

Two different RCW's are provided for the demo purpose:

1. The RCW which has SB_EN bit set as 0 (sben0) and can be used when ITS = 1 i.e user wants to initiate secure boot flow using fuse.
2. The RCW which has the SB_EN bit set as 1 (sben1)and can be used when user wants to initiate secure boot using RCW.

10.3.1.7.1.2 SDK/ Images required for the demo

Given below are the images required for the demo which are built with Yocto as part of the SDK:

1. RCW with PBI commands
2. ESBC (U-Boot)
3. uImage (Linux Image) *
4. rootfs Image *
5. Device tree *

Please refer to User Manual QorIQ DPAA SDK for detailed description on how to run Yocto Build. Once the build process finishes, all the binaries would be present at the following location:

build_<platform>_release/tmp/deploy/images

The images will be created with the following names:

u-boot-<platform>.bin	U-Boot binary image for Secure Boot
uImage-<platform>.bin	kernel image that can be loaded with U-Boot
fsl-image-core-<platform>.ext2.gz.u-boot	ramdisk filesystem image that can be loaded with U-Boot
uImage-<platform>.dtb	device tree binary(dtb) for kernel bootup

RCW files will be present in the path ***build_<platform>_release/tmp/deploy/images/rcw***

NOTE

* Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB)

10.3.1.7.2 Chain of Trust

This section presents the steps needed to be followed in order to execute Chain of Trust.

Steps in the demo would be:

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to validate next level images, i.e rootfs, linux uImage and device tree.
5. Once all the images are validated, bootm command in boot script would be executed which would pass control to linux.

10.3.1.7.2.1 Other images required for the demo

Apart from SDK images described above, the following images are also required:

1. CSF Header of the ESBC u-boot image
2. CSF Header of the uImage *
3. CSF Header of the rootfs image *
4. CSF Header of the device tree *
5. Boot Script
6. CSF Header of the boot script

The following section describes how to create the CSF headers and boot script.

NOTE

* Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB). For these platforms a single CSF Header is required.

10.3.1.7.2.2 Boot Script and Signing the images

User can sign all the images with same public/private key pair or can use different key pairs to sign the images. Section below describes both the processes.

CST tool used for signing the images is provided as a package with yocto and is built for host. It can be run from your host machine.

Install path for CST binaries in yocto:

```
tmp/sysroots/x86_64-linux/usr/bin/cst/
```

CST uses openssl libraries, version 0.9.8.

In the Yocto environment, the user needs to use below commands to rebuild cst:

1. bitbake cst-native –c cleanall
2. bitbake cst-native
3. Modify the CST source code if needed.
4. bitbake cst-native –c cleanall
5. bitbake cst-native –c patch

Note: after step5, CST binary will be put to build_<platform_release/tmp/sysroots/x86_64-linux/usr/bin/cst/ directory.

Table 304. Platforms supported in SDK for Secure Boot

Soc	<platform> supported in SDK	<platform> supported in CST
B4860	b4860qds	b4860 [nor]
P2041	p2041rdb	p3_p4_p5 [nor]
P3041	p3041ds	p3_p4_p5 [nor/nand]
P4080	p4080ds	p3_p4_p5 [nor]
P5020	p5020ds	p3_p4_p5 [nor/nand]
P5040	p5040ds	p3_p4_p5 [nor/nand]
T1024	t1024rdb	t1_t2_t4 [nor]
T104x	t1040rdb, t1042rdb	t1_t2_t4 [nor/nand]
T2080	t2080qds, t2080rdb	t1_t2_t4 [nor]
T4240	t4240qds	t1_t2_t4 [nor]
LS1021	ls1021aqds	ls1 [nor]
LS1021	ls1021atwr	ls1 [nor/sd]
LS1043	ls1043ardb	ls1043 [nor/nand_sd]
LS1043	ls1043aqds	ls1043 [nor]
LS1046	ls1046aqds	ls1046 [nor]
LS1046	ls0146ardb	ls0146 [qspi/nand/sd]
LS1012	ls1012ardb	ls1012 [qspi]

NOTE

Some platforms with ARMv8 core have support for LINUX boot using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB).

For such platforms, in CST as well instead of 3 different input files for signing uImage, rootfs and dtb, there is a single input file named **input_kernel_secure** which can be used to sign the single kernel FIT image.

This applies to all subsequent sections below.

10.3.1.7.2.2.1 Signing the images using same key pair

CSF header needs to be generated for all the images. More details on the commands provided by CST can be found in Section .

1. Generate the key pair to be used for signing the image

   ```
   ./gen_keys 1024
   ```

   Key pair - public key file - srk.pub and private key in srk.priv would be generated.

2. Obtain hash string of the key pair generated to be programmed in SFP

   ```
   ./uni_sign --hash input_files/uni_sign/<platform>/input_uboot_secure
   ```

   This would provide you the 256 bit hash in form of string of the key pair generated in the previous step. The hash has to be programmed in the SRK hash Fuse.

3. Create CSF header for u-boot Image.

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_uboot_secure
   ```

   The input fields are specified in input_uboot_secure file. Please ensure that the filename mentioned in the input_uboot_secure is same as copied in the cst directory.

4. Create CSF header for Linux uImage

   ./uni_sign input_files/uni_sign/<platform>/input_uimage_secure

   uImage.bin would be validated form u-boot. The flash address used here is according to the address map of u-boot. Please ensure that filename mentioned in the input_uimage_secure is same as copied in the cst directory.

5. Create CSF header for rootfs

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_rootfs_secure
   ```

   ```
   Please make sure that filename mentioned in the input_rootfs_secure is same as copied in the cst directory
   ```

6. Create CSF Header for hardware device tree

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_dtb_secure
   ```

   ```
   Please make sure that filename mentioned in the input_dtb_secure is same as copied in the cst directory
   ```

7. Create Boot script

   Bootscript is a U-Boot script image. Steps to create bootscript are given below :

   a. Create a text file bootscript.txt with following commands.

   ```
   esbc_validate <uImage CSF Header address>

   esbc_validate <dtb CSF Header address>

   esbc_validate <rootfs CSF Header address>

   bootm <uImage Address> <rootfs address> <dtb address>
   ```

   b. Then you will have to use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

   powerpc arch:: `/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript.txt bootscript`

   arm arch:: `/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript.txt bootscript`

8. Generate CSF hdr for the boot script

   ./uni_sign input_files/uni_sign/<platform>/input_bootscript_secure

   The fields can be changed in the input files for the images based on the requirement.

10.3.1.7.2.2.2 Signing the images using different key pair

If boot script is also signed with a different key, remember to define the macro "**CONFIG_BOOTSCRIPT_KEY_HASH**" with the hash of the key used to sign the boot script in file *arch/powerpc/asm/include/fsl_secure_boot.h*. *ESBC u-boot would have to be recompiled if any change in this file is made*.

1. Generate the key pair to be used for signing the image

   ```
   ./gen_keys 1024 -p u-boot.priv -k u-boot.pub
   ```

   Key pair - public key file - u-boot.pub and private key in u-boot.priv would be generated.

2. Obtain hash string of the key pair generated to be programmed in SFP

   ```
   ./uni_sign --hash input_files/uni_sign/<platform>/input_uboot_secure
   ```

   This would provide you the 256 bit hash in form of string of the key pair generated in the previous step. The hash has to be programmed in the SRK hash Fuse.

3. Create CSF header for u-boot Image.

   Open `input_files/uni_sign/<platform>/input_uboot_secure` and change PRI_KEY and PUB_KEY to u-noot.priv and u-boot.pub respectively and run the following command.

   ./uni_sign input_files/uni_sign/<platform>/input_uboot_secure

4. Create CSF header for Linux uImage using different key pair.

   Repeat step 1 to generate another key pair.

   ```
   ./gen_keys 1024 -p lnx.priv -k lnx.pub
   ```

   Open `input_files/uni_sign/<platform>/input_uimage_secure` and change PRI_KEY and PUB_KEY to lnx.priv and lnx.pub respectively and run the following command. **./uni_sign input_files/uni_sign/<platform>/input_uimage_secure**

   Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <lnx_key_hash>

5. Create CSF header for rootfs

   ```
   ./gen_keys 1024 -p rootfs.priv -k rootfs.pub
   ```

   Open `input_files/uni_sign/<platform>/input_rootfs_secure` and change PRI_KEY and PUB_KEY to rootfs.priv and rootfs.pub respectively and run the following command.

   ./uni_sign input_files/uni_sign/<platform>/input_rootfs_secure

   Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <rootfs_key_hash>

6. Create CSF Header for hardware device tree

   ```
   ./gen_keys 1024 -p dtb.priv -k dtb.pub
   ```

   Open `input_files/uni_sign/<platform>/input_dtb_secure` and change PRI_KEY and PUB_KEY to dtb.priv and dtb.pub respectively and run the following command. **./uni_sign input_files/uni_sign/<platform>/input_dtb_secure**

   Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <dtb_key_hash>

7. Write Boot script

   Bootscript is a U-Boot script image. Steps to create bootscript are given below :

   a. Create a text file bootscript.txt with following commands.

   ```
   esbc_validate <uImage CSF Header address>  <lnx_key_hash>
   esbc_validate <dtb CSF Header address>  <dtb_key_hash>
   esbc_validate <rootfs CSF Header address>  <rootfs_key_hahs>
   bootm <uImage Address> <rootfs address> <dtb address>
   ```

   NOTE

   Hashes would be the 256 bit string hash. These are the hashes of the key used to sign the respective images.

   b. Generate header over bootscript.txt which will be consumed by uboot command source

powerpc arch :: tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript.txt bootscript

arm arch :: tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript.txt bootscript

8. Generate CSF hdr for the boot script

```
./gen_keys 1024 -p bs.priv -k bs.pub
```

Open `input_files/uni_sign/<platform>/input_dtb_secure` and change `PRI_KEY` and `PUB_KEY` to `bs.priv` and `bs.pub` respectively and run the following command.

```
./uni_sign input_files/uni_sign/<platform>/input_dtb_secure
```

10.3.1.7.2.3 Running secure boot (Chain of Trust)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
   a. **Flow A**

      Program the ITS fuse. Use RCW with SB_EN=0

      Or

   b. **Flow B**

      For protyping phase, don't blow the ITS fuse, but use rcw with SB_EN = 1.

      Note: For P3/P4/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)

2. Blow other required fuses on the board. (OTPMK and SRK hash[17]) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

> **NOTE**
>
> SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of Signing the images using same key pair on page 1954
>
> For testing purpose, the SRK Hash can be written in the mirror registers.
>
> gen_otpmk_drbg utility in cst can be used to generate otpmk key.

3. Flash all the generated images at locations as described in the address map (specified for different platforms under the topic 'Appendix <platform> Secure Boot Demo').
   a. **Flow A** - All the images would have to be flashed at the current bank addresses. Once ITS fuse is blown, the control would automatically shift to ISBC on power on.
   b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.
4. Give a power on cycle to the board.
   a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the signed linux, rootfs and dtb images
      - Linux would come up
   b. **Flow B** (*Secure boot Images flashed on alternate Bank*)

[17] Blowing of **OTPMK**is essential to run secure boot for both Production (Flow A) and Prototyping/Development (Flow B).

For **SRK Hash**,in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

- On power on cycle, u-boot prompt on bank 0 would come up.
- On switching to alternate bank, the secure boot flow as mentioned above would execute.

10.3.1.7.3 Chain of Trust with Confidentiality

This section presents the steps needed to be followed in order to execute Chain of Trust with confidentiality.

The demo would be divided into two parts:

1. Creating /encrypting images in form of blobs.
2. Decrypting the images, and booting from decrypted images.

Steps in the demo would be:

Step 1: Creating blobs

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to encapsullate next level images, i.e rootfs, linux uImage and device tree.

blob encapsulation command::

blob enc src dst len km - Encapsulate and create blob of data

$len - Number of bytes to be encapsulated.

$src - The address where image to be encapsulated is present.

$dst - The address where encapsulated image will be stored.

$km - It is the address where the key modifier is stored. The modifier is required and used as key for cryptographic operation. Key modifier should be 16 bytes long.

Step 2: Decrypting blob and booting

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to decapsulate/decrypt next level images, i.e rootfs, linux uImage and device tree.
5. After decryption, bootm command would be executed in boot script to pass control to Linux.

blob decapsulation command::

blob dec src dst len km - Decapsulate the image and recover the data

$len - Number of bytes to be decapsulated.

$src - The address where encapsulated image is present.

$dst - The address where decapsulated image will be stored.

$km - It is the address where the key modifier is stored. The modifier is required and used as key for cryptographic operation. Key modifier should be 16 bytes long. It should be same as passed while encapsulating the image.

10.3.1.7.3.1 Other images required for the demo

Apart from SDK images described above, the following images are also required:

1. Encap Boot script
2. Decap Boot script
3. CSF header for ESBC u-boot Image

4. CSF Header of the encap boot script
5. CSF Header of the decap boot script

The following section describes how to create the CSF headers and boot script.

10.3.1.7.3.2 Encap Bootscript

1. Create a bootscript_en.txt file with following commands:

```
blob enc <uImage address> 0x10000000 <uImage size> <key_modifier address>

erase <encapsulated uImage address> +<encapsulated uImage size>
cp.b 0x10000000 <encapsulated uImage address> <encapsulated uImage size>

blob enc <rootfs address> 0x20000000 <rootfs size> <key_modifier address>

erase <encapsulated rootfs address> +<encapsulated rootfs size>
cp.b 0x20000000 <encapsulated rootfs address>  <encapsulated rootfs size>

blob enc <dtb address> 0x1000000 <dtb size> <key_modifier address>

erase <encapsulated dtb address> +<encapsulated dtb size>
cp.b 0x1000000 <encapsulated dtb address>  <encapsulated dtb size>
```

   For the addresses to load images refer Section for different platforms under the topic 'Appendix <platform> Secure Boot Demo'

2. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript_en.txt
bootscript_encap
```

10.3.1.7.3.3 Decap Bootscript

1. Create a bootscript_de.txt file with following commands:

```
blob dec <encapsulated uImage address> 0x10000000 <uImage size + 0x30> <key_modifier address>

blob dec <encapsulated rootfs address> 0x20000000 <rootfs size + 0x30> <key_modifier address>

blob dec <encapsulated dtb address> 0x1000000 <dtb size + 0x30> <key_modifier address>

bootm 0x10000000 0x20000000 0x1000000
```

   For the addresses to load images refer section for different platforms under the topic 'Appendix <platform> Secure Boot Demo'

   The script decapsulates/decrypts the blob created by earlier boot script and boots using them.

> **NOTE**
>
> 0x30(48 bytes) should be added in the size of encapsulated images while decapsulating them. Always 48B are added at the end of the encapsulated image which needs to be added while providing the size of image to be decapsulated in blob dec command.

2. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript_de.txt
bootscript_decap
```

10.3.1.7.3.4 Creating CSF Headers

- **CSF Header for ESBC**

  Use the command given below to generate the hdr for u-boot binary.

  ./uni_sign input_files/uni_sign/<platform>/<input file for uboot>

  Please change the binary name as per your uboot binary in “IMAGE_1”.

- **CSF Header for bootscript_encap and bootscript_decap**

  Use the command given below to generate the headers for bootscripts

  ./uni_sign input_files/uni_sign/<platform>/<input file for bootscript>

  Please change the binary name as per your bootscript in “IMAGE_1”.

10.3.1.7.3.5 Running secure boot (Chain of Trust with Confidentiality)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
   a. **Flow A**

      Program the ITS fuse. Use RCW with SB_EN=0

      Or

   b. **Flow B**

      For protyping phase, don't blow the ITS fuse, but use rcw with SB_EN = 1.

      Note: For P3/P4/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)

2. Blow other required fuses on the board. (OTPMK and SRK hash[18]) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

   NOTE

   SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot.

   For testing purpose, the SRK Hash can be written in the mirror registers.

   gen_otpmk_drbg utility in cst can be used to generate otpmk key.

3. Flash all the generated images at locations as described in the address map specified for different platforms under the topic 'Appendix <platform> Secure Boot Demo'
   - Uboot binary
   - CSF Header of uboot
   - Linux uImage
   - Rootfs
   - Device tree
   - bootscript_encap

[18] Blowing of **OTPMK**is essential to run secure boot for both Production (Flow A) and Prototyping/Development (Flow B).

For **SRK Hash**,in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

- CSF Header for bootscript_encap

a. **Flow A** - All the images would have to be flashed at the current bank addresses. Once ITS fuse is blown, the control would automatically shift to ISBC on power on.

b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.

4. Give a power on cycle to the board.
   a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the bootscript.
      - Bootscript would encapsulate the Linux, rootfs and device tree and store the blobs at the desired locations.
   b. **Flow B** (*Secure boot Images flashed on alternate Bank*)
      - On power on cycle, u-boot prompt on bank 0 would come up.
      - On switching to alternate bank, the secure boot flow as mentioned above would execute.
5. Give a power on cycle to the board.
6. Replace the encap bootscript and its CSF header with decap bootscript and the CSF header of the decap bootscript respectively.
7. Give a power on cycle to the board.
   a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the bootscript.
      - Bootscript would decapsulate the Linux, rootfs and device tree bloband store them on DDR
      - Bootm commnd in bootscript would execute on successful decapsulation
      - Linux prompt would come up. .
   b. **Flow B** (*Secure boot Images flashed on alternate Bank*)
      - On power on cycle, u-boot prompt on bank 0 would come up.
      - On switching to alternate bank, the secure boot flow as mentioned above would execute.

10.3.1.7.4 NAND Secure Boot (Chain of Trust)

This section presents the steps and images needed for running Secure Boot Chain of Trust from NAND on **P3/P5 and T1**.

The procedure for running Secure boot from NAND is same as Secure Boot from NOR. The only difference is that in case of NOR, image is not required to be copied from NOR while in case of NAND, images have to be copied from NAND to SRAM/ DDR before validation.

Images Required for Demo

- **For P3/P5 platform**
  1. PBL.bin : For P3/P5 platforms, the PBL.bin is generated using QCVS Tool. It creates the RCW along with PBI commands. ESBC (U-boot) and CSF Header for U-Boot are added using ACS_WRITE PBI commands. (For details/ screenshots refer Using QCVS Tool (Secure Boot From NAND) on page 1997)
  2. u-boot.bin
  3. uImage (Linux Image)
  4. rootfs
  5. dtb (Device Tree)

6. CSF Header of the uImage
7. CSF Header of the u-boot.bin
8. CSF Header of the rootfs image
9. CSF Header of the device tree
10. Boot Script
11. CSF Header of the Boot Script

- **For T1 and TA2.x based platforms**

1. u-boot image : The SRAM is of 256KB so, SPL image is loaded there and the main u-boot is loaded by the SPL to DDR. To add ACS commands to rcw file in this case, CST tool can be used.
2. u-boot-spl.bin
3. u-boot.bin
4. uImage (Linux Image) *
5. rootfs *
6. dtb (Device Tree) *
7. CSF Header of the u-boot-spl.bin
8. CSF Header of the u-boot.bin
9. CSF Header of the uImage *
10. CSF Header of the rootfs image *
11. CSF Header of the device tree *
12. Boot Script
13. CSF Header of the Boot Script

NOTE

Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB). For these platforms a single CSF Header is required.

Boot Script

The sample bootscript.txt would have the following commands:

```
# Read uImage & Header
nand read <uImage DDR> <uImage NAND> <uImage size>
nand read <uImage Header DDR> <uImage Header NAND> <uImage Header size>

# Read rootfs & Header
nand read <rootfs DDR> <rootfs NAND> <rootfs size>
nand read <rootfs Header DDR> <rootfs Header NAND> <rootfs Header size>

# Read dtb & Header
nand read <dtb DDR> <dtb NAND> <dtb size>
nand read <dtb Header DDR> <dtb Header NAND> <dtb Header size>

# Validate and Boot
esbc_validate <uImage Header DDR>
esbc_validate <DTB Header DDR>
esbc_validate <rootfs Header DDR>
bootm <uImage DDR> <rootfs DDR> <dtb DDR>
```

Image Signing

- The image signing process for P3/P5 will remain same as in case of NOR Boot Script and Signing the images on page 1953

<platform> will be p3_p4_p5/nand for P3/P5

- The image signing process for signing for T1 and TA2.x platforms in case of nand boot is as below

  The CSF headers of images are to be generated as follows

  1. Generate the key pair to be used for signing the image

     ```
     ./gen_keys 1024
     ```

     Key pair - public key file - srk.pub and private key in srk.priv would be generated

  2. Create CSF header for above mentioned images

     ```
     ./uni_sign input_files/uni_sign/<platform>/nand/input_uboot_secure
     ```

     The input fields are specified in input_uboot_secure file. Please ensure that the filename mentioned in the input_uboot_secure is same as copied in the cst directory

     Similarly for other images

     ```
     ./uni_sign input_files/uni_sign/<platform>/nand/input_uimage_secure
     ./uni_sign input_files/uni_sign/<platform>/nand/input_bootscript_secure
     ./uni_sign input_files/uni_sign/<platform>/nand/input_rootfs_secure
     ./uni_sign input_files/uni_sign/<platform>/nand/input_dtb_secure
     ./uni_sign input_files/uni_sign/<platform>/nand/input_spl_uboot_secure
     ```

     For platforms using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB), only one image used to be signed instead of three.

     ```
     ./uni_sign input_files/uni_sign/<platform>/nand/input_kernel_secure
     ```

  3. Generate new RCW (which will contain ACS command to copy u-boot-spl ad its header to OCRAM.

     Usage :: $**./uni_pbi [options] <input_file>**

     ```
     ./uni_pbi input_files/create_pbi/<platform>/input_pbi_nand_secure
     ```

NOTE

ISBC Key Extension Feature is not applicable for Secure Boot from NAND.

10.3.1.7.4.1 Running Secure Boot Chain of Trust (from NAND)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.

   a. **Flow A**

      Program the ITS fuse. Use RCW with SB_EN=0

      Or

   b. **Flow B**

      For protyping phase, don't blow the ITS fuse, but use rcw with SB_EN = 1.

      Note: For P3/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)

2. Blow other required fuses on the board. (OTPMK and SRK hash[19]) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

   NOTE

   SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of Signing the images using same key pair on page 1954

   For testing purpose, the SRK Hash can be written in the mirror registers.

   gen_otpmk_drbg utility in cst can be used to generate otpmk key.

3. Flash all the generated images on NAND Flash at locations as described in the address map (specified for different platforms under the topic 'Appendix <platform> Secure Boot Demo').
4. Switch to NAND Boot.
   a. **FLOW A (for P3/P5 platforms)**

      Change the Switch Settings to change the RCW_SRC to NAND and power on the board.

      OR

      Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot promt issue the following command in case of P3/P5 platforms

      ```
      mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d 0xf0;mw.b
      0xffdf0010 0; mw.b 0xffdf0010 1
      ```

      - The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
      - ISBC code would get control, validate the ESBC image
      - ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
      - The Boot Script has commands to copy the linux images and their respective headers from NAND to DDR, validate the signed linux, rootfs and dtb images.
      - Linux would be booted.

   b. **FLOW B (for T1 and TA2.x platforms)**

   - PBL will load u-boot-spl and its header to OCRAM, u-boot and its header will be appended to rcw file that wil be copied to nand flash along with rcw.
   - ISBC will validate u-boot spl and then u-boot-spl will load the main u-boot to DDR and will validate main u-boot (ESBC).
   - ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
   - The Boot Script has commands to copy the linux images and their respective headers from NAND to DDR, validate the signed linux, rootfs and dtb images.
   - Linux would be booted.

[19] Blowing of **OTPMK**is essential to run secure boot for both Production (Flow A) and Prototyping/Development (Flow B).

For **SRK Hash**,in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

10.3.1.7.5 SD Secure Boot (Chain of Trust)

This section presents the steps and images needed for running Secure Boot Chain of Trust from SD

The procedure for booting from SD is similar to NOR, the only difference being, SD is a non XIP memory so the images are to be copied from SD to DDR before validation

The images needed to run SD secure boot

1. RCW with PBI commands (rcw.bin)
2. U-Boot ESBC images
   a. u-boot-spl.bin
   b. u-boot-dtb.bin
3. uImage (uImage.bin)*
4. rootfs Image (rootfs)*
5. Device tree (uImage.dtb)*

NOTE

Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB). For these platforms a single CSF Header is required.

Sample BootScript

```
mmc rescan
setenv bootfile uImage.bin
setenv fdtfile uImage.dtb
setenv consoledev ttyS0
setenv loadaddr 0x83000000
setenv fdtaddr  0x83800000
setenv bootargs root=/dev/mmcblk0p2 rw rootdelay=5 console=$consoledev,$baudrate;
mmcinfo;
ext2load mmc 0:2 82020000 /boot/hdr_dtb.out
ext2load mmc 0:2 82000000 /boot/hdr_linux.out
ext2load mmc 0:2 0x83000000 /boot/$bootfile;
ext2load mmc 0:2 0x83800000 /boot/$fdtfile;
esbc_validate 0x82000000
esbc_validate 0x82020000
bootm $loadaddr - $fdtaddr
```

Signing the images

The CSF headers of images are to be generated as follows

1. Generate the key pair to be used for signing the image

   ```
   ./gen_keys 1024
   ```

   Key pair - public key file - srk.pub and private key in srk.priv would be generated

2. Create CSF header for above mentioned images

   ```
   ./uni_sign input_files/uni_sign/<platform>/sdboot/input_uboot_secure
   ```

   The input fields are specified in input_uboot_secure file. Please ensure that the filename mentioned in the input_uboot_secure is same as copied in the cst directory

Similarly for other images

```
./uni_sign input_files/uni_sign/<platform>/sdboot/input_uimage_secure
./uni_sign input_files/uni_sign/<platform>/sdboot/input_bootscript_secure
./uni_sign input_files/uni_sign/<platform>/sdboot/input_rootfs_secure
./uni_sign input_files/uni_sign/<platform>/sdboot/input_dtb_secure
./uni_sign input_files/uni_sign/<platform>/sdboot/input_spl_uboot_secure
```

3. Generate new Rcw

   Usage :: $**./uni_pbi [options] <input_file>**

```
./uni_pbi input_files/create_pbi/<platform>/input_pbi_sd_secure
```

   The above command will add ACS Write command to the **rcw** to write u-boot SPL and its header from SD card to OCRAM, and also append bootscript, uboot and their header to it

Running Secure Boot Chain of Trust(from SD)

There are two porcedures to run secure boot

- In one flow, all images are flashed on SD in raw form. The images are then copied to OCRAM/DDR and then validated.

  To view commands for this flow refer Appendix LS1020 Secure Boot demo on page 2008 , for LS1021 specific SD boot

- Alternate method to boot from SD is by mounting the root file system on SD. In that case rootfs won't be validated. The procedure for alternate SD Secure boot is mentioned below.

1. Set up the board for secure boot flow
   a. Use rcw with SB_EN =1
   b. Blow required fuses on the board (OTPMK and SRK hash). For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide

> **NOTE**
>
> Blowing of OTPMKis essential to run secure boot for both Production (Flow A) and Prototyping/ Development (Flow B).
>
> For SRK Hash,in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.
>
> Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

2. Connect the card reader with SD card to the Linux Host PC.The partition on SD card can also be created by connecting it to the board used once the linux is up using any other boot source (eg nor)

   Make two partitions in SD card, both MS DOS partition.

```
fdisk /dev/mmcblk0
```

   p – to see the partitions already created

   n – to create new partitions

```
partition no 1
first sector default
sector +1G (to indicate the size of first partition
```

   L – the type of partition

   w -to write the partition

3. After creating partition, use mkfs.ext2 to create filesystems

```
mkfs.vfat /dev/mmcblk0p1
mkfs.ext2 /dev/mmcblk0p2
```

4. Create temp directory in host PC and mount the ext2 partition to the temp

```
mkdir /mnt
mount /dev/mmcblk0p2 /mnt/
cd /mnt/
```

5. Copy the file system to harddisk by extracing the QorIQ_SDK_<Version>_<release date>_ROOTFS_Image.tar.gz. Remove the tarball after extracting rootfs

```
tar -zxvf QorIQ_SDK_<Version>_<release date>_ROOTFS_Image.tar.gz
rm QorIQ_SDK_V2.0_CORTEXA7_20160527_ROOTFS_Image.tar.gz
```

6. Make sure the kernel image, dtb file and their headers are in /mnt/boot directory, then umount the /mnt

```
scp -r $path/uImage.* ./boot
scp -r $path/hdr_linux.out ./boot
scp -r $path/hdr_dtb.out ./boot
umount /mnt
```

7. Write the new RCW file created, to 1st partition of SD at 8th sector

   The below command can be used to write to SD

   The **default** switch settings is such that the **boot source is nor**

```
setenv path <path>
tftp 81000000 $path/u-boot-with-spl-pbl-sec.bin
mmc erase 8 0x940
mmc write 0x81000000 8 0x940
```

Execution FLOW

- PBL will load u-boot-spl and its header to OCRAM, u-boot and its header will be appended to RCW file that wil be copied to SD flash along with RCW.
- ISBC will validate u-boot spl and then u-boot-spl will load the main u-boot to DDR and will validate main u-boot (ESBC).
- ESBC image would further copy the Boot Script Header and Boot Script from SD to DDR, validate the boot script and execute it.
- The Boot Script has commands to copy the linux images and their respective headers from SD to DDR, validate the signed linux, rootfs and dtb images.
- Linux would be booted.

10.3.1.7.6 NAND Secure Boot (Chain of Trust with Confidentiality)

This section presents the steps and images needed for running Secure Boot Chain of Trust with Confidentiality from NAND on **P3/P5**

The procedure for running Secure boot from NAND is same as Secure Boot from NAND. The only difference is that in case of NOR, image is not required to be copied from NOR while in case of NAND, images have to be copied from NAND to SRAM/ DDR before validation.

Images Required for Demo

1. PBL.bin

The PBL.bin is generated using QCVS Tool. It creates the RCW along with PBI commands. ESBC (U-boot) and CSF Header for U-Boot are added using ACS_WRITE PBI commands. (For details/screenshots refer Using QCVS Tool (Secure Boot From NAND) on page 1997)

2. uImage (Linux Image)
3. rootfs
4. dtb (Device Tree)
5. Encap Boot Script
6. CSF Header of the Encap Boot Script
7. Decap Boot Script
8. CSH Header for Decap Boot Script

Encap Boot Script

```
# uImage
nand read <uImage DDR> <uImage NAND> <uImage size>
esbc_blob_encap <uImage DDR> 0x10000000 <uImage size> 0x11223344556677889900aabbccddeeff
nand erase <encapsulated uImage NAND> <encapsulated uImage size>
nand write 0x10000000 <encapsulated uImage NAND> <encapsulated uImage size>

# rootfs
nand read <rootfs DDR> <rootfs NAND> <rootfs size>
esbc_blob_encap <rootfs DDR> 0x10000000 <rootfs size> 0x11223344556677889900aabbccddeeff
nand erase <encapsulated rootfs NAND> <encapsulated rootfs size>
nand write 0x10000000 <encapsulated rootfs NAND> <encapsulated rootfs size>

# dtb
nand read <dtb DDR> <dtb NAND> <dtb size>
esbc_blob_encap <dtb DDR> 0x10000000 <dtb size> 0x11223344556677889900aabbccddeeff
nand erase <encapsulated dtb NAND> <encapsulated dtb size>
nand write 0x10000000 <encapsulated dtb NAND> <encapsulated dtb size>
```

Decap Boot Script

```
nand read <encapsulated uImage DDR> <encapsulated uImage NAND> <encapsulated uImage size>
esbc_blob_decap <encapsulated uImage DDR> 0x10000000 <uImage size>
0x11223344556677889900aabbccddeeff

nand read <encapsulated rootfs DDR> <encapsulated rootfs NAND> <encapsulated rootfs size>
esbc_blob_decap <encapsulated rootfs DDR> 0x20000000 <rootfs size>
0x11223344556677889900aabbccddeeff

nand read <encapsulated dtb DDR> <encapsulated dtb NAND> <encapsulated dtb size>
esbc_blob_decap <encapsulated dtb DDR> 0x1000000 <dtb size> 0x11223344556677889900aabbccddeeff

bootm 0x10000000 0x20000000 0x1000000
```

Image Signing

The image signing process will remain same as in case of NOR NAND Secure Boot (Chain of Trust with Confidentiality) on page 1967

<platform> will be p3_p4_p5/nand

10.3.1.7.6.1 Running Secure Boot Chain of Trust with Confidentiality (from NAND)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
   a. **Flow A**

      Program the ITS fuse. Use RCW with SB_EN=0

      Or

   b. **Flow B**

      For protyping phase, don't blow the ITS fuse, but use rcw with SB_EN = 1.

      Note: For P3/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)
2. Blow other required fuses on the board. (OTPMK and SRK hash[20]) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

   NOTE

   SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of Signing the images using same key pair on page 1954

   For testing purpose, the SRK Hash can be written in the mirror registers.

   gen_otpmk_drbg utility in cst can be used to generate otpmk key.
3. Flash all the generated images on NAND Flash at locations as described in the address map (specified for different platforms under the topic 'Appendix <platform> Secure Boot Demo')
   a. PBL.bin
   b. LINUX Images (uImage, dtb, rootfs)
   c. CSF Header for bootscript_encap)
   d. bootscript_encap
4. Switch to NAND Boot.
   a. **FLOW A**

      Change the Switch Settings to change the RCW_SRC to NAND and power on the board.
   b. **FLOW B**

      Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot promt issue the following command.

```
mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d 0xf0;mw.b
0xffdf0010 0; mw.b 0xffdf0010 1
```

   - The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
   - ISBC code would get control, validate the ESBC image.
   - ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.

[20] Blowing of **OTPMK**is essential to run secure boot for both Production (Flow A) and Prototyping/Development (Flow B).

For **SRK Hash**,in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

- Bootscript would encapsulate the Linux, rootfs and device tree and store the blobs at the desired locations.

5. Revert the switch settings to earlier RCW_SRC and power on the board. Replace the encap bootscript and its CSF header with decap bootscript and the CSF header of the decap bootscript respectively.
6. Switch to NAND Boot.
   a. **FLOW A**

      Change the Switch Settings to change the RCW_SRC to NAND and power on the board.

   b. **FLOW B**

      Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot promt issue the following command.

```
mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d 0xf0;mw.b
0xffdf0010 0; mw.b 0xffdf0010 1
```

- The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
- ISBC code would get control, validate the ESBC image.
- ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
- Bootscript would copy the Linux, rootfs and device tree blobs on DDR and then decapsulate them on DDR.
- Bootm commnd in bootscript would execute on successful decapsulation.
- Linux prompt would come up.

10.3.1.8 Troubleshooting

Table 305. Troubleshooting

	Symptoms	Reasons and/or Recommended actions
1.	No print on UART console.	• Check the status register of sec mon block (location 0xfe314014). Refer to the details of the register from the Reference Manual. Bits OTPMK_ZERO, OTMPK_SYNDROME and PE should be 0 otherwise there is some error in the OTPMK fuse blown by you. • If OTMPK fuse is correct (see Step 1), check the SCRATCHRW2 register for errors. Refer to Section for error codes. • If **Error code = 0** then check the Security Monitor state in HPSR register of Sec Mon. **Sec Mon in Check State (0x9)** If ITS fuse = 1, then it means ISBC code has reset the board. This may be due to the following reasons: Hash of the public key used to sign the ESBC u-boot doesn't match with the value in SRK Hash Fuse Or Signature verification of the image failed **Sec Mon in Trusted State (0xd) or Non Secure State (0xb)** Check the entry point field in the ESBC header. It should be 0xcffffffc for the demo described in Section 4. If entry point is correct, ensure that u-boot image has been compiled with the required secure boot configuration.
2.	Instead of linux prompt, you get a u-boot command prompt instead of linux prompt.	You have not booted in secure boot mode. You never get a u-boot prompt in secure boot flow. Check Step 1 in Running secure boot (Chain of Trust) on page 1957. You would reach this stage if ITS = 0 and you are using rcw where sben0 is present in its name.
3	u-boot hangs or board resets	Some validation failure occurred in ESBC u-boot. Error code and description would be printed on u-boot console. Refer to for more details on errors.

10.3.1.9 CSF Header Data Structure

The CSF Header provides the ISBC with most of the information needed to validate the image.

P3/P4/P5 Platforms

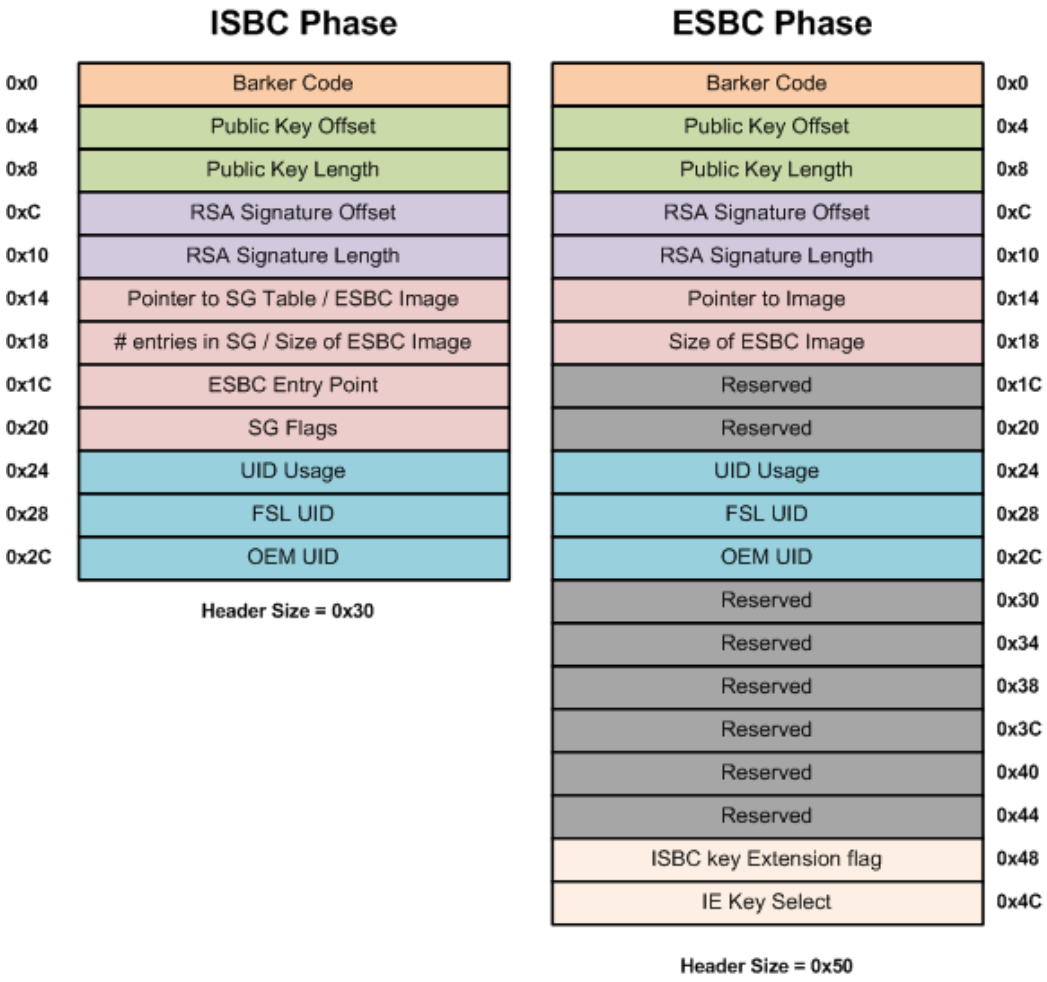


Figure 318. CSF Header for P3/P4/P5 (ISBC and ESBC Phase)

Table 306. CSF Header Format (P3/P4/P5 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	**Barker code**. This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.
0x04-0x07	**Public key offset**. This location contains an offset in bytes of the public key from the start of CSF header. Using this offset and the public key length, the public key is read.
0x08-0x0b	**Public key length** in bytes. (Value populated here should be twice of Modulus size). Supported sizes are 256, 512 or 1024 bytes (2 * 1024, 2 * 2048 , 2 * 4096 bits).

Table continues on the next page...

Table 306. CSF Header Format (P3/P4/P5 Platforms) (continued)

Offset	Data Bits [0:31]
0x0c-0x0f	**RSA Signature offset**. This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x10-0x13	**RSA Signature length** in bytes.
0x14-0x17	For ISBC Phase: Based on the Scatter Gather flag in CSF header, this location can either be treated as **Pointer to Scatter Gather table or the address of ESBC image**. For ESBC Phase: This location is treated as **address of image**(linux/bootscript/rootfs/dtb) to be validated.
0x18-0x1b	For ISBC Phase: Based on the Scatter gather flag in CSF Header, this location can either be treated as **number of entries in SG table or ESBC image size** in bytes. For ESBC Phase: Size of image to be validated
0x1c-0x1f	For ISBC Phase: **ESBC entry poin**t. ISBC transfers control to this location upon successful validation of ESBC image(s). For ESBC Phase: Reserved.
0x20-0x23	For ISBC Phase: **Scatter Gather flag**. 0x0000 - 0x14-0x17 is a pointer to the ESBC image 0x0001 - 0x14-0x17 is a pointer to a scatter/gather table. For ESBC Phase: Reserved
0x24-0x27	**Unique ID Usage**. UIDs present in the CSF Header are compared to the corresponding UIDs in the SFP, and are included in the ESBC validation. 0x0000 - No UIDs are present in CSF header 0x0001 - FSL_UID and OEM_UID are present in CSF header 0x0002 - Only OEM_UID present in CSF header 0x0004 - Only FSL_UID present in CSF header
0x28-0x2b	**NXP unique ID**. A unique 32 bit value, which is specific to NXP. This value is compared with the FSL ID in Secure Fuse Processor 's FSL-ID registers

Table continues on the next page...

Table 306. CSF Header Format (P3/P4/P5 Platforms) (continued)

Offset	Data Bits [0:31]
0x2c-0x2f	**OEM unique ID**. A unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID in Secure Fuse Processor 's OEM-ID registers
0x30-0x47	For ISBC Phase: Not Applicable For ESBC Phase: Reserved
0x48-0x4b	For ISBC Phase: Not Applicable For ESBC Phase: **ISBC key Extension flag**. If this flag is set, key to be used for validation needs to be picked up from IE Key table.
0x4c-0x4f	For ISBC Phase: Not Applicable For ESBC Phase: **IE Key Select**. Key Number to be used from the IE Key Table if IE flag is set.

Table 307. Scatter Gather Table Format (P3/P4/P5 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	Length in bytes of the first segment of the ESBC image.
0x04-0x07	Pointer to first segment of ESBC image.
0x08-0x0b	Length in bytes of the second segment of the ESBC image.
0x0c-0x0f	Pointer to second segment of ESBC image.
0xww-0xxx	Length in bytes of the nth segment of the ESBC image.
0xyy-0xzz	Pointer to nth segment of ESBC image

Table 308. Signature (P3/P4/P5 Platforms)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

Table 309. Public key (P3/P4/P5 Platforms)

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

B4/T1/T2/T4 Platforms

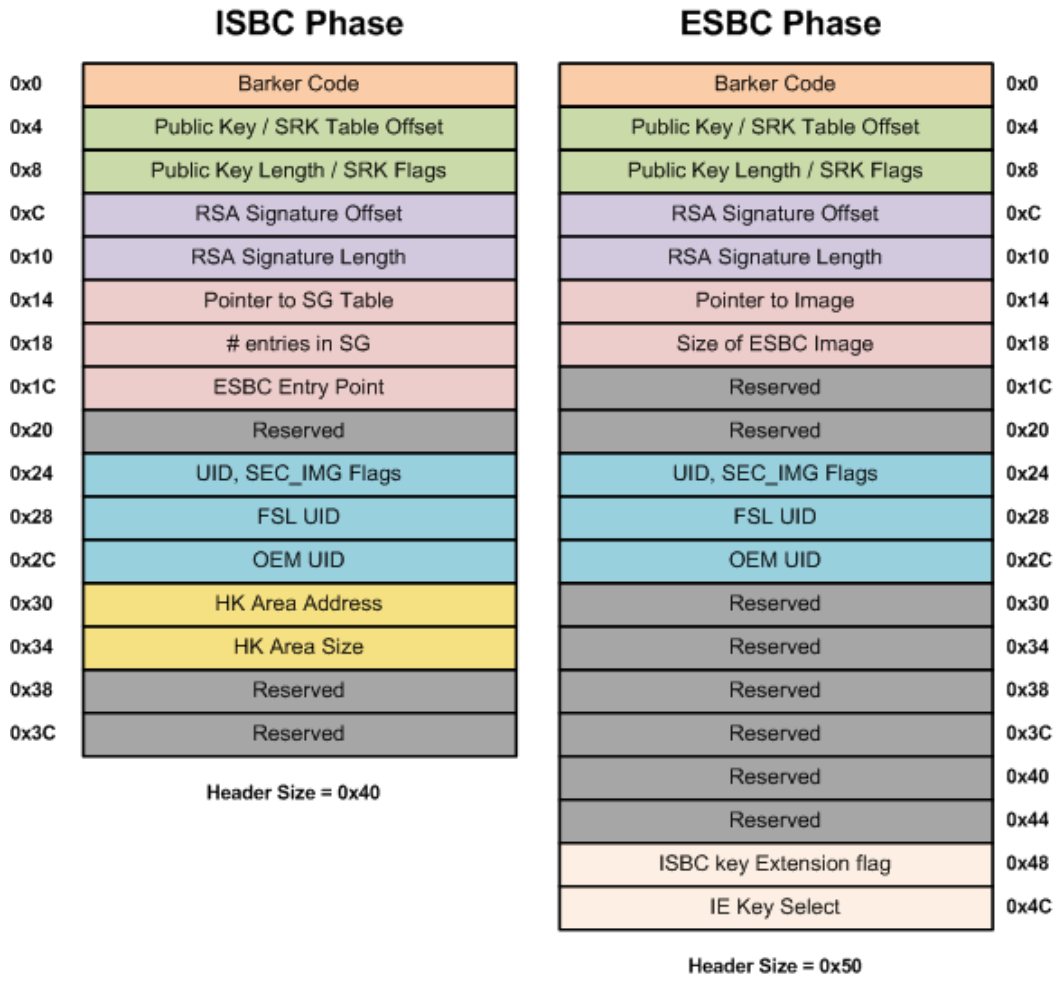


Figure 319. CSF Header for B4/T1/T2/T4 (ISBC and ESBC Phase)

Table 310. CSF Header Format (B4/T1/T2/T4 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	**Barker code**. This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.

Table continues on the next page...

Table 310. CSF Header Format (B4/T1/T2/T4 Platforms) (continued)

Offset	Data Bits [0:31]
0x04-0x07	If the srk_table_flag is not set : • **Public key offset**: This location contains an address which is the offset of the public key from the start of CSF header. Using this offset and the public key length, the public key is read. If srk_table_flag is set: • **Srk table offset**: This location contains an address which is the offset of the srk table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.
0x08	**Srk table flag**. This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.
0x09-0x0b	If the srk_table_flag is not set : • **Public key length**: This location contains the length of the public key in bytes. If srk_table_flag is set: • 0x09 – **Key Number from srk table** which is to be used for verification. • 0x0a-0x0b – **Number of entries in srk table**. Minimum number of entries in table = 1, Maximum = 4.
0x0c-0x0f	**RSA Signature offset**. This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x10-0x13	**RSA Signature length** in bytes.
0x14-0x17	For ISBC Phase: **SG Table offset** This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries is SG Table, the SG table is read. For ESBC Phase: Address of the image to be validated.
0x18-0x1b	For ISBC Phase: **Number of entries in SG Table** (Earlier ,Based on the Scatter gather flag in CSF Header, this location can either be treated as number of entries in SG table or ESBC image size in bytes.). For ESBC Phase: **Size of Image** to be validated

Table continues on the next page...

Table 310. CSF Header Format (B4/T1/T2/T4 Platforms) (continued)

Offset	Data Bits [0:31]
0x1c-0x1f	For ISBC Phase: **ESBC entry point**. ISBC transfers control to this location upon successful validation of ESBC image(s). For ESBC Phase: Reserved
0x20-0x23	**Reserved** .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.)
0x24	For ISBC Phase: Reserved For ESBC Phase: Reserved
0x25	For ISBC Phase: **Secondary Image flag** Indicates if user has a secondary image available in case of failures in validating primary iamge.Valid in case of primary Images's Header. For ESBC Phase:Reserved
0x26-0x27	**Unique ID Usage** This location contains a flag which specifies one of these possibilities • 0x00 - No UID's present • 0x01 - FSL UID and OEM UID are present • 0x02 - Only FSL UID is present • 0x04 - Only OEM UID is present
0x28-0x2b	**NXP unique ID**. A unique 32 bit value, which is specific to NXP. This value is compared with the FSL ID in Secure Fuse Processor 's FSL-ID registers
0x2c-0x2f	**OEM unique ID**. A unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID in Secure Fuse Processor 's OEM-ID registers
0x30-0x33	For ISBC Phase: **Housekeeping area address** This is address of start of a memory which can be accessed by devices on SOC bus. (DDR, L3 cache configured as SRAM). The area should have been pre-configured by user through PBL commands or configuration header For ESBC Phase: Reserved

Table continues on the next page...

Table 310. CSF Header Format (B4/T1/T2/T4 Platforms) (continued)

Offset	Data Bits [0:31]
0x34-0x37	For ISBC Phase: **Size of the housekeeping area** Size of the pre-configured memory which can be used by Boot Rom Code. For ESBC Phase: Reserved
0x38-0x3f	Reserved
0x40-0x47	For ISBC Phase: Not Applicable For ESBC Phase: Reserved
0x48-0x4b	For ISBC Phase: Not Applicable For ESBC Phase: **ISBC key Extension flag** If this flag is set, key to be used for validation needs to be picked up from IE Key table.
0x4c-0x4f	For ISBC Phase: Not Applicable For ESBC Phase: **IE Key Select** Key Number to be used from the IE Key Table if IE flag is set.

Table 311. Scatter Gather Table Format (B4/T1/T2/T4 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1.
0x04-0x07	Target where the ESBC Image 1 can be found. This field is ignored in case of PBL based SOC's.
0x08-0x0b	Source Address of ESBC Image 1
0x0c-0x0f	Destination Address of ESBC Image 1 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.
0x10-0x13	Length. This location specifies the length in bytes of the ESBC image 2.
0x14-0x17	Target where the ESBC Image 2 can be found. This field is ignored in case of PBL based SOC's.
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	Destination Address of ESBC Image 2 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.

Table 312. Signature (B4/T1/T2/T4 Platforms)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

Table 313. Public key (B4/T1/T2/T4 Platforms)

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

Table 314. SRK Table (B4/T1/T2/T4 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

LS1 Platform

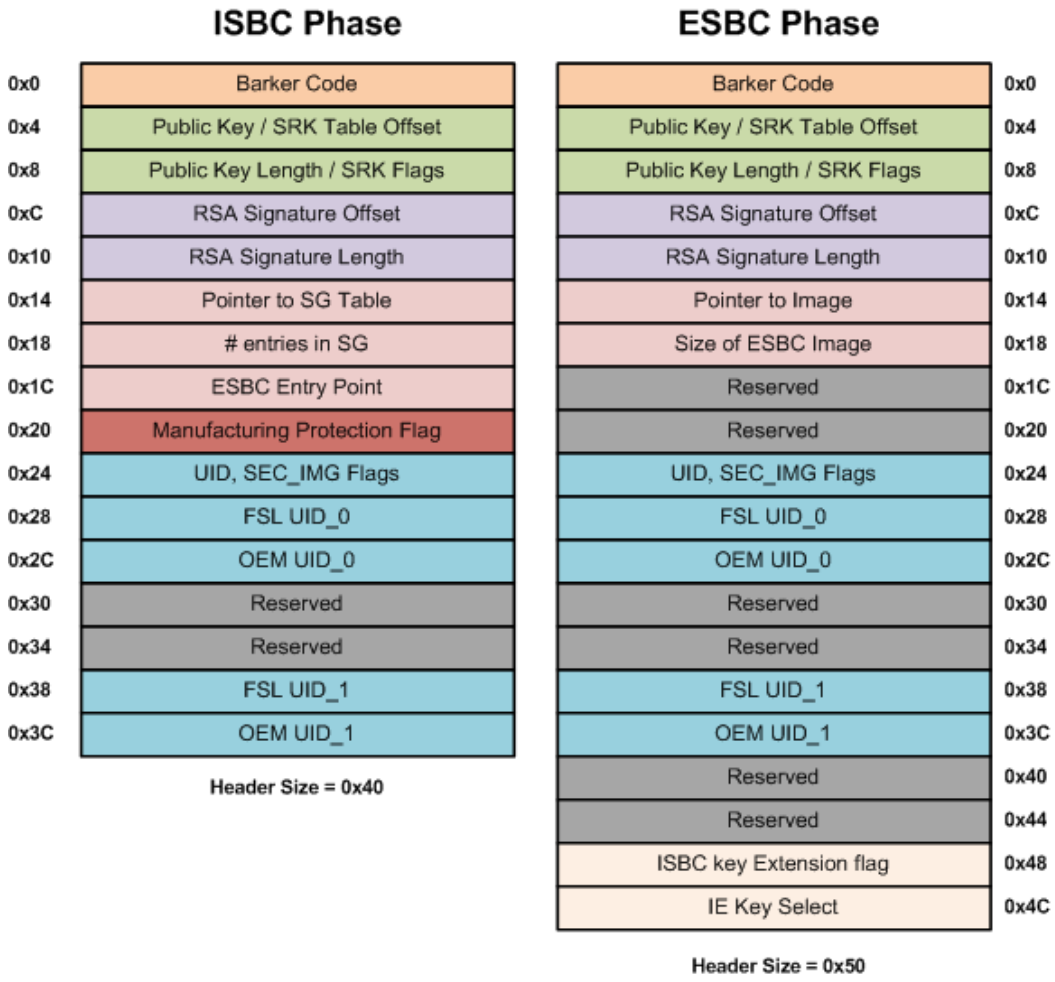


Figure 320. CSF Header for LS1 (ISBC and ESBC Phase)

Table 315. CSF Header Format (LS1 Platform)

Offset	Data Bits [0:31]
0x00-0x03	**Barker code**. This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.
0x07-0x04	If the srk_table_flag is not set : • **Public key offset**: This location contains an address which is the offset of the public key from the start of CSF header. Using this offset and the public key length, the public key is read. If srk_table_flag is set: • **Srk table offset**: This location contains an address which is the offset of the srk table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.

Table continues on the next page...

Table 315. CSF Header Format (LS1 Platform) (continued)

Offset	Data Bits [0:31]
0x08	**Srk table flag**. This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.
0x0b-0x09	If the srk_table_flag is not set : • **0x0b-0x9** -- **Public key length**: This location contains the length of the public key in bytes. If srk_table_flag is set: • **0x09** – **Key Number from srk table** which is to be used for verification. • **0x0b-0x0a** – **Number of entries in srk table**. Minimum number of entries in table = 1, Maximum = 4.
0x0f-0x0c	**RSA Signature offset**. This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x13-0x10	**RSA Signature length** in bytes.
0x17-0x14	**For ISBC Phase**: **SG Table offset** This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries is SG Table, the SG table is read. **For ESBC Phase**: Address of the image to be validated.
0x1b-0x18	**For ISBC Phase**: **Number of entries in SG Table** (Earlier ,Based on the Scatter gather flag in CSF Header, this location can either be treated as number of entries in SG table or ESBC image size in bytes.). **For ESBC Phase** **Size of image** to be validated
0x1f-0x1c	**For ISBC Phase**: **ESBC entry point**. ISBC transfers control to this location upon successful validation of ESBC image(s). **For ESBC Phase**: Reserved
0x21-0x20	**Manufacturing Protection Flag** Indicates if manufacturing protection has to be enabled or not in ISBC.
0x23-0x22	**Reserved** .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.)

Table continues on the next page...

Table 315. CSF Header Format (LS1 Platform) (continued)

Offset	Data Bits [0:31]
0x24	For ISBC Phase: Reserved For ESBC Phase: Reserved
0x25	**For ISBC Phase** **Secondary Image flag** Indicates if user has a secondary image available in case of failures in validating primary iamge.Valid in case of primary Images's Header. **For ESBC Phase**:Reserved
0x27-0x26	**Unique ID Usage** This location contains a flag which specifies one of these possibilities • 0x00 - No UID's present • 0x01 - FSL UID and OEM UID are present • 0x02 - Only FSL UID is present • 0x04 - Only OEM UID is present
0x2b-0x28	**NXP unique ID 0** Upper 32 bits of a unique 64 bit value, which is specific to NXP. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x2f-0x2c	**OEM unique ID 0** Upper 32 bits of a unique 64 bit value, which is specific to OEM. This value is compared with the OEM ID 0 in Secure Fuse Processor 's OEM-ID registers
0x37-0x30	Reserved
0x3b-0x38	**NXP unique ID 1** Lower 32 bits of a unique 64 bit value, which is specific to NXP. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x3f-0x3c	**OEM unique ID 1** Lower 32 bits of a unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID 1 in Secure Fuse Processor 's OEM-ID registers
0x40-0x47	For ISBC Phase: Not Applicable For ESBC Phase: Reserved
0x48-0x4b	**For ISBC Phase**: Not Applicable **For ESBC Phase**: **ISBC key Extension flag** If this flag is set, key to be used for validation needs to be picked up from IE Key table.

Table continues on the next page...

Table 315. CSF Header Format (LS1 Platform) (continued)

Offset	Data Bits [0:31]
0x4c-0x4f	**For ISBC Phase**: Not Applicable **For ESBC Phase**: **IE Key Select** Key Number to be used from the IE Key Table if IE flag is set.

Table 316. Scatter Gather Table Format (LS1 Platform)

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1.
0x04-0x07	Target where the ESBC Image 1 can be found. This field is ignored in case of PBL based SOC's.
0x08-0x0b	Source Address of ESBC Image 1
0x0c-0x0f	Destination Address of ESBC Image 1 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.
0x10-0x13	Length. This location specifies the length in bytes of the ESBC image 2.
0x14-0x17	Target where the ESBC Image 2 can be found. This field is ignored in case of PBL based SOC's.
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	Destination Address of ESBC Image 2 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.

Table 317. Signature (LS1 Platform)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

Table 318. Public key (LS1 Platform)

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

Table 319. SRK Table (LS1 Platform)

Offset	Data Bits [0:31]
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

LS1043/LS1046/LS1012 Platforms

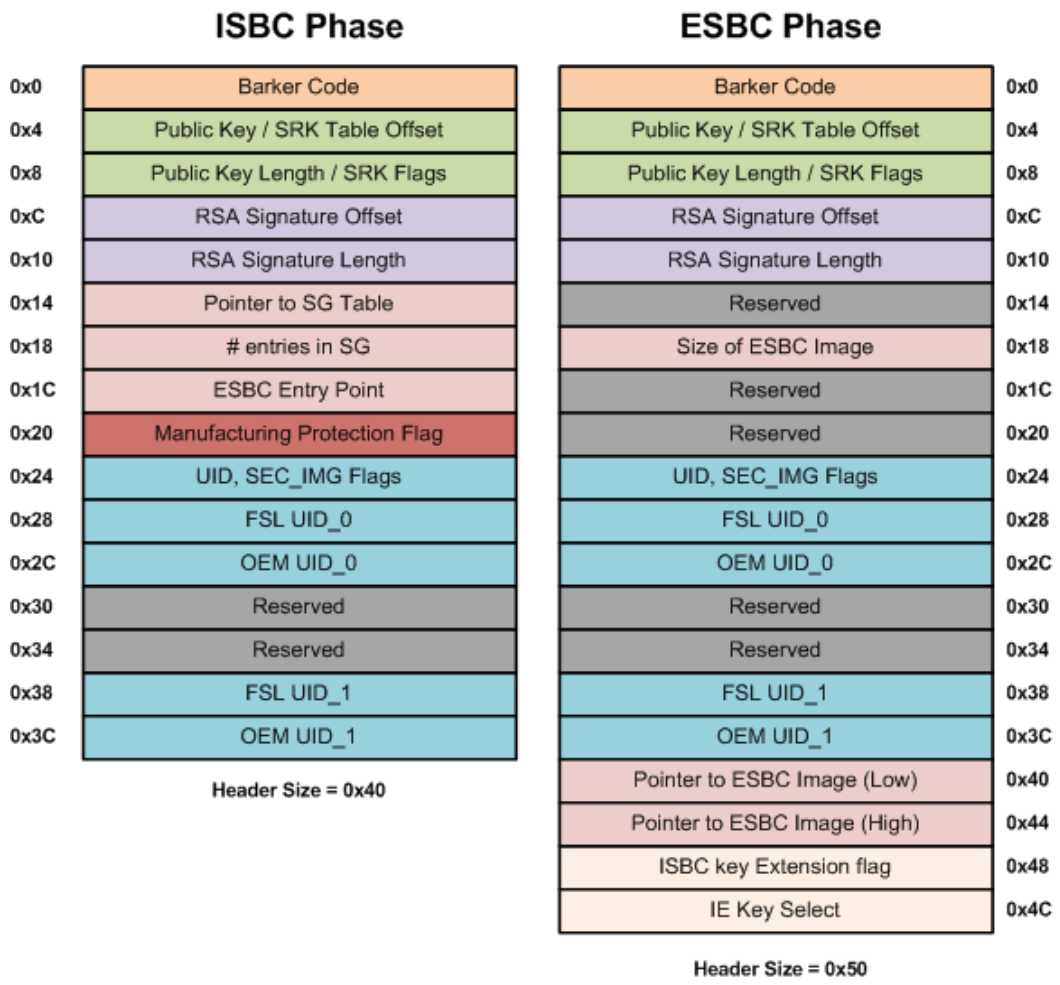


Figure 321. CSF Header for LS1043/LS1046//LS1012 (ISBC and ESBC Phase)

Table 320. CSF Header Format (LS1043/LS1046/LS1012 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	**Barker code**. This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.
0x07-0x04	If the srk_table_flag is not set : • **Public key offset**: This location contains an address which is the offset of the public key from the start of CSF header. Using this offset and the public key length, the public key is read. If srk_table_flag is set: • **Srk table offset**: This location contains an address which is the offset of the srk table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.

Table continues on the next page...

Table 320. CSF Header Format (LS1043/LS1046/LS1012 Platforms) (continued)

Offset	Data Bits [0:31]
0x08	**Srk table flag**. This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.
0x0b-0x09	If the srk_table_flag is not set : • **0x0b-0x9** -- **Public key length**: This location contains the length of the public key in bytes. If srk_table_flag is set: • **0x09** – **Key Number from srk table** which is to be used for verification. • **0x0b-0x0a** – **Number of entries in srk table**. Minimum number of entries in table = 1, Maximum = 4.
0x0f-0x0c	**RSA Signature offset**. This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x13-0x10	**RSA Signature length** in bytes.
0x17-0x14	**For ISBC Phase**: **SG Table offset** This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries is SG Table, the SG table is read. **For ESBC Phase**: Reserved
0x1b-0x18	**For ISBC Phase**: **Number of entries in SG Table** (Earlier ,Based on the Scatter gather flag in CSF Header, this location can either be treated as number of entries in SG table or ESBC image size in bytes.). **For ESBC Phase** **Size of image** to be validated
0x1f-0x1c	**For ISBC Phase**: **ESBC entry point**. ISBC transfers control to this location upon successful validation of ESBC image(s). **For ESBC Phase**: Reserved
0x21-0x20	**Manufacturing Protection Flag** Indicates if manufacturing protection has to be enabled or not in ISBC.
0x23-0x22	**Reserved** .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.)

Table continues on the next page...

Table 320. CSF Header Format (LS1043/LS1046/LS1012 Platforms) (continued)

Offset	Data Bits [0:31]
0x24	For ISBC Phase: Reserved For ESBC Phase: Reserved
0x25	**For ISBC Phase** **Secondary Image flag** Indicates if user has a secondary image available in case of failures in validating primary iamge.Valid in case of primary Images's Header. **For ESBC Phase**:Reserved
0x27-0x26	**Unique ID Usage** This location contains a flag which specifies one of these possibilities • 0x00 - No UID's present • 0x01 - FSL UID and OEM UID are present • 0x02 - Only FSL UID is present • 0x04 - Only OEM UID is present
0x2b-0x28	**NXP unique ID 0** Upper 32 bits of a unique 64 bit value, which is specific to NXP. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x2f-0x2c	**OEM unique ID 0** Upper 32 bits of a unique 64 bit value, which is specific to OEM. This value is compared with the OEM ID 0 in Secure Fuse Processor 's OEM-ID registers
0x37-0x30	Reserved
0x3b-0x38	**NXP unique ID 1** Lower 32 bits of a unique 64 bit value, which is specific to NXP. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x3f-0x3c	**OEM unique ID 1** Lower 32 bits of a unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID 1 in Secure Fuse Processor 's OEM-ID registers
0x40-0x47	**For ISBC Phase**: Not Applicable **For ESBC Phase**: 64 bit pointer to ESBC image
0x48-0x4b	**For ISBC Phase**: Not Applicable **For ESBC Phase**: **ISBC key Extension flag** If this flag is set, key to be used for validation needs to be picked up from IE Key table.

Table continues on the next page...

Table 320. CSF Header Format (LS1043/LS1046/LS1012 Platforms) (continued)

Offset	Data Bits [0:31]
0x4c-0x4f	**For ISBC Phase**: Not Applicable **For ESBC Phase**: **IE Key Select** Key Number to be used from the IE Key Table if IE flag is set.

Table 321. Scatter Gather Table Format (LS1043/LS1046/LS1012 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1.
0x04-0x07	Target where the ESBC Image 1 can be found. This field is ignored in case of PBL based SOC's.
0x08-0x0b	Source Address of ESBC Image 1
0x0c-0x0f	Destination Address of ESBC Image 1 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.
0x10-0x13	Length. This location specifies the length in bytes of the ESBC image 2.
0x14-0x17	Target where the ESBC Image 2 can be found. This field is ignored in case of PBL based SOC's.
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	Destination Address of ESBC Image 2 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.

Table 322. Signature (LS1043/LS1046/LS1012 Platforms)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

Table 323. Public key (LS1043/LS1046/LS1012 Platforms)

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

Table 324. SRK Table (LS1043/LS1046/LS1012 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

10.3.1.10 ISBC Validation Error Codes

P3/P4/P5 platforms

Table 325. ISBC Validation Failures (P3/P4/P5 platforms)

Value	Code	Definition
0x1	CPUID_NO_MATCH	ISBC is not running on CPU0
0x2	ESBC_HDR_LOC	ESBC header location is not in 3.5G space
0x4	ESBC_HEADER_BARKER	Barker code in the header is incorrect.
0x8	ESBC_HEADER_KEY_LEN	Length of public key in header is not one of the supported values.
0x10	ESBC_HEADER_SIGN_LEN	Length of RSA signature in header is not one of the supported values.
0x20	ESBC_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key is not twice the length of the RSA signature
0x40	ESBC_HEADER_SG_TABLE_ADDR_NULL	SG table/ESBC image address (0x14-0x17 in CSF Header) is null
0x80	ESBC_HEADER_SG_TABLE_ADDR_NOT_IN_3_5G	SG table/ESBC image address (0x14-0x17 in CSF Header) is beyond 3.5G
0x100	ESBC_HEADER_KEY_MOD_1	Most significant bit of modulus in header is zero.

Table continues on the next page...

Table 325. ISBC Validation Failures (P3/P4/P5 platforms) (continued)

Value	Code	Definition
0x200	ESBC_HEADER_KEY_MOD_2	Modulus in header is even number
0x400	ESBC_HEADER_SIG_KEY_MOD	Signature value is greater than modulus in header
0x800	ESBC_HEADER_SG_ENTRIES_NUL	SG Table contains zero entries
0x1000	ESBC_HEADER_SG_ENTRIES_NOT_IN_3_5G	Address in SG entry in not in 3.5G
0x2000	ESBC_HEADER_SG_ESBC_EP	ESBC entry point in header not within ESBC address range
0x4000	HASH_COMPARE_KEY	Super Root Key Hash Comparison failure. Mismatch in the hash of the public key as present in the header with the value in the SRK HASH fuse.
0x8000	HASH_COMPARE_EM	RSA signature check failure. Signature provided by you in the header doesn't match with the signature of the ESBC image generated by ISBC. The ESBC image loaded by you may be different than the image used while generating the signature(using CST)
0x10000	SSM_CHECKSTS	SEC_MON State Machine not in CHECK state at start of ISBC. Some Security violation could have occurred. Details can be found in P4080 Reference Manual.
0x20000	SSM_TRUSTSTS	SEC_MON State Machine not in TRUSTED state at end of ISBC.
0x40000	FSL_UID	FSL_UID in ESBC Header did not match the FSL_UID in SFP
0x80000	OEM_UID	OEM_UID in ESBC Header did not match the OEM_UID in SFP
0x100000	BAD_ADDRESS	A Data / Instruction TLB Exception occurred during ISBC execution
0x200000	MISC	E500mc exception other than TLB
0x400000	ESBC_HEADER_SG_ENTIRES_BAD	SG Table too large (too many entries)

NOTE

For error codes 0x2 - 0x2000 i,e errors in the ESBC Header, check the value of that particular field by dumping the header.

B4/T1/T2/T4/LS1/LS1043/LS1046/LS1012 platforms

Errors in the system can be of following types:

1. Core Exceptions
2. System State Failures

3. Header Checking Failures
    a. General Failures
    b. Key/Signature/UID related errors
4. Verification Failures
5. SEC/PAMU errors

Table 326. Core Exceptions (LS1 platform)

Value	Code	Definition
0x1	ERROR_UNDEFINED_INSTRUCTION	Occurs if neither the processor nor any attached co-processor recognizes the currently executing instruction.
0x2	ERROR_SWI	Software Interrupt is a user-defined interrupt instruction. It allows a program running in User mode, for example, to request privileged operations that run in Supervisor mode.
0x3	ERROR_PREFETCH_ABORT	Occurs when the processor attempts to execute an instruction that has been prefetched from an illegal address.
0x4	ERROR_DATA_ABORT	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
0x5	ERROR_IRQ	Occurs when the processor external interrupt request pin is asserted (LOW) and IRQ interrupts are enabled.
0x6	ERROR_FIQ	Occurs when the processor external fast interrupt request pin is asserted (LOW) and FIQ interrupts are enabled.

Table 327. Core Exceptions (LS1043/LS1046/LS1012 platforms)

Error Code	Value
Current EL with SP0	
ERROR_EXCEPTION_SYNC_SP0	0x01
ERROR_EXCEPTION_IRQ_SP0	0x02
ERROR_EXCEPTION_FIQ_SP0	0x03
ERROR_EXCEPTION_SERROR_SP0	0x04
Current EL with SPx	
ERROR_EXCEPTION_SYNC_SPX	0x05
ERROR_EXCEPTION_IRQ_SPX	0x06
ERROR_EXCEPTION_FIQ_SPX	0x07
ERROR_EXCEPTION_SERROR_SPX	0x08
Lower EL using AArch64	
ERROR_EXCEPTION_SYNC_L64	0x11
ERROR_EXCEPTION_IRQ_L64	0x12

Table continues on the next page...

Table 327. Core Exceptions (LS1043/LS1046/LS1012 platforms) (continued)

ERROR_EXCEPTION_FIQ_L64	0x13
ERROR_EXCEPTION_SERROR_L64	0x14
Lower EL using AArch32	
ERROR_EXCEPTION_SYNC_L32	0x15
ERROR_EXCEPTION_IRQ_L32	0x16
ERROR_EXCEPTION_FIQ_L32	0x17
ERROR_EXCEPTION_SERROR_L32	0x18

Table 328. Core Exceptions (B4/T1/T2/T4 platforms)

Value	Code	Definition
0x1	ERROR_MACHINECHECK	Machine check Exception
0x2	ERROR_DSI	DSI Exception
0x3	ERROR_ISI	ISI Exception
0x4	ERROR_CRITICAL	Critical Exception
0x5	ERROR_ALIGN	Alignment Exception
0x6	ERROR_PROG	Program Exception
0x13	ERROR_DATA_TLB	Data TLB Miss
0x14	ERROR_INST_TLB	Instruction TLb Miss
0x20	ERROR_MISC	Any other exception

Table 329. System State Failures (B4/T1/T2/T4/LS1/LS1043/LS1046/LS1012 platforms)

Value	Code	Definition
0x100	ERROR_CORE_NON_ZERO	ISBC is not running on CPU0
0x101	ERROR_STATE_NOT_CHECK	SEC_MON State Machine not in CHECK state at start of ISBC. Some Security violation could have occurred.
0x102	ERROR2_STATE_NOT_CHECK	SEC_MON State Machine not in CHECK state, when trying to transition it to Trusted/Non Secure/Soft Fail state
0x103	ERROR_SSM_TRUSTSTS	SEC_MON State Machine not in TRUSTED state at end of ISBC.

Table 330. General Header Checking Failures (B4/T1/T2/T4/LS1/LS1043/LS1046/LS1012 platforms)

Value	Code	Definition
0x301	ERROR_ESBC_HDR_LOC	ESBC header location is not in 3.5G space
0x302	ERROR_ESBC_HEADER_BARKER	Barker code in the header is incorrect.
0x303	ERROR_ESBC_HEADER_SG_ENTRIES_NOT_IN_3_5G	SG table/ESBC image address (header address + image offset in sg table) is beyond 3.5G
0x303	ERROR_ESBC_HEADER_SG_ENTRIES_ON_OCRAM	One Entry in the SG table is on OCRAM
0x304	ERROR_ESBC_HEADER_SG_ESBC_EP	ESBC entry point in header not within ESBC address range
0x305	ERROR_SGL_ENTIRES_NOT_SUPPORTED	Number of entries in SG table exceeds maximum limit i.e 8
0x306	ERROR_ESBC_HEADER_HKAREA_LEN_ZERO	Houskeeping area not provided in header
0x307	ERROR_ESBC_HEADER_HKAREA_NOT_IN_3_5G	House keeping area not in 3.5G boundary
0x308	ERROR_ESBC_HEADER_HKAREA_LEN_INSUFFICIENT	Housekeeping area length provided is not sufficient.
0x309	ERROR_SG_TABLE_NOT_IN_3_5	SG Table is not in 3.5G boundary
0x309	ERROR_SG_TABLE_ON_OCRAM	SG table is on OCRAM
0x310	ERROR_ESBC_HEADER_HKAREA_NOT_4K_ALIGNED	House keeping area is not aligned to 4K boundary
0x311	ERROR_SGL_ENTRIES_SIZE_ZERO	SG table has entry with size zero.

Table 331. Key/Signature/UID related errors (B4/T1/T2/T4/LS1/LS1043/LS1046/LS1012 platforms)

Value	Code	Definition
0x320	ERROR_ESBC_HEADER_KEY_LEN	Length of public key in header is not one of the supported values.
0x321	ERROR_ESBC_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key is not twice the length of the RSA signature
0x322	ERROR_ESBC_HEADER_KEY_MOD_1	Most significant bit of modulus in header is zero.
0x323	ERROR_ESBC_HEADER_KEY_MOD_2	Modulus in header is even number
0x324	ERROR_ESBC_HEADER_SIG_KEY_MOD	Signature value is greater than modulus in header
0x325	ERROR_FSL_UID	FSL_UID in ESBC Header did not match the FSL_UID in SFP if fsl uid flag Is 1
0x326	ERROR_OEM_UID	OEM_UID in ESBC Header did not match the OEM_UID in SFP if oem uid flag is 1.

Table continues on the next page...

Table 331. Key/Signature/UID related errors (B4/T1/T2/T4/LS1/LS1043/LS1046/LS1012 platforms) (continued)

Value	Code	Definition
0x327	ERROR_INVALID_SRK_NUM_ENTRY	Number of entries field in CSF Header is > 4(This is when srk_flag in header is 1)
0x328	ERROR_INVALID_KEY_NUM	Key number to be used from srk table is not present in table. (This is when srk_flag in header is 1)
0x329	ERROR_KEY_REVOKED	Key selected from srk table has been revoked(This is when srk_flag in header is 1)
0x32a	ERROR_INVALID_SRK_ENTRY_KEYLEN	Key length specified in one of the entries in srk table is not one of the supported values (This is when srk_flag in header is 1)
0x32b	ERROR_SRK_TBL_NOT_IN_3_5	SRK Table is not in 3.5G boundary (This is when srk_flag in header is 1)
0x32b	ERROR_SRK_TBL_ON_OCRAM	SRK Table is on OCRAM
0x32c	ERROR_KEY_NOT_IN_3_5G	Key is not in 3.5G boundary
0x32c	ERROR_KEY_ON_OCRAM	Key on OCRAM

Table 332. Verification Failures (B4/T1/T2/T4/LS1/LS1043/LS1046/LS1012 platforms)

Value	Code	Definition
0x340	ERROR_HASH_COMPARE_KEY	Super Root Key Hash Comparison failure. Mismatch in the hash of the public key/srk table as present in the header with the value in the SRK HASH fuse.
0x341	ERROR_HASH_COMPARE_EM	RSA signature check failure. Signature provided by you in the header doesn't match with the signature of the ESBC image generated by ISBC. The ESBC image loaded by you may be different than the image used while generating the signature(using CST)

Table 333. SEC/PAMU Failures (B4/T1/T2/T4/LS1/LS1043/LS1046/LS1012 platforms)

Value	Code	Definition
0x700	ERROR_SEC_ENQ	Error when enqueuing to SEC
0x701	ERROR_SEC_DEQ	Sec Block returned some error when dequeuing from it.
0x702	ERROR_SEC_DEQ_TO	Timeout when trying to deq from SEC
0x800	ERROR_PAMU	Error while programming PAACT/SPAACT tables in PAMU (For PowerPC platforms only)

10.3.1.11 ESBC Validation Error Codes

For trust arch version 1.x and 2.x.

Table 334. ESBC Validation Failures

Value	Code	Definition
0x0	ERROR_ESBC_CLIENT_MAX	NULL
0x4	ERROR_ESBC_CLIENT_HEADER_BARKER	Wrong barker code in header
0x8	ERROR_ESBC_CLIENT_HEADER_KEY_LEN	Wrong public key length in header
0x10	ERROR_ESBC_CLIENT_HEADER_SIG_LEN	Wrong signature length in header
0x11	ERROR_ESBC_CLIENT_HEADER_KEY_REVOKED	Key used to sign the image revoked
0x12	ERROR_ESBC_CLIENT_HEADER_INVALID_SRK_NUM_ENTRY	Wrong key entry
0x13	ERROR_ESBC_CLIENT_HEADER_INVALID_KEY_NUM	Selected key no. not in SRK table
0x14	ERROR_ESBC_CLIENT_HEADER_INV_SRK_ENTRY_KEYLEN	Unsupported key length of key in SRK table
0x15	ERROR_ESBC_CLIENT_HEADER_IE_KEY_REVOKED	Selected key in IE key table revoked
0x16	ERROR_ESBC_CLIENT_HEADER_INVALID_IE_NUM_ENTRY	Wrong IE Key entry
0x17	ERROR_ESBC_CLIENT_HEADER_INVALID_IE_KEY_NUM	Selected key no. not in IE Key table
0x18	ERROR_ESBC_CLIENT_HEADER_INV_IE_ENTRY_KEYLEN	Unsupported key length of key in IE Key table
0x19	ERROR_IE_TABLE_NOT_FOUND	information about IE table missing
0x20	ERROR_ESBC_CLIENT_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key length not twice of signature length
0x21	ERROR_KEY_TABLE_NOT_FOUND	SRK Key/key table not found
0x40	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_1	Public key Modulus most significant bit not set
0x80	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_2	Public key Modulus in header not odd
0x100	ERROR_ESBC_CLIENT_HEADER_SIG_KEY_MOD	Signature not less than modulus
0x200	ERROR_ESBC_CLIENT_HEADER_SG_ESBC_EP	Entry Point error

Table continues on the next page...

Table 334. ESBC Validation Failures (continued)

Value	Code	Definition
0x400	ERROR_ESBC_CLIENT_HASH_COMPARE_KEY	Public key hash comparison failed
0x800	ERROR_ESBC_CLIENT_HASH_COMPARE_EM	RSA verification failed
0x1000	ERROR_ESBC_CLIENT_SSM_TRUSTSTS	SNVS not in TRUSTED state
0x2000	ERROR_ESBC_CLIENT_BAD_ADDRESS	Bad address error
0x4000	ERROR_ESBC_CLIENT_MISC	Miscallaneous error
0x8000	ERROR_ESBC_CLIENT_HEADER_SG_ENTIRES_BAD	Incorrect entries in SG table
0x10000	ERROR_ESBC_CLIENT_HEADER_SG	No SG support
0x20000	ERROR_ESBC_CLIENT_HEADER_IMG_SIZE	Invalid Image size
0x40000	ERROR_ESBC_WRONG_CMD	Failure in command/Unknown command/Wrong arguments of boot script.
0x80000	ERROR_ESBC_MISSING_BOOTM	Bootm command missing from boot script.

10.3.1.12 Trust Architecture and SFP Information

SoC	Trust Arch. Version	SFP Version	POVDD	DRVR		OTPMK		SNVS/SFP Register to check Hamming Error
				Algo (CST)	Register to check Hamming Error	Algo (CST)	Register to check Hamming Error	
P4080 rev1	1	1	1.5 V	A	None/ Simulation	1	SNVS	SecMon_HP Status (HPSR)
P4080 rev2	1	1.1	1.5 V	B	None/ Simulation	1	SNVS	
P4080 rev3	1	2.2	1.5 V	B	None/ Simulation	1	SNVS	
P3041	1.1	2	1.5 V	B	None/ Simulation	1	SNVS	
P5020	1.1	2	1.5 V	B	None/ Simulation	1	SNVS	
P5040	1.1	2.2	1.5 V	B	None/ Simulation	1	SNVS	

Table continues on the next page...

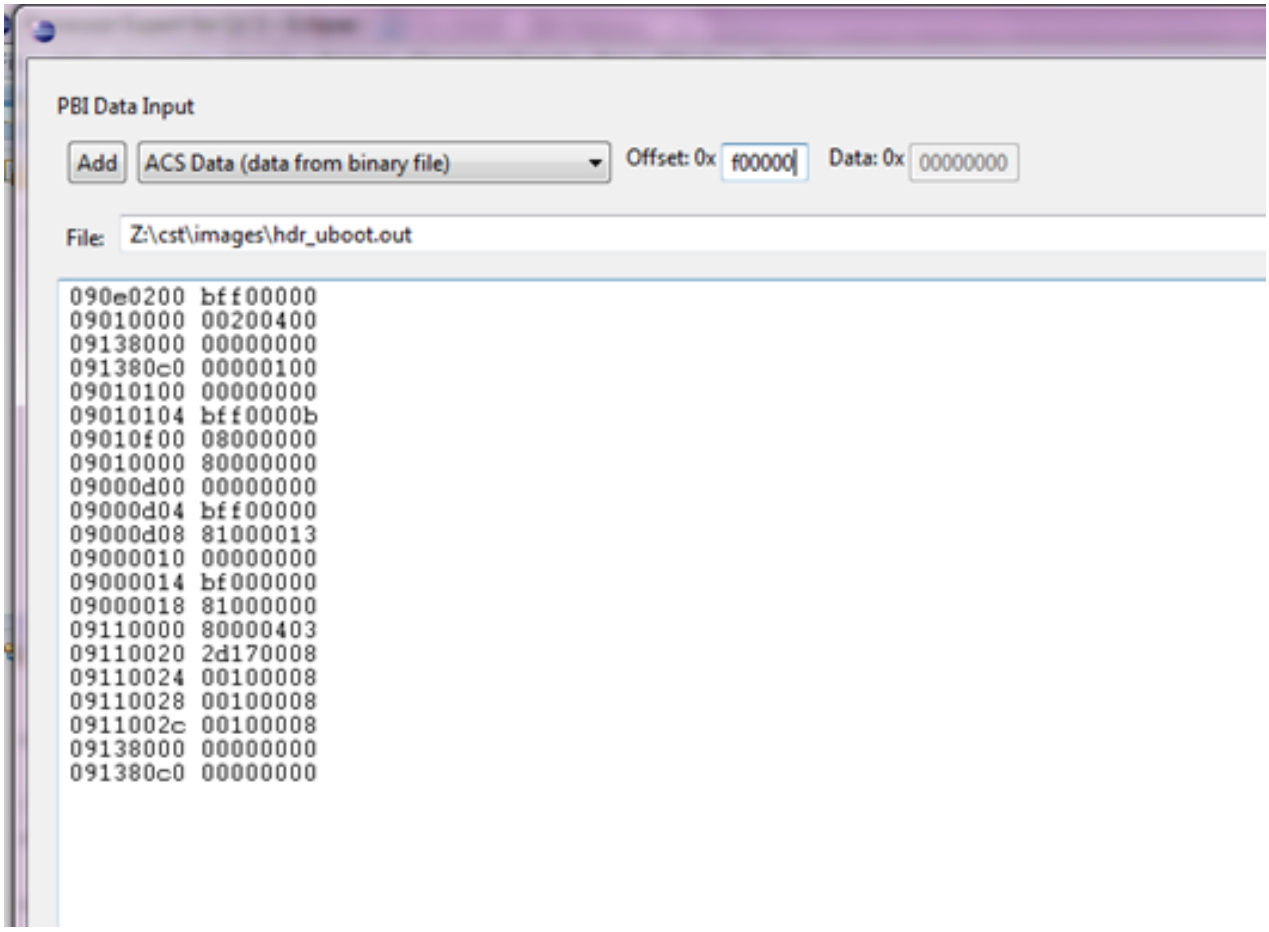
Table continued from the previous page...

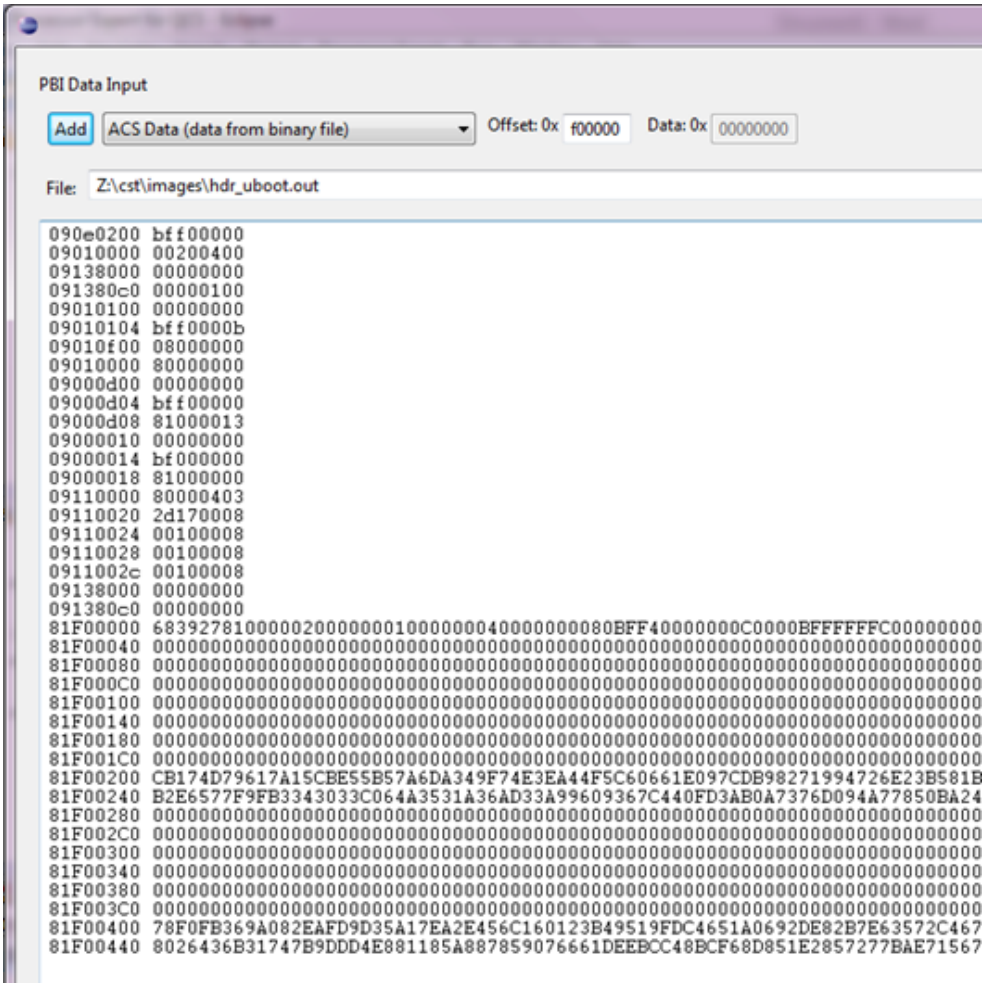
P5021	1.1	2.2	1.5 V	B	None/ Simulation	1	SNVS	
T4240 rev1	2	3.1	1.89 V	A	SFP	2	SFP	SFP Secret Value Hamming Error Status Register (SFP_SVHESR)
T4240 rev2	2	3.2	1.89 V	A	SFP	2	SFP	
B4860 rev1	2	3.1	1.89 V	A	SFP	2	SFP	
B4860 rev2	2	3.2	1.89 V	A	SFP	2	SFP	
T2080	2	3.2	1.89 V	A	SFP	2	SFP	
T1040	2	3.2	1.89 V	A	SFP	2	SFP	
T1023	2	3.2	1.89 V	A	SFP	2	SFP	
LS1020A	2.1	3.3	1.89 V	A	SFP	2	SFP	
LS1043A	2.1	3.3	1.89 V	A	SFP	2	SFP	
LS1046A	2.1	3.3	1.89 V	A	SFP	2	SFP	
LS1012	2.1	3.3	1.89 V	A	SFP	2	SFP	

10.3.1.13 Using QCVS Tool (Secure Boot From NAND)

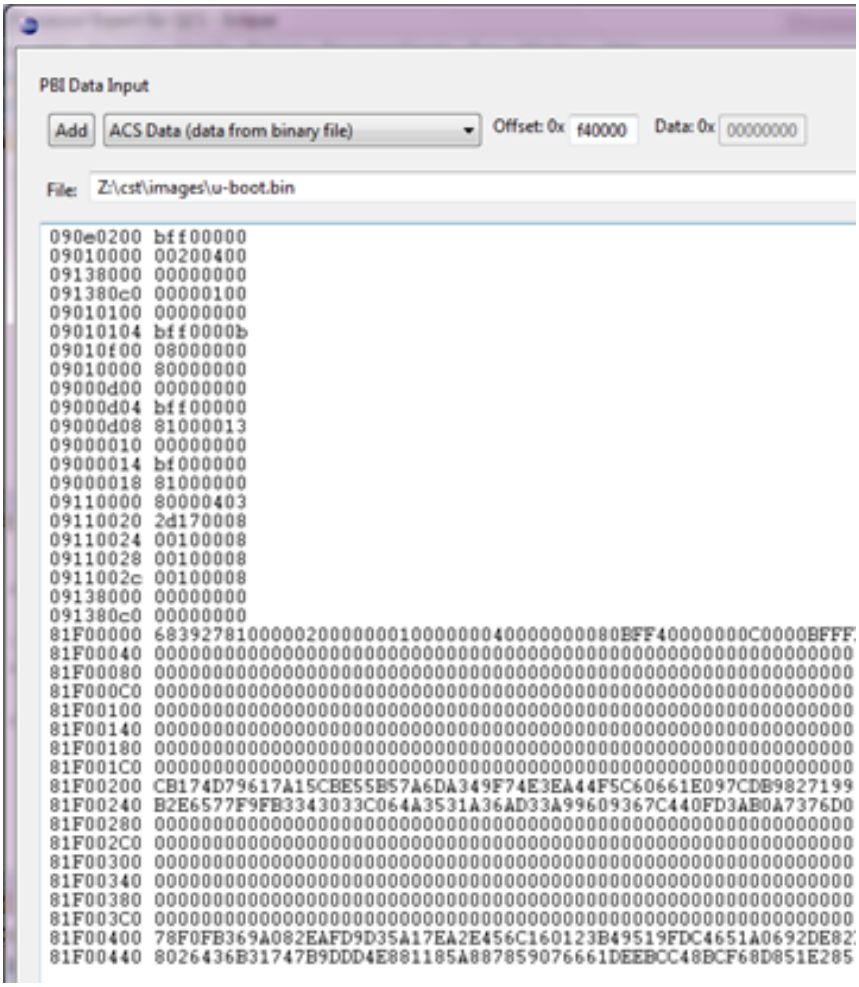
Use NXP's QCVS tool for adding the hdr_uboot.out and u-boot.bin in terms of ALT_CONFIG_WRITE PBI commands at required addresses. The below screenshots describe the usage of QCVS Tool.

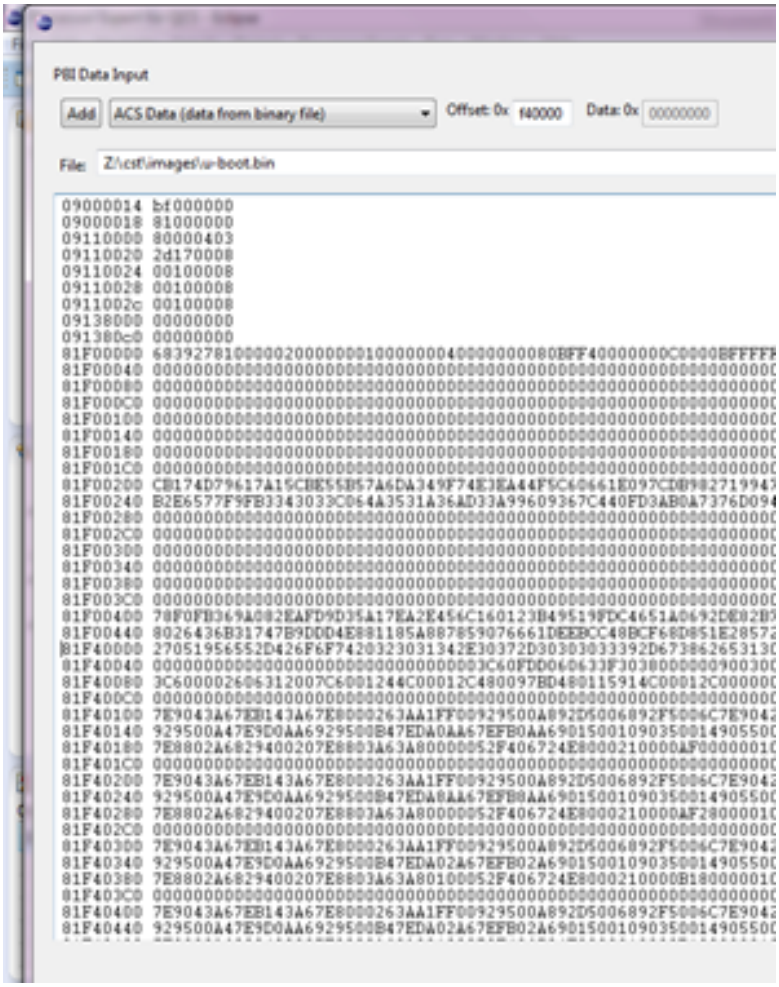
1. Import rcw.bin from SDK in QCVS Tool.
2. Add ACS Data hdr_uboot.out @ 0xF00000.


PBI Data Input
Add
ACS Data (data from binary file)
Offset: 0x f00000
Data: 0x 00000000
File: Z:\cst\images\hdr_uboot.out
090e0200 bff00000
09010000 00200400
09138000 00000000
091380c0 00000100
09010100 00000000
09010104 bff0000b
09010f00 08000000
09010000 80000000
09000d00 00000000
09000d04 bff00000
09000d08 81000013
09000010 00000000
09000014 bf000000
09000018 81000000
09110000 80000403
09110020 2d170008
09110024 00100008
09110028 00100008
0911002c 00100008
09138000 00000000
091380c0 00000000

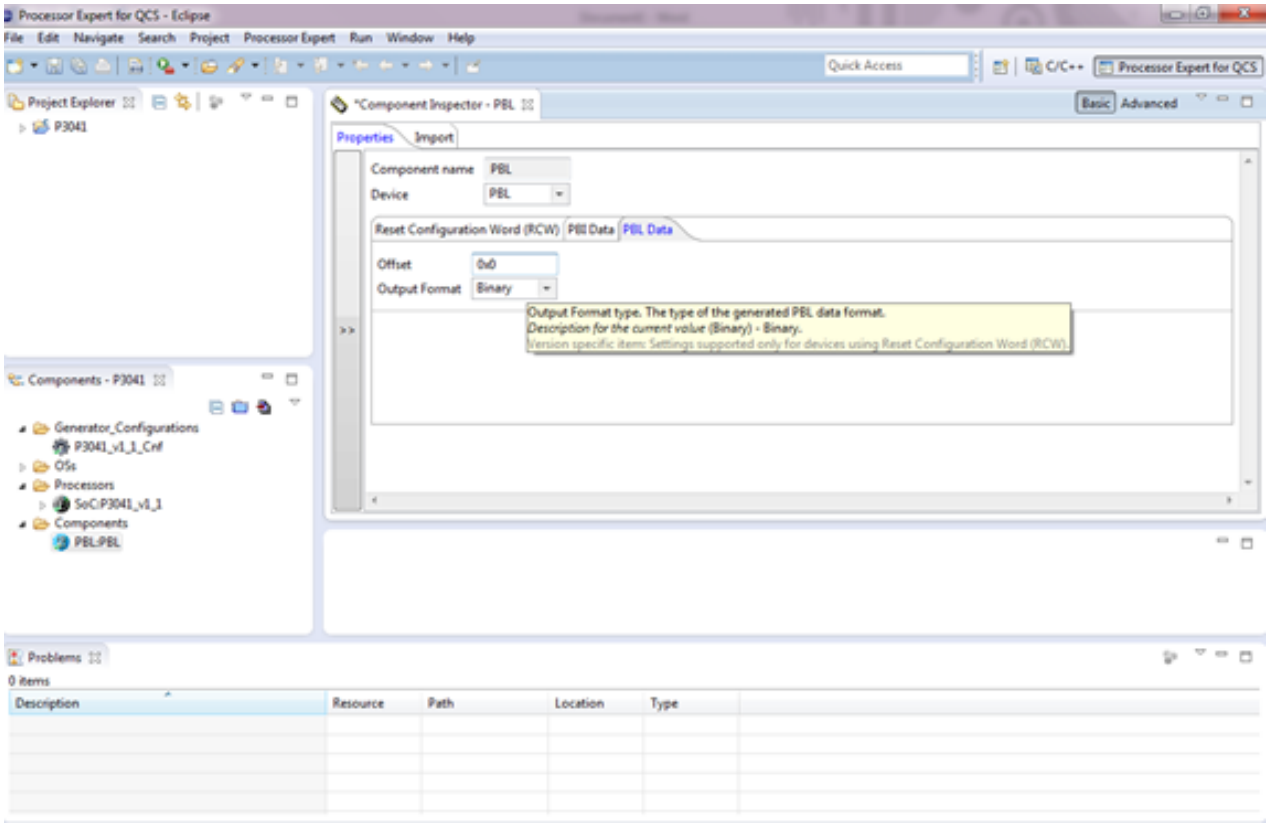


3. Add ACS Data u-boot.bin @0xF40000.

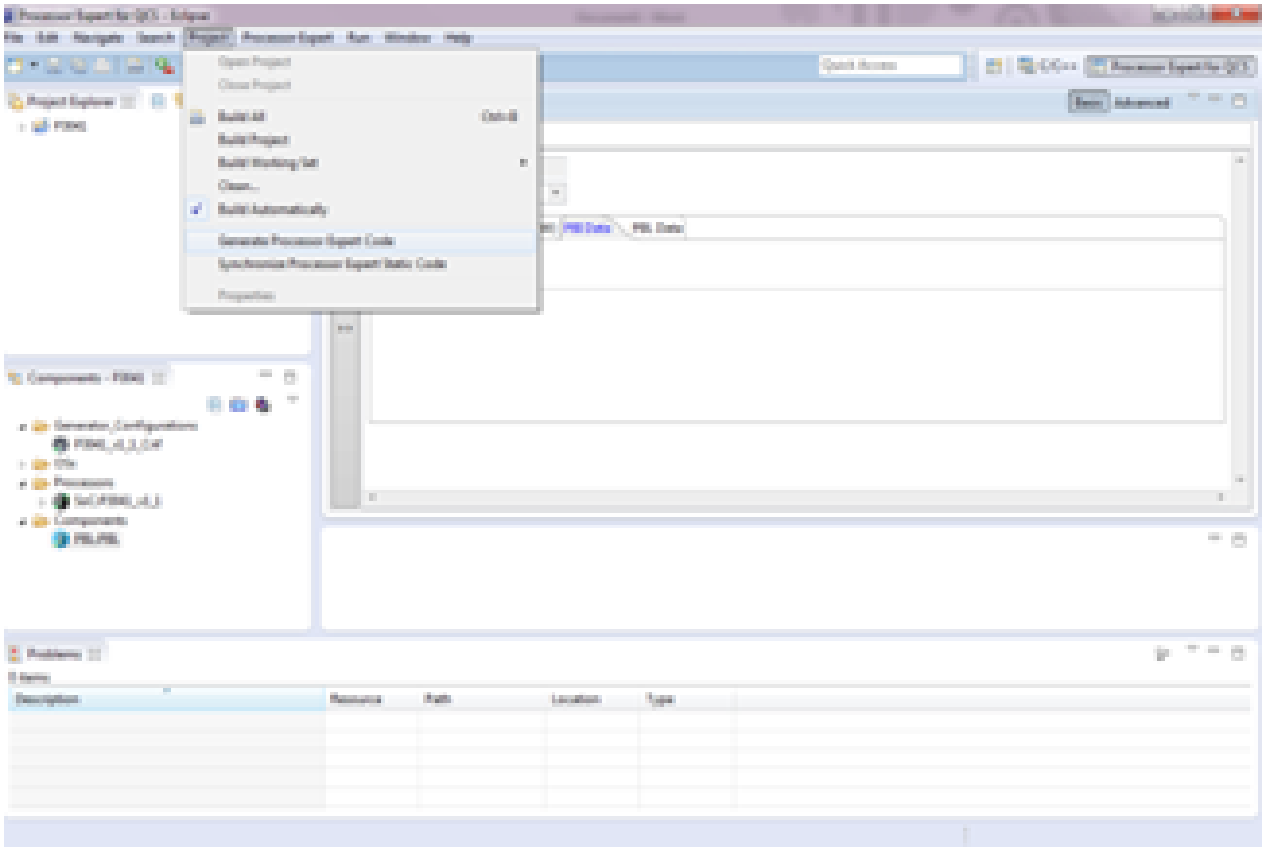

PBI Data Input
Add
ACS Data (data from binary file)
Offset: 0x f40000
Data: 0x 00000000
File: Z:\cst\images\u-boot.bin
090e0200 bff00000
09010000 00200400
09138000 00000000
091380c0 00000100
09010100 00000000
09010104 bff0000b
09010f00 08000000
09010000 80000000
09000d00 00000000
09000d04 bff00000
09000d08 81000013
09000010 00000000
09000014 bf000000
09000018 81000000
09110000 80000403
09110020 2d170008
09110024 00100008
09110028 00100008
0911002c 00100008
09138000 00000000
091380c0 00000000
81F00000 6839278100000200000001000000040000000080BFF40000000C0000BFFF
81F00040 0000000000000000000000000000000000000000000000000000000000000000
81F00080 0000000000000000000000000000000000000000000000000000000000000000
81F000C0 0000000000000000000000000000000000000000000000000000000000000000
81F00100 0000000000000000000000000000000000000000000000000000000000000000
81F00140 0000000000000000000000000000000000000000000000000000000000000000
81F00180 0000000000000000000000000000000000000000000000000000000000000000
81F001C0 0000000000000000000000000000000000000000000000000000000000000000
81F00200 CB174D79617A15CBE55B57A6DA349F74E3EA44F5C60661E097CDB9827199
81F00240 B2E6577F9FB3343033C064A3531A36AD33A99609367C440FD3AB0A7376D0
81F00280 0000000000000000000000000000000000000000000000000000000000000000
81F002C0 0000000000000000000000000000000000000000000000000000000000000000
81F00300 0000000000000000000000000000000000000000000000000000000000000000
81F00340 0000000000000000000000000000000000000000000000000000000000000000
81F00380 0000000000000000000000000000000000000000000000000000000000000000
81F003C0 0000000000000000000000000000000000000000000000000000000000000000
81F00400 78F0FB369A082EAFD9D35A17EA2E456C160123B49519FDC4651A0692DE82
81F00440 8026436B31747B9DDD4E881185A887859076661DEEBCC48BCF68D851E285



4. Make Sure that the output format is Binary.



5. Generate Processor Expert Code to get PBL.bin.



.

10.3.1.14 Appendix P3/P4/P5/T1_T2_T4 Secure Boot demo

P3/P4/P5/T1_T2_T4 Platforms

Table 335. Memory Map for P3/P4/P5/T1_T2_T4

Address(NOR vBank 0)	Address (NOR Alternate Bank)[21]	Definition (Chain of Trust)	Definition (Chain of Trust with Confidentiality)	Size Reserved (KB)
E8000000	EC000000	RCW	RCW	128
E8020000	EC020000	uImage	uImage	8064
E8800000	EC800000	DTB	DTB	1024
E8900000	EC900000			1024
E8A00000	ECA00000	Bootscript	Bootscript	1024
E8B00000	ECB00000	ESBC U-Boot HEADER	ESBC U-Boot HEADER	1024
E8C00000	ECC00000			2048
E8E00000	ECE00000	Bootscript Header	Bootscript Header	1024
E8F00000	ECF00000			1024
E9000000	ED000000	uImage Heaer		1024
E9100000	ED100000	DTB Header		1024
E9200000	ED200000	Rootfs Header		1024
E9300000	ED300000	Rootfs	Rootfs	29696
EC020000	E8020000		uImage (ENCAPSULATED)	8064
EC800000	E8800000		DTB (ENCAPSULATED)	1024
ED300000	E9300000		Rootfs (ENCAPSULATED)	29696
EFF20000	EBF20000	u-boot env	u-boot env (current bank)	128
EFF40000	EBF40000	u-boot	u-boot	768

[21] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

Chain Of Trust Boot Script Used as per Address Map

```
esbc_validate 0xE9000000
esbc_validate 0xE9100000
esbc_validate 0xE9200000
bootm 0xE8020000 0xE9300000 0xE8800000
```

Useful U-Boot and CCS Commands for P3/P4/P5

```
protect off all;
setenv dir <tftp_path>
tftp 1000000 $dir/rcw.bin; erase EC000000 +$filesize; cp.b 1000000 EC000000 $filesize;
tftp 1000000 $dir/hdr_uboot.out;erase ECB00000 +$filesize; cp.b 1000000 ECB00000 $filesize;
tftp 1000000 $dir/u-boot.bin;erase EBF40000 +$filesize; cp.b 1000000 EBF40000 $filesize ;

tftp 1000000 $dir/hdr_bs.out;erase 0xECE00000 +$filesize;cp.b 1000000 0xECE00000 $filesize;
tftp 1000000 $dir/hdr_linux.out;erase 0xED000000 +$filesize;cp.b 1000000 0xED000000 $filesize;
tftp 1000000 $dir/hdr_rootfs.out;erase 0xED200000 +$filesize;cp.b 1000000 0xED200000 $filesize;
tftp 1000000 $dir/hdr_dtb.out;erase 0xED100000 +$filesize;cp.b 1000000 0xED100000 $filesize;

tftp 1000000 $dir/rootfs;erase 0xED300000 +$filesize;cp.b 1000000 0xED300000 $filesize;
tftp 1000000 $dir/bootscript;erase 0xECA00000 +$filesize;cp.b 1000000 0xECA00000 $filesize;
tftp 1000000 $dir/uImage.bin;erase 0xEC020000 +$filesize;cp.b 1000000 0xEC020000 $filesize;
tftp 1000000 $dir/uImage.dtb;erase 0xEC800000 +$filesize;cp.b 1000000 0xEC800000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_chain p3040
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  0 0xfe314014 4 0 1
ccs::display_mem  0 0xfe0e0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem 0 0xfe0e807c 4 0 <SRKH1>
ccs::write_mem 0 0xfe0e8080 4 0 <SRKH2>
ccs::write_mem 0 0xfe0e8084 4 0 <SRKH3>
ccs::write_mem 0 0xfe0e8088 4 0 <SRKH4>
ccs::write_mem 0 0xfe0e808c 4 0 <SRKH5>
ccs::write_mem 0 0xfe0e8090 4 0 <SRKH6>
ccs::write_mem 0 0xfe0e8094 4 0 <SRKH7>
ccs::write_mem 0 0xfe0e8098 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem 0 0xfe0e00e4 4 0 0x00000001
```

Useful U-Boot and CCS Commands for T1_T2_T4

```
protect off all;
setenv dir <tftp_path>
tftp 1000000 $dir/rcw.bin; erase EC000000 +$filesize; cp.b 1000000 EC000000 $filesize;
tftp 1000000 $dir/hdr_uboot.out;erase ECB00000 +$filesize; cp.b 1000000 ECB00000 $filesize;
tftp 1000000 $dir/u-boot.bin;erase EBF40000 +$filesize; cp.b 1000000 EBF40000 $filesize ;

tftp 1000000 $dir/hdr_bs.out;erase 0xECE00000 +$filesize;cp.b 1000000 0xECE00000 $filesize;
tftp 1000000 $dir/hdr_linux.out;erase 0xED000000 +$filesize;cp.b 1000000 0xED000000 $filesize;
tftp 1000000 $dir/hdr_rootfs.out;erase 0xED200000 +$filesize;cp.b 1000000 0xED200000 $filesize;
tftp 1000000 $dir/hdr_dtb.out;erase 0xED100000 +$filesize;cp.b 1000000 0xED100000 $filesize;
```

```
tftp 1000000 $dir/rootfs;erase 0xED300000 +$filesize;cp.b 1000000 0xED300000 $filesize;
tftp 1000000 $dir/bootscript;erase 0xECA00000 +$filesize;cp.b 1000000 0xECA00000 $filesize;
tftp 1000000 $dir/uImage.bin;erase 0xEC020000 +$filesize;cp.b 1000000 0xEC020000 $filesize;
tftp 1000000 $dir/uImage.dtb;erase 0xEC800000 +$filesize;cp.b 1000000 0xEC800000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_chain {t4240 j2i2cs}
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  0 0xfe314014 4 0 1
ccs::display_mem  0 0xfe0e0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem 0 0xfe0e823c 4 0 <SRKH1>
ccs::write_mem 0 0xfe0e8240 4 0 <SRKH2>
ccs::write_mem 0 0xfe0e8244 4 0 <SRKH3>
ccs::write_mem 0 0xfe0e8248 4 0 <SRKH4>
ccs::write_mem 0 0xfe0e824c 4 0 <SRKH5>
ccs::write_mem 0 0xfe0e8250 4 0 <SRKH6>
ccs::write_mem 0 0xfe0e8254 4 0 <SRKH7>
ccs::write_mem 0 0xfe0e8258 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem 0 0xfe0e00e4 4 0 0x00000001
```

10.3.1.15 Appendix B4 Secure Boot demo

Boot Source

Table 336. Memory map for B4 platforms

Address(NOR vBank 0)	Address (NOR Alternate Bank)[22]	Definition (Chain of Trust)	Definition (Chain of Trust with Confidentiality)	Size Reserved (KB)
EC000000	EE000000	RCW	RCW	128
EC020000	EE020000	uImage	uImage	7040
EC700000	EE700000	Bootscript	Bootscript	1024
EC800000	EE800000	DTB	DTB	1024
EC900000	EE900000			1024
ECA00000	EEA00000			1024
ECB00000	EEB00000	ESBC U-Boot HEADER	ESBC U-Boot HEADER	1024

Table continues on the next page...

[22] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

Table 336. Memory map for B4 platforms (continued)

Address(NOR vBank 0)	Address (NOR Alternate Bank)[22]	Definition (Chain of Trust)	Definition (Chain of Trust with Confidentiality)	Size Reserved (KB)
ECC00000	EEC00000	Bootscript Header	Bootscript Header	1024
ECD00000	EED00000	uImage Heaer		1024
ECE00000	EEE00000	Rootfs Header		1024
ECF00000	EEF00000	DTB Header		1024
ED300000	EF300000		DTB (Encapsulated)	1024
ED500000	EF500000		uImage (Encapsulated)	10496
EE020000	EC020000		rootfs/ rootfs (ENCAPSULATED)	31232
EFF20000	EDF20000	u-boot env	u-boot env	128
EFF40000	EDF40000	u-boot	u-boot	768

NOTE

To use 512KB u-boot, change u-boot address from xxx40000 to xxx80000

Chain Of Trust Boot Script Used as per Address Map

```
esbc_validate 0xECD00000
esbc_validate 0xECE00000
esbc_validate 0xECF00000
bootm 0xEC020000 0xEE020000 0xEC800000
```

Refer to the ‘Product Execution’ Section and blow the required fuses as mentioned in it before executing the following commands

Useful U-Boot and CCS Commands

```
protect off all;
setenv dir <tftp_path>
tftp 1000000 $dir/rcw.bin; erase EE000000 +$filesize; cp.b 1000000 EE000000 $filesize;
tftp 1000000 $dir/hdr_uboot.out;erase EEB00000 +$filesize; cp.b 1000000 EEB00000 $filesize;
tftp 1000000 $dir/u-boot.bin;erase EDF40000 +$filesize; cp.b 1000000 EDF40000 $filesize ;

tftp 1000000 $dir/hdr_bs.out;erase 0xEEC00000 +$filesize;cp.b 1000000 0xEEC00000 $filesize;
tftp 1000000 $dir/hdr_linux.out;erase 0xEED00000 +$filesize;cp.b 1000000 0xEED00000 $filesize;
tftp 1000000 $dir/hdr_rootfs.out;erase 0xEEE00000 +$filesize;cp.b 1000000 0xEEE00000 $filesize;
```

[22] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

```
tftp 1000000 $dir/hdr_dtb.out;erase 0xEEF00000 +$filesize;cp.b 1000000 0xEEF00000 $filesize;

tftp 1000000 $dir/rootfs;erase 0xEC020000 +$filesize;cp.b 1000000 0xEC020000 $filesize;
tftp 1000000 $dir/bootscript;erase 0xEE700000 +$filesize;cp.b 1000000 0xEE700000 $filesize;
tftp 1000000 $dir/uImage.bin;erase 0xEE020000 +$filesize;cp.b 1000000 0xEE020000 $filesize;
tftp 1000000 $dir/uImage.dtb;erase 0xEE800000 +$filesize;cp.b 1000000 0xEE800000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_chain {b4860 j2i2cs}
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  0 0xfe314014 4 0 1
ccs::display_mem  0 0xfe0e0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem 0 0xfe0e823c 4 0 <SRKH1>
ccs::write_mem 0 0xfe0e8240 4 0 <SRKH2>
ccs::write_mem 0 0xfe0e8244 4 0 <SRKH3>
ccs::write_mem 0 0xfe0e8248 4 0 <SRKH4>
ccs::write_mem 0 0xfe0e824c 4 0 <SRKH5>
ccs::write_mem 0 0xfe0e8250 4 0 <SRKH6>
ccs::write_mem 0 0xfe0e8254 4 0 <SRKH7>
ccs::write_mem 0 0xfe0e8258 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem 0 0xfe0e00e4 4 0 0x00000001
```

10.3.1.16 Appendix LS1012 Secure Boot demo

Boot Source

Table 337. Memory map for LS1012 platform

Address QSPI[23]	Definition (Chain of Trust)	Size Reserved (KB)
40000000	RCW	128
40060000	Bootscript	128
40080000	ESBC U-Boot HEADER	128
400C0000	Bootscript Header	128
40100000	PPA Header	128
40480000	ESBC U-Boot	1024
40500000	PPA FIT Image	2048
40A00000[24]	Kernel FIT Image	54272
43200000	kernel Header	128

[23] QSPI by default works in 64bit Big Endian as XIP memory. So the data has to be 64 bit byte swapped before loading on QSPI.
[24] The Kernel Image and its CSF Header may be placed on any other memory as well. The same must be copied to DDR before validation and Booting.

Chain of Trust Boot Script Used as per Address Map

```
#Copy the Kernel Image from Flash to DDR
cp.b 0x40A00000 0xa0000000 0x2800000
#Validate the Kernel Image (The header has Image address as 0x81000000)
esbc_validate 0x43200000
#Boot the validated Kernel FIT Image.
setenv bootargs "console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm 0xa0000000
```

Refer to the ‘Product Execution’ section and blow the required fuses as mentioned in it before executing the following commands

Useful U-Boot and CCS Commands

```
protect off all;
setenv path <tftp_path>
sf probe 0:0
tftp 0x80000000 $path/rcw.bin; sf erase 0x0 20000; sf write 0x80000000 0x0 20000
tftp 0x80000000 $path/hdr_uboot.out; sf erase 0x80000 20000; sf write 0x80000000 0x80000 20000
tftp 0x80000000 $path/u-boot.bin; sf erase 0x100000 80000; sf write 0x80000000 0x100000 80000
tftp 0x80000000 $path/hdr_bs.out; sf erase 0xC0000 20000; sf write 0x80000000 0xC0000 20000
tftp 0x80000000 $path/bootscript; sf erase 0x60000 20000; sf write 0x80000000 0x60000 20000
tftp 0x80000000 $path/hdr_kernel.out; sf erase 0x3200000 20000; sf write 0x80000000 0x3200000
20000
tftp 0x80000000 $path/kernel.itb; sf erase 0xA00000 0x2800000; sf write 0x80000000 0xA00000
0x2800000
tftp 0x80000000 $path/hdr_ppa.out; sf erase 0x480000 20000; sf write 0x80000000 0x480000 20000
tftp 0x80000000 $path/ppa.itb; sf erase 0x500000 40000; sf write 0x80000000 0x500000 40000
```

```
# Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem <dap chain pos> 0x1ee0200 4 0 4
#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

10.3.1.17 Appendix LS1020 Secure Boot demo

Boot Source

Table 338. Memory map for LS1 platforms

Address NOR(vBank 0)	Address (NOR Alternate Bank)[25]	Definition (Chain of Trust)	Size Reserved(KB)
60000000	64000000	RCW	128
60020000	64020000	DTB	256
60060000	64060000	Bootscript	128
60080000	64080000	ESBC U-Boot HEADER	128
600A0000	640A0000	Bootscript Header	128
60100000	64100000	ESBC U-Boot	1024
60200000	64200000	uImage	8192
60A00000	64A00000	Rootfs	54272
63F00000	67F00000	uImage Header	128
63F20000	67F20000	DTB Header	128
63F40000	67F40000	rootfs Header	128

Chain Of Trust Boot Script Used as per Address Map

```
esbc_validate 0x63F20000
esbc_validate 0x63F00000
esbc_validate 0x63F40000
bootm 0x60200000 0x60A00000 0x60020000
```

Refer to the 'Product Execution' Section and blow the required fuses as mentioned in it before executing the following commands

Useful U-Boot and CCS Commands

```
protect off all;
setenv path <tftp_path>
tftp 80000000 $path/rcw.bin;erase 64000000 +$filesize;cp.b 80000000 64000000 $filesize;
tftp 80000000 $path/hdr_uboot.out;erase 64080000 +$filesize;cp.b 80000000 64080000 $filesize;
tftp 80000000 $path/u-boot.bin;erase 64100000 +$filesize;cp.b 80000000 64100000 $filesize;

tftp 80000000 $path/hdr_bs.out;erase 640A0000 +$filesize;cp.b 80000000 640A0000 $filesize;
tftp 80000000 $path/bootscript;erase 64060000 +$filesize;cp.b 80000000 64060000 $filesize;

tftp 80000000 $path/hdr_dtb.out;erase 67F20000 +$filesize;cp.b 80000000 67F20000 $filesize;
tftp 80000000 $path/uImage.dtb;erase 64020000 +$filesize;cp.b 80000000 64020000 $filesize;

tftp 80000000 $path/hdr_linux.out;erase 67F00000 +$filesize;cp.b 80000000 67F00000 $filesize;
tftp 80000000 $path/uImage.bin;erase 64200000 +$filesize;cp.b 80000000 64200000 $filesize;
```

[25] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

```
tftp 80000000 $path/hdr_rootfs.out;erase 67F40000 +$filesize;cp.b 80000000 67F40000 $filesize;
tftp 80000000 $path/rootfs;erase 64a00000 +$filesize;cp.b 80000000 64a00000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1020a dap sap2}
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem  <dap chain pos> 0x1ee0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

Boot Source

Table 339. Memory map for LS1 platforms

Address SD	Definition (Chain of Trust)	Size Reserved(Sector)
8	u-boot-with-spl-pbl-sec.bin (rcw_sec.bin, u-boot, hdr_uboot.out, bootscript, hdr_bs.out)	940
1f280	hdr_linux.out	100
1f380	hdr_dtb.out	100
1f480	hdr_rootfs.out	100
1f580	uImage.bin	2000
4880	uImage.dtb	200
4a80	rootfs	19000

Chain Of Trust Boot Script Used as per Address Map

```
mmc read 0x82000000 0x1F280 0x100
mmc read 0x82020000 0x1F380 0x100
mmc read 0x82040000 0x1F480 0x100
mmc read 0x83000000 0x1F580 0x2000
mmc read 0x83800000 0x4880 0x200
mmc read 0x8f000000 0x4A80 0x19000
```

```
echo Validate uImage.bin
esbc_validate 0x82000000
echo Validate uImage.dtb
esbc_validate 0x82020000
echo Validate rootfs
esbc_validate 0x82040000
bootm 0x83000000 0x8f000000 0x83800000
```

Refer to the ‘Product Execution’ Section and blow the required fuses as mentioned in it before executing the following commands

Useful U-Boot and CCS Commands

```
setenv path b57223/LS1021/ls1021_idc/sdboot
tftp 81000000 $path/rcw_pbi_sec.bin
mmc erase 8 0x940
mmc write 0x81000000 8 0x940
tftp 81000000 $path/hdr_linux.out
mmc erase 0x1f280 0x100
mmc write 0x81000000 0x1F280 0x100
tftp 81000000 $path/hdr_dtb.out
mmc erase 0x1f380 0x100
mmc write 0x81000000 0x1F380 0x100
tftp 81000000 $path/hdr_rootfs.out
mmc erase 0x1f480 0x100
mmc write 0x81000000 0x1F480 0x100
tftp 81000000 $path/uImage.bin
mmc erase 0x1F580 0x2000
mmc write 0x81000000 0x1F580 0x2000
tftp 81000000 $path/uImage.dtb
mmc erase 0x4880 0x200
mmc write 0x81000000 0x4880 0x200
tftp 81000000 $path/rootfs
mmc erase 0x4a80 0x19000
mmc write 0x81000000 0x4A80 0x19000
```

```
# Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1020a dap sap2}
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem  <dap chain pos> 0x1ee0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

10.3.1.18 Appendix LS1043 Secure Boot demo

Boot Source "NOR"

Table 340. Memory map for LS1043 platforms

Address NOR(vBank 0)	Address (NOR Alternate Bank)[26]	Definition (Chain of Trust)	Size Reserved (KB)
60000000	64000000	RCW	128
60060000	64060000	Bootscript	128
60080000	64080000	ESBC U-Boot HEADER	128
600A0000	640A0000	Bootscript Header	128
600C0000	640C0000	PPA Header	128
60100000	64100000	ESBC U-Boot	1024
60500000	64500000	PPA FIT Image	2048
60A00000*	64A00000*	Kernel FIT Image	54272
63F40000	64F40000	kernel Header	128

NOTE

For LS1043 Bootscript, kernel image must be copied to DDR address 0x81000000 before issuing esbc_validate command.

Chain of Trust Boot Script Used as per Address Map

```
# Copy the Kernel Image from Flash to DDR
cp.b 0x60A00000 0x81000000 0x3500000

# Validate the Kernel Image (The header has Image address as 0x81000000)
esbc_validate 0x63F40000

# Boot the validated Kernel FIT Image.
setenv bootargs "console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";

bootm $img_addr
```

Refer to the ‘Product Execution’ Section and blow the required fuses as mentioned in it before executing the following commands

Useful U-Boot and CCS Commands

```
protect off all;
setenv path <tftp_path>
```

[26] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

```
tftp 80000000 $path/rcw.bin;erase 64000000 +$filesize;cp.b 80000000 64000000 $filesize;
tftp 80000000 $path/hdr_uboot.out;erase 64080000 +$filesize;cp.b 80000000 64080000 $filesize;
tftp 80000000 $path/u-boot.bin;erase 64100000 +$filesize;cp.b 80000000 64100000 $filesize;

tftp 80000000 $path/hdr_bs.out;erase 640A0000 +$filesize;cp.b 80000000 640A0000 $filesize;
tftp 80000000 $path/bootscript;erase 64060000 +$filesize;cp.b 80000000 64060000 $filesize;

tftp 80000000 $path/hdr_kernel.out;erase 67F40000 +$filesize;cp.b 80000000 67F40000 $filesize;
tftp 80000000 $path/kernel.itb;erase 64a00000 +$filesize;cp.b 80000000 64a00000 $filesize;

tftp 80000000 $path/hdr_ppa.out;erase 640C0000 +$filesize;cp.b 80000000 640C0000 $filesize;
tftp 80000000 $path/ppa.itb;erase 64500000 +$filesize;cp.b 80000000 64500000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem  <dap chain pos> 0x1ee0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

Boot Source "SD"

Table 341. Memory map for LS1043 platforms

Address SD	Definition (Chain of Trust)	Size Reserved (KB)
8	u-boot-spl-pbl-sec.bin (rcw_sec.bin, u-boot, hdr_uboot.out, bootscript, hdr_bs.out)	940
1f480	hdr_kernel.out	100
4a80	kernel.out	19000

Chain of Trust Boot Script Used as per Address Map

```
mmc rescan
mmc read 0x81000000 0x1F480 0x100
mmc read 0xa0000000 0x4A80 0x19000
echo Validate kernel.itb
esbc_validate 0x81000000
setenv bootargs "console=ttyS0,115200 root=/dev/ram0
```

```
earlycon=uart8250,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm a0000000
```

Useful U-Boot and CCS Commands

```
setenv path <path>
tftp 81000000 $path/u-boot-with-spl-pbl-sec.bin
mmc erase 8 0x940
mmc write 0x81000000 8 0x940
tftp 81000000 $path/hdr_kernel.out
mmc erase 0x1f480 0x100
mmc write 0x81000000 0x1F480 0x100
tftp 81000000 $path/kernel.itb
mmc erase 0x4a80 0x19000
mmc write 81000000 0x4a80 0x19000
```

```
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap chain pos>  0x1e90014 4 0 4
ccs::display_mem <dap chain pos>  0x1ee0200 4 0 4
#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos>  0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos>  0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos>  0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos>  0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos>  0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos>  0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos>  0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos>  0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos>  0x1ee00e4 4 0 0x00000001
```

Boot Source "NAND"

Table 342. Memory map for LS1043 platforms

Address NAND	Definition (Chain of Trust)	Size Reserved (KB)
0x0	u-boot-spl-pbl-sec.bin (rcw_sec.bin, u-boot, hdr_uboot.out, bootscript, hdr_bs.out)	900000
0x900000	hdr_kernel.out	2000
0x2000000	kernel.out	2000000

Chain of Trust Boot Script Used as per Address Map

```
mmc rescan
nand read 0x81000000 0x900000 0x2000
nand read 0xa0000000 0x2000000 0x2000000
echo Validate kernel.itb
esbc_validate 0x81000000
```

```
setenv bootargs "console=ttyS0,115200 root=/dev/ram0
earlycon=uart8250,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm a0000000
```

Useful U-Boot and CCS Commands

```
setenv path <path>
tftp 81000000 $path/u-boot-with-spl-pbl-sec.bin
nand erase 0 0x900000
nand write 81000000 0 900000
tftp 81000000 $path/hdr_kernel.out
nand erase 900000 0x2000
nand write 0x81000000 0x900000 0x2000
tftp 81000000 $path/kernel.itb
nand erase 2000000 0x2000000
nand write 0x81000000 0x2000000 0x2000000
```

```
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap chain pos>  0x1e90014 4 0 4
ccs::display_mem <dap chain pos>  0x1ee0200 4 0 4
#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos>  0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos>  0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos>  0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos>  0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos>  0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos>  0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos>  0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos>  0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos>  0x1ee00e4 4 0 0x00000001
```

10.3.1.19 Appendix LS1046 Secure Boot demo

The addresses below are effective addresses as mapped by u-boot

Boot Source: NOR

Table 343. Memory Map for LS1046 Platforms

Address NOR(vBank 0)	Address (NOR Alternate Bank)	Definition (Chain of Trust)	Size Reserved (KB)
60000000	64000000	RCW	128
60060000	64060000	Bootscript	128
60080000	64080000	ESBC U-Boot HEADER	128
600A0000	640A0000	Bootscript Header	128

Table continues on the next page...

Table 343. Memory Map for LS1046 Platforms (continued)

600C0000	640C0000	PPA Header	128
60100000	64100000	ESBC U-Boot	1024
60500000	64500000	PPA FIT Image	2048
60A00000*	64A00000*	Kernel FIT Image	54272
63F40000	64F40000	kernel Header	128

NOTE

For LS1046 Bootscript, kernel image must be copied to DDR address 0x81000000 before issuing esbc_validate command.

Chain Of Trust Boot Script Used as per Address Map

```
#Copy the Kernel Image from Flash to DDRcp.b
0x60A00000 0x81000000 0x3500000
#Validate the Kernel Image (The header has Image address as 0x81000000)
esbc_validate 0x63F40000
#Boot the validated Kernel FIT Image.
setenv bootargs "console=ttyS0,115200 root=/dev/ram0
earlycon=uart8250,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm
$img_addr
```

Useful U-Boot and CCS Commands

```
protect off all;
setenv path <tftp_path>
tftp 80000000 $path/rcw.bin;erase 64000000 +$filesize;cp.b 80000000 64000000 $filesize;
tftp 80000000 $path/hdr_uboot.out;erase 64080000 +$filesize;cp.b 80000000 64080000 $filesize;
tftp 80000000 $path/u-boot.bin;erase 64100000 +$filesize;cp.b 80000000 64100000 $filesize;
tftp 80000000 $path/hdr_bs.out;erase 640A0000 +$filesize;cp.b 80000000 640A0000 $filesize;
tftp 80000000 $path/bootscript;erase 64060000 +$filesize;cp.b 80000000 64060000 $filesize;
tftp 80000000 $path/hdr_kernel.out;erase 67F40000 +$filesize;cp.b 80000000 67F40000 $filesize;
tftp 80000000 $path/kernel.itb;erase 64a00000 +$filesize;cp.b 80000000 64a00000 $filesize;
tftp 80000000 $path/hdr_ppa.out;erase 640C0000 +$filesize;cp.b 80000000 640C0000 $filesize;
tftp 80000000 $path/ppa.itb;erase 64500000 +$filesize;cp.b 80000000 64500000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem <dap chain pos> 0x1ee0200 4 0 4
#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
```

```
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

Boot Source: QSPI

Table 344. Memory Map for LS1046 Platform

Address QSPI[27]	Definition (Chain of Trust)	Size Reserved (KB)
40000000	RCW	128
40800000	Bootscript	128
40700000	ESBC U-Boot HEADER	128
40780000	Bootscript Header	128
40740000	PPA Header	128
40100000	ESBC U-Boot	1024
40500000	PPA FIT Image	2048
41000000[28]	Kernel FIT Image	54272
407C0000	kernel Header	128

Chain Of Trust Boot Script Used as per Address Map

```
#Copy the Kernel Image from Flash to DDR
cp.b 0x41000000 0x81000000 0x2800000
#Validate the Kernel Image (The header has Image address as 0x81000000)
esbc_validate 0x407c0000
#Boot the validated Kernel FIT Image.
setenv bootargs "console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm $img_addr
```

Chain of Trust Boot Script Used as per Address Map (For Encapsulation)

```
sf probe 0:0
blob enc 0x41000000 0x80000000 0x2000000 0x40000000
sf erase 0x1000000 +2020000;
sf write  0x80000000 0x1000000 2020000;
```

Chain of Trust Boot Script Used as per Address Map (For Decapsulation)

```
sf probe 0:0
sf read 0x84000000 0x1000000 2020000
blob dec 0x84000000 0x81000000 0x2000000 0x40000000
esbc_validate 0x407c0000
Boot the validated Kernel FIT Image.
```

[27] QSPI by default works in 64bit Big Endian as XIP memory. So the data has to be 64 bit byte swapped before loading on QSPI.

[28] The Kernel Image and its CSF Header may be placed on any other memory as well. The same must be copied to DDR before validation and Booting.

```
setenv bootargs "console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm $img_addr
```

Useful U-Boot and CCS Commands

```
protect off all;
setenv path <tftp_path>
sf probe 0:1
tftp 0x80000000 $path/rcw_swap.bin; sf erase 0x0 +$filesize; sf write 0x80000000 0x0 $filesize;
tftp 0x80000000 $path/hdr_uboot_swap.out; sf erase 0x700000 +$filesize; sf write 0x80000000
0x700000 $filesize;
tftp 0x80000000 $path/u-boot_swap.bin; sf erase 0x100000 +$filesize; sf write 0x80000000
0x100000 $filesize;
tftp 0x80000000 $path/hdr_ppa_swap.out; sf erase 0x740000 +$filesize; sf write 0x80000000
0x740000 $filesize;
tftp 0x80000000 $path/ppa_swap.itb; sf erase 0x500000 +$filesize; sf write 0x80000000
0x500000 $filesize;
tftp 0x80000000 $path/hdr_bs_swap.out; sf erase 0x780000 +$filesize; sf write 0x80000000
0x780000 $filesize;
tftp 0x80000000 $path/bootscript_swap; sf erase 0x800000 +$filesize; sf write 0x80000000
0x800000 $filesize;
tftp 0x80000000 $path/hdr_kernel_swap.out; sf erase 0x7c0000 +$filesize; sf write 0x80000000
0x7c0000 $filesize;
tftp 0x80000000 $path/kernel_swap.itb; sf erase 0x1000000 +$filesize; sf write 0x80000000
0x1000000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_server 0 10
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem <dap chain pos> 0x1ee0200 4 0 4
#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

Boot Source: SD

Table 345. Memory Map for LS1046 Platform

Address SD	**Definition (Chain of Trust)**	**Size Reserved (Sector)**

Table continues on the next page...

Table 345. Memory Map for LS1046 Platform (continued)

8	u-boot-spl-pbl-sec.bin (rcw_sec.bin, u-boot, hdr_uboot.out, bootscript, hdr_bs.out)	8
1f480	hdr_kernel.out	100
4a80	kernel.itb	19000

Chain of Trust Boot Script Used as per Address Map

```
mmc rescan
mmc read 0x81000000 0x1F480 0x100
mmc read 0xa0000000 0x4A80 0x19000
echo Validate kernel.itbesbc_validate 0x81000000
setenv bootargs "console=ttyS0,115200 root=/dev/ram0
earlycon=uart8250,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm a0000000
```

Useful U-Boot and CCS Commands

```
setenv path <path>
tftp 81000000 $path/u-boot-with-spl-pbl-sec.bin
mmc erase 8 0x940
mmc write 0x81000000 8 0x940
tftp 81000000 $path/hdr_kernel.out
mmc erase 0x1f480 0x100
mmc write 0x81000000 0x1F480 0x100
tftp 81000000 $path/kernel.itb
mmc erase 0x4a80 0x19000
mmc write 81000000 0x4a80 0x19000
```

```
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap chain pos>  0x1e90014 4 0 4
ccs::display_mem <dap chain pos>  0x1ee0200 4 0 4
#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos>  0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos>  0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos>  0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos>  0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos>  0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos>  0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos>  0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos>  0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos>  0x1ee00e4 4 0 0x00000001
```

10.3.1.20 Appendix P3/P5/T1 NAND Secure Boot

Table 346. Memory Map for P3/P5 NAND SECURE BOOT

Description (Chain of Trust)	Description (Chain of Trust with Confidentiality)	Address on NAND	Address on DDR (Image copied to from NAND)	Size
PBL.bin (RCW, PBI, U-Boot, U-boot Header)	PBL.bin (RCW, PBI, U-Boot, U-boot Header)	0x00000000	--	0xD0000
Boot Script Header	Boot Script Header	0x00800000	0x00010000	0x2000
Boot Script	Boot Script	0x00802000	0x00012000	0x2000
uImage Header		0x00804000	0x00014000	0x2000
dtb Header		0x00806000	0x00016000	0x2000
rootfs Header		0x00808000	0x00018000	0x2000
uImage	uImage	0x06500000	0x01000000	0x410000
dtb	dtb	0x06b00000	0x00c00000	0x9000
rootfs	rootfs	0x02000000	0x02000000	0x2000000
	uImage (Encapsulated)	0x06500000	0x10000000	0x410000
	dtb (Encapsulated)	0x06b00000	0x10000000	0x9000
	rootfs (Encapsulated)	0x02000000	0x10000000	0x2000000

Chain Of Trust Boot Script Used for P3/P5 as per Address Map

```
#UImage and Header
nand read 0x01000000 0x06500000 0x410000
nand read 0x00014000 0x00804000 0x2000
#Rootfs and HEader
nand read 0x02000000 0x02000000 0x2000000
nand read 0x00016000 0x00806000 0x2000
#DTB and Header
nand read 0x00c00000 0x06b00000 0x9000
nand read 0x00018000 0x00808000 0x2000
esbc_validate 0x00014000
esbc_validate 0x00016000
esbc_validate 0x00018000
bootm 0x01000000 0x02000000 0x00c00000
```

Table 347. Memory Map for T1042 NAND SECURE BOOT

Description	Address on NAND	Address on DDR (Image copied to from NAND)	Size
PBL.bin (RCW, PBI Commands, u-boot-spl, hdr_uboot_spl)	0x00000000	--	0x200000

Table continues on the next page...

Table 347. Memory Map for T1042 NAND SECURE BOOT (continued)

u-boot.bin	0x00040000	0x30000000	0xc0000
hdr_uboot.out	0x00100000	0x30000000	0x4000
Boot Script Header	0x00800000	0x00010000	0x2000
Boot Script	0x00802000	0x00012000	0x2000
uImage Header	0x00804000	0x00014000	0x2000
dtb Header	0x00806000	0x00016000	0x2000
rootfs Header	0x00808000	0x00018000	0x2000
uImage	0x06500000	0x01000000	0x450000
dtb	0x06b00000	0x00c00000	0x9000
rootfs	0x02000000	0x02000000	0x2000000

Chain Of Trust Boot Script Used for T104x as per Address Map

```
#Read Images from NAND to DDR
#UImage and Header
nand read 0x01000000 0x06500000 0x450000
nand read 0x00014000 0x00804000 0x2000
#Rootfs and Header
nand read 0x02000000 0x02000000 0x2000000
nand read 0x00016000 0x00806000 0x2000
#DTB and Header
nand read 0x00c00000 0x06b00000 0x9000
nand read 0x00018000 0x00808000 0x2000
#Validate and Boot
esbc_validate 0x00014000
esbc_validate 0x00016000
esbc_validate 0x00018000
bootm 0x01000000 0x02000000 0x00c00000
```

10.3.2 Software-PreBoot Loader (PBL) Based Platforms

10.3.2.1 Introduction

This document is intended for end-users to demonstrate the image validation process. The image validation can be split into stages, where each stage performs a specific function and validates the subsequent stage before passing control to that stage. In the example, the ESBC is NXP provided reference code referred to as ESBC uboot.

Chain of Trust ESBC uboot performs minimal SoC configuration before validating the Next Executable using the same CSF header format as the ISBC used to validate ESBC Uboot. The CSF Header and signature are added to the Next Executable using the NXP Code Signing Tool.

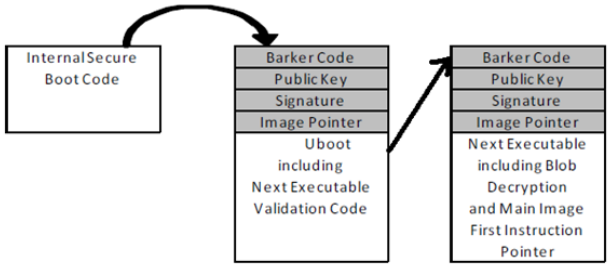


Figure 322. Chain of Trust

Chain of Trust with confidentiality

The validated ESBC uboot image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Cryptographic blob mechanism is used to establish Chain of trust with confidentiality.

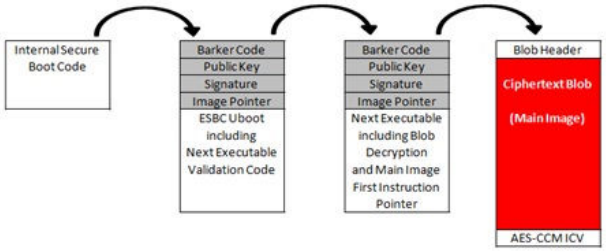


Figure 323. Chain of Trust with confidentiality

This document provides more details on the secure boot flow, ISBC, ESBC and NXP Code signing tool.

10.3.2.2 Image Signing

10.3.2.2.1 Overview

This document describes how to use secure boot capability in non-pbl platforms. The QorIQ platforms are cost-effective, low-power, highly integrated host processor that addresses the requirements of several routing, gateways, storage, consumer, and industrial applications. It does not incorporate the PreBoot Loader hardware.

Non-PBL platform supports two booting modes namely trusted or secure boot and legacy/non-secure boot.

Trusted mode — Boot images are validated against a digital signature before executing. The trusted mode gives a benefit that only OEM's signed boot images can be executed on these SOCs.

Legacy boot/non-secure boot — Booting happens from various boot targets as supported by SOC. SD/SDHC/MMC and SPI boot methods are supported using Internal Boot ROM.

When the device is first powered on, reset control logic blocks all device activity (including scan and debug activity) until fuse values can be accurately sensed. The most important fuse value at this stage of operation is the intent to secure (ITS) bit. If the user sets ITS, they intend for the system to operate in a secure and trusted manner. The setting of ITS determines the default settings of a range of configuration registers within the device, essentially locking down interfaces, memory permissions, and MMU configurations until trusted software is executing.

If the user sets the ITS bit, it is intended that the system operates in a secure and trusted manner. To ensure this, after the ITS bit is set, the system jumps to an internal secure ROM for booting, regardless of the value provided in the boot location POR. When execution begins from the IBR, it checks the cfg_rom_loc value that indicates the source of external secure boot code location. The value provided in the cfg_rom_loc directs the IBR to configure the respective peripheral.

If the user doesn't want to fuse the ITS bit, he can set CFG_SB_DIS to 0 to pass the system control to IBR.

The task of the IBR is to determine if subsequent code (referred to as the External Secure Boot Code, or ESBC) is valid before allowing that code to execute. To perform this validation, the IBR needs the information described in the following subsections.

10.3.2.2.2 ESBC with CF Header and CSF Header

ESBC is the generic name for the code that the ISBC validates. A few ESBC scenarios are described in later sections.

The I2C boot sequencer provides a way to configure the system using a set of address data tuples stored in I2C EEPROM. The boot sequencer works only in the legacy mode. In the secure mode the boot sequencer is disabled. However the same functionality is achieved by adding a similar data structure called a CF header or a configuration header. This header is on the lines of the SD/MMC and eSPI data structure containing the control words and configuration words. IBR running on the core first authenticates the CF Header and then executes the configuration words.

Figures below provide an example of an ESBC with CF header and CSF (Command Sequence File) Header prepended to it. The CF Header contains legacy mode fields, configuration words to initialize destination interfaces like DDR, ESBC header pointer. The CSF Header includes lengths and offset which allow the IBR to locate the operands used in ESBC image validation, as well as describe the size and location of the ESBC image itself.

NOTE

CSF Header and ESBC Header may be used synonymously in this and other NXP Trust Architecture documentation.

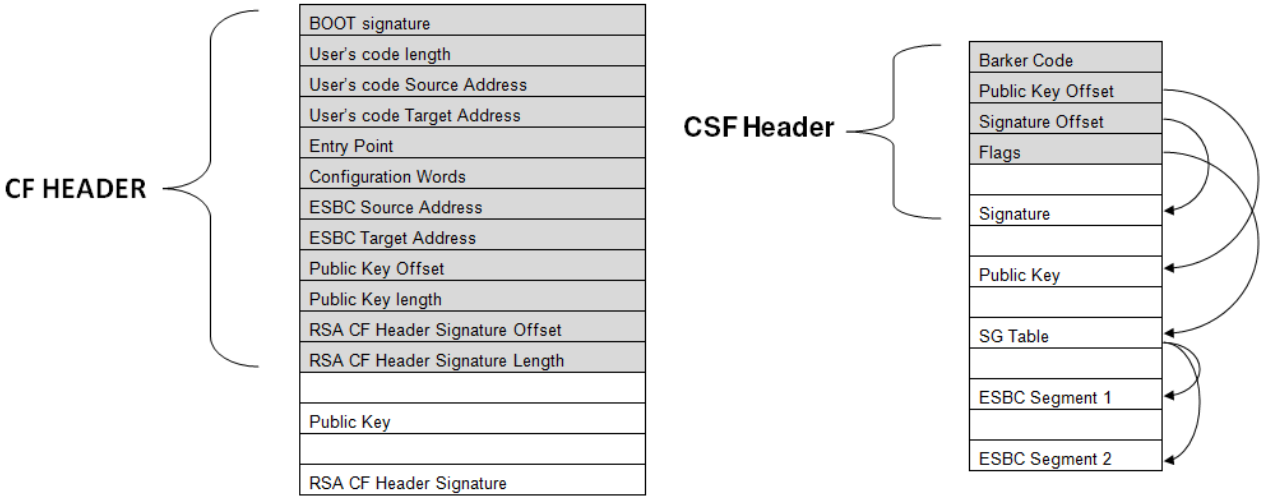


Figure 324. ESBC with CF and CSF Header for P1010/9131/9132

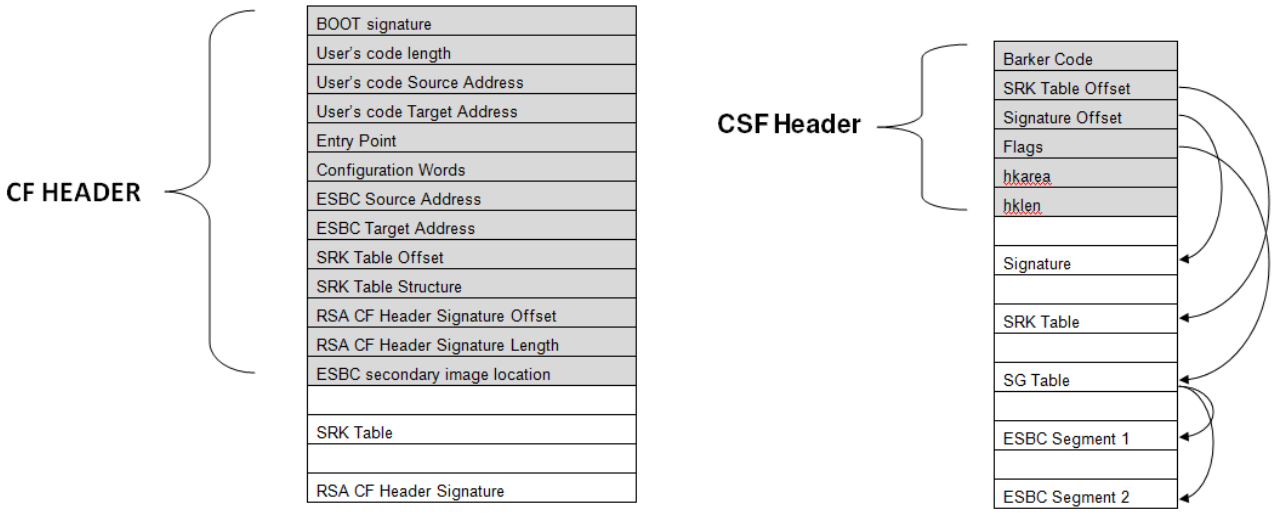


Figure 325. ESBC with CF and CSF Header for C290

10.3.2.3 Image validation

Although the CF Header and CSF Header provide most of the information needed for the IBR to perform image validation, the IBR can't begin until it finds the CF Header. The ISBC gets the CF Header pointer from 0th location of the 0th sector on the target interface. The fixed value Boot Signature (0x424f4f54) at the 0x40 offset from the start of the CF Header allows the ISBC to know it has found a CF Header. If the ISBC doesn't find the Boot Signature at the CF Header 0x40 location, the ISBC reports an error and shuts down. See Validation Failures from ISBC on page 2024 for more information on validation failures.

10.3.2.3.1 Validation failures

Validation failure can occurs at 2 stages:

1. ISBC fails to validate CF header or ESBC header
2. ESBC fails to validate the next level executable

The section below describes the details of these validation failures.

10.3.2.3.1.1 Validation Failures from ISBC

There are many reasons the IBR could fail to validate the ESBC. Technicians with debug access can check the GUTS SBDCR (CCSRBAR + 0xe0f90) register to obtain an error code.

For a list of error codes refer ISBC Validation Error Codes on page 2057

10.3.2.3.1.2 Validation Failures from ESBC

There are many reasons ESBC could fail to validate Client images or boot script. The error status message along with the code is printed on the u-boot console.

Table 348. ESBC Validation Failures

Value	Code	Definition
0x4	ERROR_ESBC_CLIENT_HEADER_BARKER	Wrong barker code in header

Table continues on the next page...

Table 348. ESBC Validation Failures (continued)

Value	Code	Definition
0x8	ERROR_ESBC_CLIENT_HEADER_KEY_LEN	Wrong public key length in header
0x10	ERROR_ESBC_CLIENT_HEADER_SIG_LEN	Wrong signature length in header
0x20	ERROR_ESBC_CLIENT_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key length not twice of signature length
0x40	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_1	Public key Modulus most significant bit not set
0x80	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_2	Public key Modulus in header not odd
0x100	ERROR_ESBC_CLIENT_HEADER_SIG_KEY_MOD	Signature not less than modulus
0x400	ERROR_ESBC_CLIENT_HASH_COMPARE_KEY	Public key hash comparison failed
0x800	ERROR_ESBC_CLIENT_HASH_COMPARE_EM	RSA verification failed
0x10000	ERROR_ESBC_CLIENT_HEADER_SG	No SG support
0x20000	ERROR_ESBC_WRONG_CMD	Failure in command/Unknown command/Wrong arguments of boot script.
0x40000	ERROR_ESBC_MISSING_BOOTM	Bootm command missing from boot script.

10.3.2.3.2 Boot script

Bootscript is a U-Boot script image which contains u-boot commands. ESBC would validate this boot script before executing commands in it.

NOTE

1. Boot script can have any commands which u-boot supports. No checking on the allowed commands in boot script. Since it is validated image, assumption is that commands in boot script would be correct.
2. If some basic scripting error done in boot script like unknown command, missing arguments, the required usage of that command and core is put in infinite loop.
3. After execution of commands in boot script, if control reaches back in u-boot, error message would be printed on u-boot console and core would be put in spin loop by command esbc_halt.
4. Scatter gather images not supported with validate command.
5. If ITS fuse is blown, any error in verification of the image would result in system reset. The error would be printed on console before system goes for a reset.

10.3.2.3.2.1 Where to place the boot script?

NXP's ESBC u-boot expects the boot script to be loaded in flash as specified in Address map used for the demo on page 2064. ESBC u-boot code assumes that the public/private key pair used to sign the boot script is same as that was used while signing the u-boot image. If user used different key pair to sign the image, hash of the N and E component of the key pair should be defined in macro:

CONFIG_BOOTSCRIPT_KEY_HASH.

Note - The hash defined should be hex value, 256 bits long.

Both the above macros can be defined or changed in the configuration file secure_boot.h at the following location in u-boot code:

```
u-boot/arch/powerpc//include/asm/fsl_secure_boot.h
```

Two new commands called esbc_validate and esbc_halt have been added in NXP ESBC u-boot.

10.3.2.3.3 What is ESBC?

Depending on the requirement, ESBC can be a monolithic image - including uboot, device trees, boot firmware, drivers along with the OS and applications or can be mini-uboot.

NXP provided ESBC consists of standard u-boot which has been signed using a private key. U-boot reserves a small space for storing environment variables. This space is typically one sector above or below the u-boot and is stored on persistent storage devices like NOR flash if macro CONFIG_ENV_IS_IN_FLASH is used. In case of secure boot, macro CONFIG_ENV_IS_NOWHERE is used and so, environment is compiled in uboot image and is called default environment. This default environment can't be stored on flash devices. User won't be able to edit this environment also as he can't reach to uboot prompt in case of secure boot. There is default boot command for secure boot in this default environment which executes on autoboot.

ESBC validates a file called boot script and on successful validation execute the commands in the boot script.

Users are free to use NXP ESBC as it is provided or to use it as reference to modify their own secure boot system.

10.3.2.3.3.1 Chain of Trust

Boot script contains information about the next level of images, e.g. Linux, HV, etc. ESBC validates these images as per their public keys and then executes bootm command to pass-on the control to next image.

Users are free to use NXP ESBC as it is provided or to use it as reference to modify their own secure boot system.

Figure below shows the Chain of trust established for Validation with this ESBC u-boot.

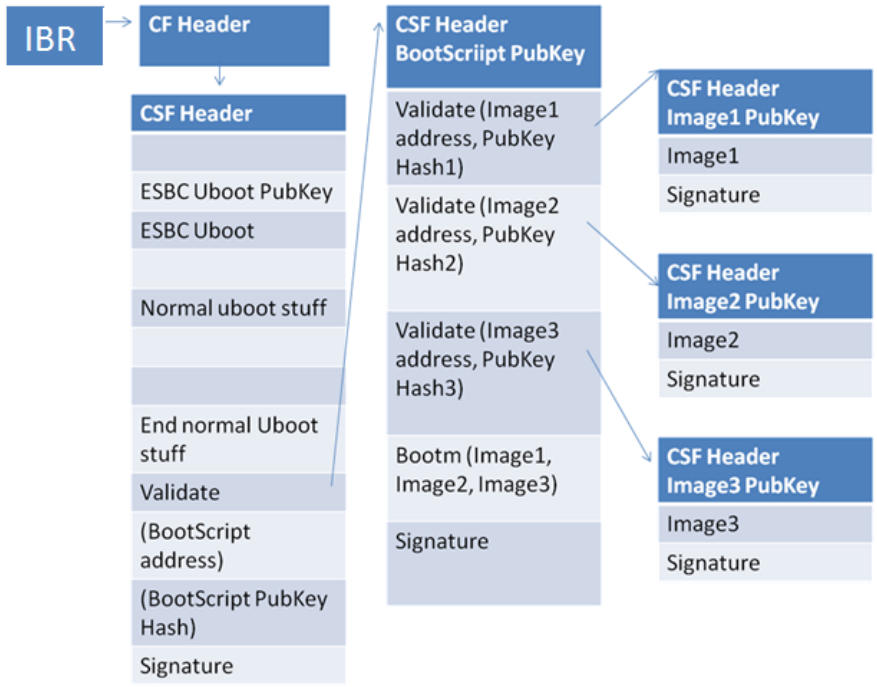


Figure 326. Secure boot flow (Chain of Trust)

10.3.2.3.3.1.1 Sample Boot Script

A sample boot script would look like:

```
...
esbc_validate <Img1 header addr> <pub_key hash>
esbc_validate <Img2 header addr> <pub_key hash>
esbc_validate <Img3 header addr> <pub_key hash>
...
bootm <img1 addr> <img2 addr> <img3 addr>
```

10.3.2.3.3.1.1.1 esbc_validate command

esbc_validate img_hdr [pub_key_hash]

Input arguments:

`img_hdr` - Location of CSF Header of the image to be validated

`pub_key_hash` - hash of the public key used to verify the image. This is optional parameter. If not provided, code makes the assumption that the key pair used to sign the image is same as that used with ISBC. So the hash of the key in the header is checked against the hash available in SRK fuse for verification.

Description:

The command would do the following:

- Perform CSF header validation on the address passed in the image header. During parsing of the header, image address in stored in an environment variable which is later used in source command in default secure boot command.
- Signature checks on the image

10.3.2.3.3.1.1.2 esbc_halt command

esbc_halt (no arguments)

Description:

The command would do the following:

This command puts core in spin loop.

After successful validation of images, bootm command in bootscript should execute and control should never reach back to uboot. If somehow, control reaches back to uboot (eg. bootm not present in bootscript), core should just spin.

10.3.2.3.3.2 Chain of Trust with Confidentiality

To establish chain of trust with confidentiality, cryptgraphic blob mechanism can be used. In this chain of trust, validated image is allowed to use the One Time Programmable Master Key to decrypt system secrets.

Two bootscripts are to be used. First encap bootscripts is used which creates a blob of the LINUX images and saves them. After this the system is booted after replacing the encap bootscript with decap bootscript which decapsulates the blobs and boot the LINUX with the images.

Figures below show the Chain of trust with confidentiality (Encapsulation and Decapsulation).

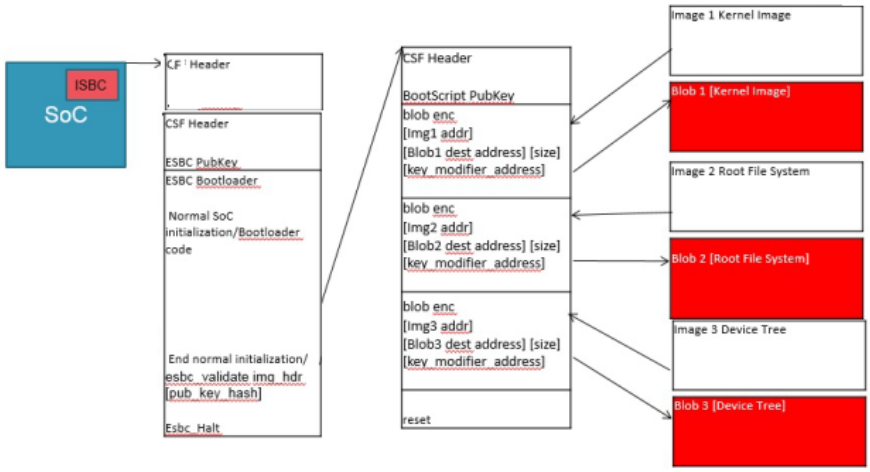


Figure 327. Chain of Trust with Confidentiality (Encapsulation)

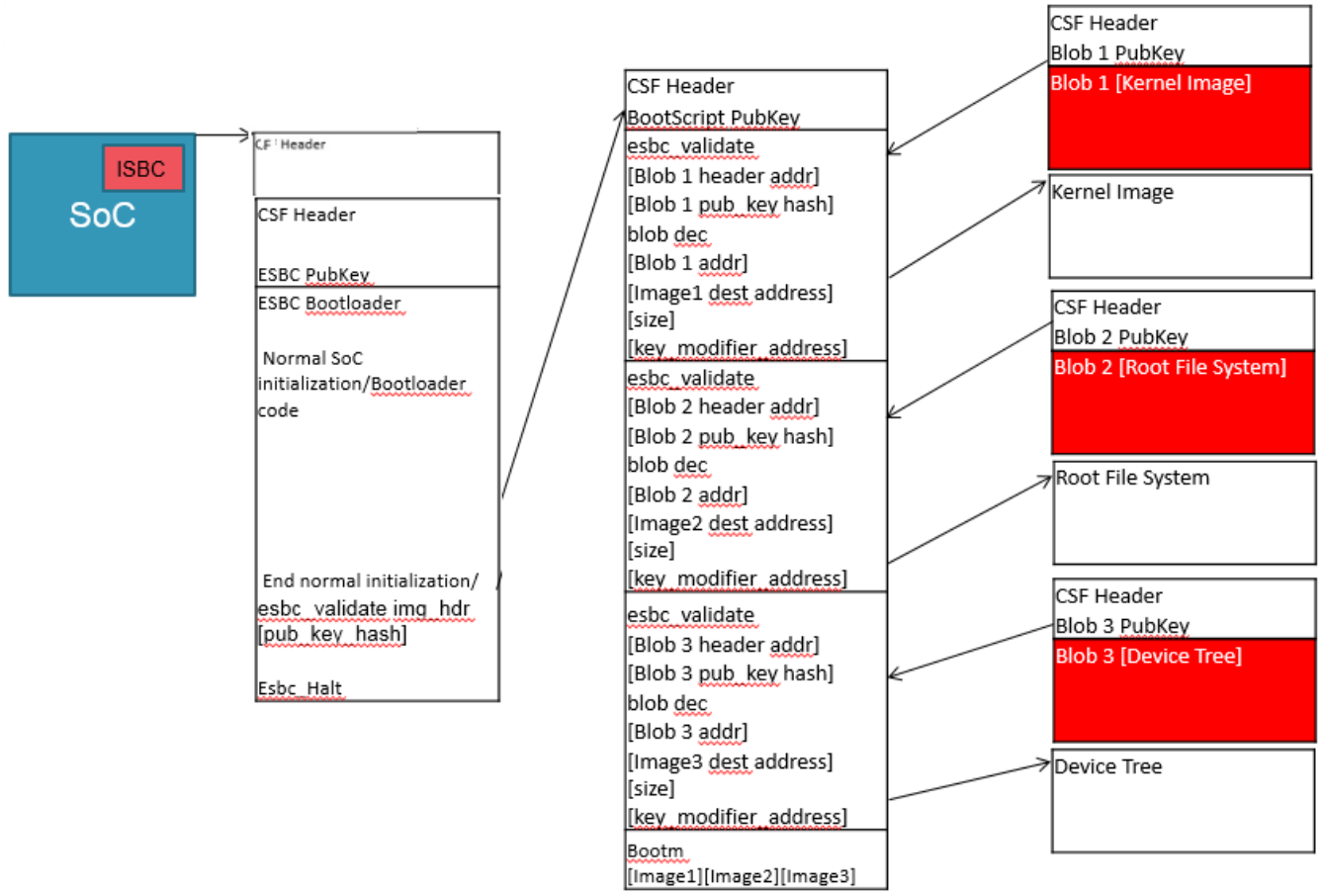


Figure 328. Chain of Trust with Confidentiality (Decapsulation)

10.3.2.3.3.2.1 Sample Encap Boot Script

A sample encap boot script would look like:

```
...
blob enc <Img1 addr> <Img1 dest addr> <Img1 size> <16byte key modifier address>
erase <encap Img1 addr> +<Img1 size + 0x30>
cp.b <Img1 dest addr> <encap Img1 addr> <Img1 size + 0x30>

blob enc <Img2 addr> <Img2 dest addr> <Img2 size> <16byte key modifier address>
erase <encap Img2 addr> +<Img2 size + 0x30>
cp.b <Img2 dest addr> <encap Img2 addr> <Img2 size + 0x30>

blob enc <Img3 addr> <Img3 dest addr> <Img3 size> <16byte key modifier address>
erase <encap Img3 addr> +<Img3 size + 0x30>
cp.b <Img3 dest addr> <encap Img3 addr> <Img3 size + 0x30>

...
```

10.3.2.3.3.2.1.1 blob enc command

blob enc <src location> <dst location> <length> <16 byte key modifier address>

Input arguments:

`src location` - Address of the image to be encapsulated

`dst location` - Address where the blob will be created

`length` - Size of the image to be encapsulated

`key modifier address` - Address of any 16 bytes long random number

Description:

The command would do the following:

- Create a cryptographic blob of the image placed at src location and place the blob at dst location.

10.3.2.3.3.2.2 Sample Decap Boot Script

A sample decap boot script would look like:

```
...
blob dec <Img1 blob addr> <Img1 dest addr> <Img1 size + 0x30> <16byte key modifier address>
blob dec <Img2 blob addr> <Img2 dest addr> <Img2 size + 0x30> <16byte key modifier address>
blob dec <Img3 blob addr> <Img3 dest addr> <Img3 size + 0x30> <16byte key modifier address>
...
bootm <Img1 dest addr> <Img2 dest addr> <Img3 dest addr>
```

10.3.2.3.3.2.2.1 blob dec command

blob dec <src location> <dst location> <length> <16 byte key modifier address>

Input arguments:

`src location` - Address of the image blob to be decapsulated

`dst location` - Address where the decapsulated image will be placed

`length` - Expected Size of the image after decapsulation.

`key modifier` - Address of any 16 bytes long random number(Same as that used for Encapsulation)

Description:

The command would do the following:

- Decapsulate the blob placed at src location and place the decapsulated data of expected size at dst location.

10.3.2.4 Product execution

This section presents the steps needed to be followed in order to properly run the software product according to its intended use and functionalities.

10.3.2.4.1 Getting started

In the demo we map the flash to the following addresses for the ISBC code to validate ESBC in NOR Flash using configuration words in CF Header.

- P1010 : 0xce000000
- BSC9131 : 0xce000000
- BSC9132 : 0xc8000000
- C290: 0xcc000000

Once the control reaches the ESBC code the earlier mapping of flash is removed and flash is mapped to address based on the platform as specified in .

NOTE

- ISBC expects the code to be validated i.e. ESBC code to be within **0 – 3.5G** address map.
- It is recommended not to create any LAW using CF words in CF Header at LAW 0 before U-boot. As ISBC creates a LAW 0 entry for nor and nand which will be removed in U-boot startup code.

The example below demonstrates the secure-boot flow with all the images loaded in NOR Flash.

10.3.2.4.1.1 SDK Images required for the demo

Given below are the images required for the demo which are built with Yocto as part of the SDK:

1. ESBC (U-Boot)
2. uImage (Linux Image)
3. rootfs Image
4. Device tree

Please refer to User Manual QorIQ DPAA SDK for detailed description on how to run Yocto Build. Once the build process finishes, all the binaries would be present at the following location:

build_<platform>_release/tmp/deploy/images

The images will be created with the following names:

u-boot-<platform>.bin	U-Boot binary image for Secure Boot
uImage-<platform>.bin	kernel image that can be loaded with U-Boot
fsl-image-core-<platform>.ext2.gz.u-boot	ramdisk filesystem image that can be loaded with U-Boot
uImage-<platform>.dtb	device tree binary(dtb) for kernel bootup

10.3.2.4.2 Chain of Trust

This section presents the steps needed to be followed in order to execute Chain of Trust.

Steps in the demo would be:

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to validate next level images, i.e rootfs, linux uImage and device tree.
5. Once all the images are validated, bootm command in boot script would be executed which would pass control to linux.

10.3.2.4.2.1 Other images required for the demo

Apart from SDK images described above, the following images are also required:

1. CF header of the ESBC u-boot image
2. CSF Header of the ESBC u-boot image
3. CSF Header of the uImage
4. CSF Header of the rootfs image
5. CSF Header of the device tree
6. Boot Script
7. CSF Header of the boot script

The following section describes how to create the CF header, CSF headers and boot script.

10.3.2.4.2.2 Boot Script and Signing the images

User can sign all the images with same public/private key pair or can use different key pairs to sign the images. Section below describes both the processes.

CST tool used for signing the images is provided as a package with yocto and is built for host. It can be run from your host machine.

Install path for CST binaries in yocto:

```
tmp/sysroots/x86_64-linux/usr/bin/cst/
```

CST uses openssl libraries, version 0.9.8.

In the Yocto environment, the user needs to use below commands to rebuild cst:

1. bitbake cst-native –c cleanall
2. bitbake cst-native
3. Modify the CST source code if needed.
4. bitbake cst-native –c cleanall
5. bitbake cst-native –c patch

Note: after step5, CST binary will be put to build_<platform>_release/tmp/sysroots/x86_64-linux/usr/bin/cst/ directory.

10.3.2.4.2.2.1 Signing the images using same key pair

CSF header needs to be generated for all the images. More details on the commands provided by CST can be found in Section .

1. Generate the key pair to be used for signing the image

   ```
   ./gen_keys 1024
   ```

   Key pair - public key file - srk.pub and private key in srk.priv would be generated.

2. Obtain hash string of the key pair generated to be programmed in SFP

   ```
   ./uni_sign --hash input_files/uni_sign/<platform>/input_uboot_secure
   ```

   This would provide you the 256 bit hash in form of string of the key pair generated in the previous step. The hash has to be programmed in the SRK hash Fuse.

3. Create CF header for u-boot Image.

   ```
   ./uni_cfsign input_files/uni_cfsign/<platform>/input_<device>_secure
   ```

   The input fields are specified in input_<device>_secure file. Please ensure that the filename mentioned in the input_<device>_secure is same as copied in the cst directory.

4. Create CSF header for u-boot Image.

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_uboot_secure
   ```

   The input fields are specified in input_uboot_secure file. Please ensure that the filename mentioned in the input_uboot_secure is same as copied in the cst directory.

5. Create CSF header for Linux uImage

   ./uni_sign input_files/uni_sign/<platform>/input_uimage_secure

   uImage.bin would be validated form u-boot. The flash address used here is according to the address map of u-boot. Please ensure that filename mentioned in the input_uimage_secure is same as copied in the cst directory.

6. Create CSF header for rootfs

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_rootfs_secure
   ```

```
Please make sure that filename mentioned in the input_rootfs_secure is same as copied in the cst
directory
```

7. Create CSF Header for hardware device tree

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_dtb_secure
   ```

   ```
   Please make sure that filename mentioned in the input_dtb_secure is same as copied in the cst
   directory
   ```

8. Create Boot script

   Bootscript is a U-Boot script image. Steps to create bootscript are given below :

   a. Create a text file bootscript.txt with following commands.

   ```
   esbc_validate <uImage CSF Header address>

   esbc_validate <dtb CSF Header address>

   esbc_validate <rootfs CSF Header address>

   bootm <uImage Address> <rootfs address> <dtb address>
   ```

   b. Then you will have to use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

   ```
   /tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript.txt
   bootscript
   ```

9. Generate CSF hdr for the boot script

   ./uni_sign input_files/uni_sign/<platform>/input_bootscript_secure

   The fields can be changed in the input files for the images based on the requirement.

10.3.2.4.2.2.2 Signing the images using different key pair

If boot script is also signed with a different key, remember to define the macro "**CONFIG_BOOTSCRIPT_KEY_HASH**" with the hash of the key used to sign the boot script in file *arch/powerpc/asm/include/fsl_secure_boot.h*. *ESBC u-boot would have to be recompiled if any change in this file is made*.

1. Generate the key pair to be used for signing the image

   ```
   ./gen_keys 1024 -p u-boot.priv -k u-boot.pub
   ```

   Key pair - public key file - u-boot.pub and private key in u-boot.priv would be generated.

2. Obtain hash string of the key pair generated to be programmed in SFP

   ```
   ./uni_sign --hash input_files/uni_sign/<platform>/input_uboot_secure
   ```

   This would provide you the 256 bit hash in form of string of the key pair generated in the previous step. The hash has to be programmed in the SRK hash Fuse.

3. Create CF header for u-boot Image.

   Open input_files/uni_cfsign/<platform>/input_<device>_secure and change PRI_KEY and PUB_KEY to u-boot.priv and u-boot.pub respectively and run the following command

   ```
   ./uni_cfsign input_files/uni_cfsign/<platform>/input_<device>_secure
   ```

4. Create CSF header for u-boot Image.

   Open input_files/uni_sign/<platform>/input_uboot_secure and change PRI_KEY and PUB_KEY to u-boot.priv and u-boot.pub respectively and run the following command

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_uboot_secure
   ```

5. Create CSF header for Linux uImage using different key pair.

   Repeat step 1 to generate another key pair.

   ```
   ./gen_keys 1024 -p lnx.priv -k lnx.pub
   ```

   Open `input_files/uni_sign/<platform>/input_uimage_secure` and change PRI_KEY and PUB_KEY to lnx.priv and lnx.pub respectively and run the following command. **`./uni_sign input_files/uni_sign/<platform>/input_uimage_secure`**

   Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <lnx_key_hash>

6. Create CSF header for rootfs

   ```
   ./gen_keys 1024 -p rootfs.priv -k rootfs.pub
   ```

   Open `input_files/uni_sign/<platform>/input_rootfs_secure` and change PRI_KEY and PUB_KEY to rootfs.priv and rootfs.pub respectively and run the following command.

   `./uni_sign input_files/uni_sign/<platform>/input_rootfs_secure`

   Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <rootfs_key_hash>

7. Create CSF Header for hardware device tree

   ```
   ./gen_keys 1024 -p dtb.priv -k dtb.pub
   ```

   Open `input_files/uni_sign/<platform>/input_dtb_secure` and change PRI_KEY and PUB_KEY to dtb.priv and dtb.pub respectively and run the following command. **`./uni_sign input_files/uni_sign/<platform>/input_dtb_secure`**

   Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <dtb_key_hash>

   Make sure that the fields in the input file input_rootfs_secure are same as mentioned in the address map

8. Write Boot script

   Bootscript is a U-Boot script image. Steps to create bootscript are given below :

   a. Create a text file bootscript.txt with following commands.

   ```
   esbc_validate <uImage CSF Header address>  <lnx_key_hash>
   esbc_validate <dtb CSF Header address>  <dtb_key_hash>
   esbc_validate <rootfs CSF Header address>  <rootfs_key_hahs>
   bootm <uImage Address> <rootfs address> <dtb address>
   ```

   NOTE

   Hashes would be the 256 bit string hash. These are the hashes of the key used to sign the respective images.

   b. Generate header over bootscript.txt which will be consumed by uboot command source

   tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript.txt bootscript

9. Generate CSF hdr for the boot script

   ```
   ./gen_keys 1024 -p bs.priv -k bs.pub
   ```

   Open `input_files/uni_sign/<platform>/input_dtb_secure` and change PRI_KEY and PUB_KEY to bs.priv and bs.pub respectively and run the following command.

   `./uni_sign input_files/uni_sign/<platform>/input_dtb_secure`

10.3.2.4.2.3 Running secure boot (Chain of Trust)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
   a. **Flow A**

      Program the ITS fuse.

      Or

   b. **Flow B**

      For protyping phase, don't blow the ITS fuse, instead set CFG_SB_DIS = 0. And put core in boot hold off mode.

   To put core in boot hold off mode :

   - P1010 : Set SW2.7 (CFG_BOOT_EN) = 1
   - BSC9132 : Set SW8.5 (CFG_BOOT_EN) = 0;
   - C293PCIE : Set SW7.1 (CFG_BOOT_EN) = 0

   To set CFG_SB_DIS = 0 on :

   - P1010 : Set SW2.1 (CFG_SB_DIS) = 0
   - BSC9131 : SW8.3 (CFG_SB_DIS) = 0 (for BSC9131QDS board)
   - BSC9132 : Set (CFG_SB_DIS) = 0 in DUTCFG12 (offset 0x06C) in FPGA image
   - C290 : Set SW7.6 (CFG_SB_DIS) = 0 (ON)

   For BSC9132 you can also program QIXIS registers from u-boot prompt using following i2c commands –

   i2c mw.l 0x66 0x6c 0xc7

   i2c mw.l 0x66 0x10 0x60

   i2c mw.l 0x66 0x10 0x30

   NOTE

   Don't power off the board if you want to retain the QIXIS settings

   In a manufacturing environment, it is recommended that all fuses be programmed at once, including the ITS and OEM Section Write Protect bits. In a prototyping environment, it may be preferable to leave ITS and Write Protect unprogrammed (relying on CFG_SB_DIS to initiate secure boot) until the developer has confidence in the secure boot process.

2. Blow other required fuses on the board (OTPMK, SRK hash and UID's).
   - OTPMK : It needs to be fused for both Flow A as well as Flow B.
   - SRK hash : It needs to be fused for Flow A but for Flow B it depends on choice either it can be fused or it can be written in shadow registers.
   - UID's: It needs to be fused for Flow A but for Flow B it depends on choice either it can be fused or it can be written in shadow registers.

   In case of Flow B, SRK hash and UID's can be written in shadow registers. As if, once fused it can't be changed.

   Note: SRK hash blown in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of Signing the images using same key pair on page 2032

3. Flashing images:
   - In case of NOR, flash-

     CF header : 0x0.

     CSF Header : as specified in CF Header.

ESBC u-boot : as specified in CSF Header.

and all other images are flashed in NOR flash at location as described in the address map (Address map used for the demo on page 2064

- In case of NAND/SPI/SD/SDHC, flash-

  CF header : 0x0.

  CSF Header : as specified in CF Header.

  ESBC u-boot : as specified in CSF Header.

  and all other images are flashed in NOR flash only at location as described in the address map (Address map used for the demo on page 2064

  NOTE

  In case of NAND/SPI/SD/SDHC secure boot, next level image address to be validated is configured in ESBC u-boot as NOR flash address.

a. **Flow A** - All the images would have to be flashed at the current bank addresses. Once ITS fuse is blown, the control would automatically shift to ISBC on power on.

b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.

4. Give a power on cycle to the board.

   a. For **Flow A** and **Flow B**

      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the signed linux, rootfs and dtb images
      - Linux would come up

   b. **Flow B**

      - Do switch settings according to the device (NAND/SPI/SD/SDHC).
      - Put the core in boot hold off mode.
      - On power on cycle, write SRKH, UIDs in the SFP shadow registers.
      - Bring the core out of boot hold off by writing to EEBPCR register.
      - On doing this, the secure boot flow as mentioned above would execute.

10.3.2.4.3 Chain of Trust with Confidentiality

This section presents the steps needed to be followed in order to execute Chain of Trust with confidentiality.

The demo would be divided into two parts:

1. Creating /encrypting images in form of blobs.
2. Decrypting the images, and booting from decrypted images.

Steps in the demo would be:

Step 1: Creating blobs

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to encapsulate next level images, i.e rootfs, linux uImage and device tree.

Step 2: Decrypting blob and booting

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to decapsulate/decrypt next level images, i.e rootfs, linux uImage and device tree.
5. After decryption, bootm command would be executed in boot script to pass control to Linux.

10.3.2.4.3.1 Other images required for the demo

Apart from SDK images described above, the following images are also required:

1. Encap Boot script
2. Decap Boot script
3. CF header for ESBC
4. CSF header for ESBC u-boot Image
5. CSF Header of the encap boot script
6. CSF Header of the decap boot script

The following section describes how to create the CSF headers and boot script.

10.3.2.4.3.2 Encap Bootscript

1. Create a bootscript_en.txt file with following commands:

```
blob enc <uImage address> 0x10000000 <uImage size> <key_modifier address>

erase <encapsulated uImage address> +<encapsulated uImage size>
cp.b 0x10000000 <encapsulated uImage address> <encapsulated uImage size>

blob enc <rootfs address> 0x20000000 <rootfs size> <key_modifier address>

erase <encapsulated rootfs address> +<encapsulated rootfs size>
cp.b 0x20000000 <encapsulated rootfs address>  <encapsulated rootfs size>

blob enc <dtb address> 0x1000000 <btb size> <key_modifier address>

erase <encapsulated dtb address> +<encapsulated dtb size>
cp.b 0x1000000 <encapsulated dtb address>  <encapsulated dtb size>
```

   For the addresses to load images refer Section Address map used for the demo on page 2064

2. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript_en.txt
bootscript_encap
```

10.3.2.4.3.3 Decap Bootscript

1. Create a bootscript_de.txt file with following commands:

```
blob dec <encapsulated uImage address> 0x10000000 <uImage size + 0x30> <key_modifier address>

blob dec <encapsulated rootfs address> 0x20000000 <rootfs size + 0x30>  <key_modifier address>

blob dec <encapsulated dtb address> 0x1000000 <dtb size + 0x30>  <key_modifier address>
```

```
bootm 0x10000000 0x20000000 0x1000000
```

For the addresses to load images refer Section Address map used for the demo on page 2064

The script decapsulates/decrypts the blob created by earlier boot script and boots using them.

NOTE

0x30(48 bytes) are added at the end of the encapsulated image which needs to be added while providing the size of image to be decapsulated in blob dec command.

2. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript_de.txt
bootscript_decap
```

10.3.2.4.3.4 Creating CSF Headers

- **CSF Header for ESBC**

  Use the command given below to generate the hdr for u-boot binary.

  ./uni_sign input_files/uni_sign/<platform>/<input file for uboot>

  Please change the binary name as per your uboot binary in “IMAGE_1”.

- **CSF Header for bootscript_encap and bootscript_decap**

  Use the command given below to generate the headers for bootscripts

  ./uni_sign input_files/uni_sign/<platform>/<input file for bootscript>

  Please change the binary name as per your bootscript in “IMAGE_1”.

10.3.2.4.3.5 Running secure boot (Chain of Trust with Confidentiality)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
   a. **Flow A**

      Program the ITS fuse.

      Or

   b. **Flow B**

      For protyping phase, don't blow the ITS fuse, instead set CFG_SB_DIS = 0 and put the core in boot hold off mode as described in Running secure boot (Chain of Trust) on page 2035 .

2. Blow other required fuses on the board (OTPMK, SRK hash and UID's).
   - OTPMK : It needs to be fused for both Flow A as well as Flow B.
   - SRK hash : It needs to be fused for Flow A but for Flow B it depends on choice either it can be fused or it can be written in shadow registers.
   - UID's: It needs to be fused for Flow A but for Flow B it depends on choice either it can be fused or it can be written in shadow registers.

   In case of Flow B, SRK hash and UID's can be written in shadow registers. As if, once fused it can't be changed.

   Note: SRK hash blown in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of Signing the images using same key pair on page 2032

3. Flashing images:

- In case of NOR, flash-

  CF header : 0x0.

  CSF Header : as specified in CF Header.

  ESBC u-boot : as specified in CSF Header.

  and all other images are flashed in NOR flash at location as described in the address map (Address map used for the demo on page 2064

- In case of NAND/SPI/SD/SDHC, flash-

  CF header : 0x0.

  CSF Header : as specified in CF Header.

  ESBC u-boot : as specified in CSF Header.

  and all other images are flashed in NOR flash only at location as described in the address map (Address map used for the demo on page 2064

a. **Flow A** - All the images would have to be flashed at the current bank addresses. Once ITS fuse is blown, the control would automatically shift to ISBC on power on.

b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.

4. Give a power on cycle to the board.

   a. For **Flow A** and **Flow B**

      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the bootscript.
      - Bootscript would encapsulate the Linux, rootfs and device tree and store the blobs at the desired locations.

   b. **Flow B**

      - Do switch settings according to the device (NAND/SPI/SD/SDHC).
      - Put the core in boot hold off mode.
      - On power on cycle, write SRKH, UIDs in the SFP shadow registers.
      - Bring the core out of boot hold off by writing to EEBPCR register.
      - On doing this, the secure boot flow as mentioned above would execute.

5. Replace the encap bootscript and its CSF header with decap bootscript and the CSF header of the decap bootscript respectively.

6. Give a power on cycle to the board.

   a. For **Flow A** and **Flow B**

      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the bootscript.
      - Bootscript would decapsulate the Linux, rootfs and device tree bloband store them on DDR
      - Bootm commnd in bootscript would execute on successful decapsulation
      - Linux prompt would come up. .

   b. **Flow B**

      - On power on cycle, u-boot prompt on bank 0 would come up.
      - On switching to alternate bank, the secure boot flow as mentioned above would execute.

10.3.2.5 Troubleshooting

Table 349. Troubleshooting

	Symptoms	Reasons and/or Recommended actions
1.	No print on UART console.	• Check the status register of sec mon block (location CCSRBAR + 0xe6014). Refer to the details of the register from the Reference Manual. Bits OTPMK_ZERO, OTMPK_SYNDROME and PE should be 0 otherwise there is some error in the OTPMK fuse blown by you. • If OTMPK fuse is correct (see Step 1), check the GUTS SBDCR (CCSRBAR + 0xe0f90) register for errors. Refer to Validation failures on page 2024 for error codes. • If **Error code = 0** then check the Security Monitor state in HPSR register of Sec Mon. **Sec Mon in Check State (0x9)** If ITS fuse = 1, then it means ISBC code has reset the board. This may be due to the following reasons: Hash of the public key used to sign the ESBC u-boot doesn't match with the value in SRK Hash Fuse Or Signature verification of the image failed **Sec Mon in Trusted State (0xd) or Non Secure State (0xb)** Check the entry point field in the ESBC header. It should be 0xcfffffffc for the demo described in Section 4. If entry point is correct, ensure that u-boot image has been signed with the correct input file.
2.	Instead of linux prompt, you get a u-boot command prompt .	You have not booted in secure boot mode. You never get a u-boot prompt in secure boot flow. Check Step 1 in Running secure boot (Chain of Trust) on page 2035. You would reach this stage if ITS = 0 and you are running normal uboot.
3	u-boot hangs or board resets	Some validation failure occurred in ESBC u-boot. Error code and description would be printed on u-boot console. Refer to Validation Failures from ESBC on page 2024 for more details on errors.

10.3.2.6 Trust Architecture and SFP Information

SoC	Trust Arch. Version	SFP Version	POVDD	DRVR		OTPMK	
				Algo (CST)	Register to check Hamming Error	Algo (CST)	Register to check Hamming Error

Table continues on the next page...

Table continued from the previous page...

P1010	1.0	2.0	1.5 V	B	None/ Simulation	1	SecMon_HP Status (HPSR)
BSC913x	1.0	2.1	1.5 V	B	None/ Simulation	1	SecMon_HP Status (HPSR)
C290	2.0	2.4	1.5 V	B	None/ Simulation	1	None/ Simulation

10.3.2.7 CF Header Data Structure Definition

The IBSC running on the core first authenticates the CF Header and then executes the configuration words.

P1010/ 9131/ 9132 Platforms

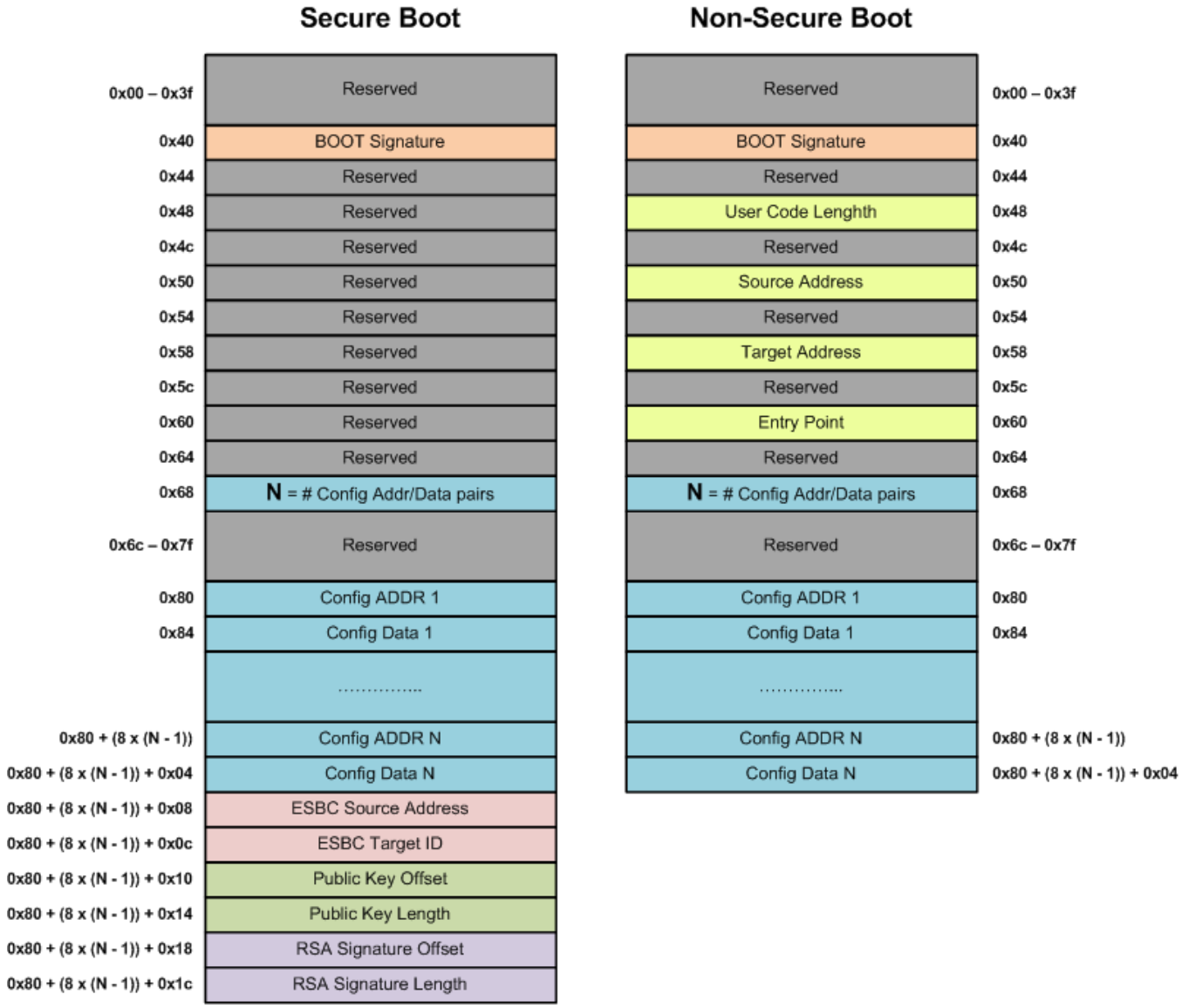


Figure 329. CF Header for P1010/ 9131/ 9132

Table 350. CF Header Format (P1010/ 9131/ 9132 Platforms)

Offset	Data Bits [0:31]
0x00-0x3F	**Reserved**
0x40-0x43	**BOOT signature**. This location should contain the value 0x424f_4f54, which is the ascii code for BOOT. The boot loader code searches for this signature. If the value in this location does not match the BOOT signature or the SD/MMC card does not contain a valid user code, the boot loader code disables the eSDHC and issues a hardware reset request of the SoC by setting RSTCR[HRESET_REQ].

Table continues on the next page...

Table 350. CF Header Format (P1010/ 9131/ 9132 Platforms) (continued)

Offset	Data Bits [0:31]
0x44-0x47	Reserved
0x48-0x4b	**User's code length**. Number of bytes in the user's code to be copied. This must be a multiple of the SD/MMC card's block size (and the user's code zero-padded if necessary to achieve that length). User's code length <= 2 GBytes. This is treated as reserved in Secure Boot Mode
0x4c-0x4f	Reserved
0x50-0x53	**Source Address**. Contains the starting address of the user's code as an offset from the SD/MMC card starting address. In Standard Capacity SD/MMC Cards, the 32-bit Source Address specifies the memory address in byte address format. This must be a multiple of the SD/MMC card's block size. In High Capacity SD Cards (>2 GBytes), the 32-bit Source Address specifies the memory address in block address format. Block length is fixed to 512 bytes as per the SD High Capacity specification . This is treated as reserved in Secure Boot Mode .
0x54-0x57	Reserved
0x58-0x5b	**Target Address**. Contains the target address in the system's local memory address space in which the user's code is copied to. This is a 32-bit effective address. The core is configured in such a way that the 36-bit real address is equal to this (with 4 most significant bits zero) . This is treated as reserved in Secure Boot Mode
0x5c-0x5f	Reserved
0x60-0x63	**Entry Point**. Contains the jump address in the system's local memory address space into the user's code first instruction to be executed. This is a 32-bit effective address. The core is configured in such a way that the 36-bit real address is equal to this (with 4 most significant bits zero). This is treated as reserved in Secure Boot Mode.
0x64-0x67	Reserved
0x68-0x6b	**N. Number of Configuration Address/Data pairs**. Must be 1<=N<=44 if running in secure mode. Must be 1<=N<=48 if running in legacy mode. (but is recommended to be as small as possible).

Table continues on the next page...

Table 350. CF Header Format (P1010/ 9131/ 9132 Platforms) (continued)

Offset	Data Bits [0:31]
0x6c-0x7f	Reserved
0x80-0x83	Configuration Address 1
0x84-0x87	Configuration Data 1
0x88-0x8b	Configuration Address 2
0x8c-0x8f	Configuration Data 2
.....	.....
0x80 + 8*(N-1)	Configuration Address N
0x80 + 8*(N-1)+4	Configuration Data N
0x80 + 8*(N-1)+8	**ESBC Source Address**
0x80 + 8*(N-1) + 0x0C	**ESBC Location Identifier (Target ID)** The TargetID should be same as cfg_rom_loc.
0x80 + 8*(N-1)+0x10	**Public Key Offset** from BOOT Signature (Actual Offset – 0x40)
0x80 + 8*(N-1) + 0x14	**Public Key Length**
0x80 + 8*(N-1) + 0x18	**RSA CF Header Signature Offset** from BOOT Signature (Actual Offset – 0x40)
0x80 + 8*(N-1) + 0x1C	**RSA CF Header Signature Length**

Table 351. Signature (P1010/ 9131/ 9132 Platforms)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CF Header.

Table 352. Public key (P1010/ 9131/ 9132 Platforms)

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

C290 Platform

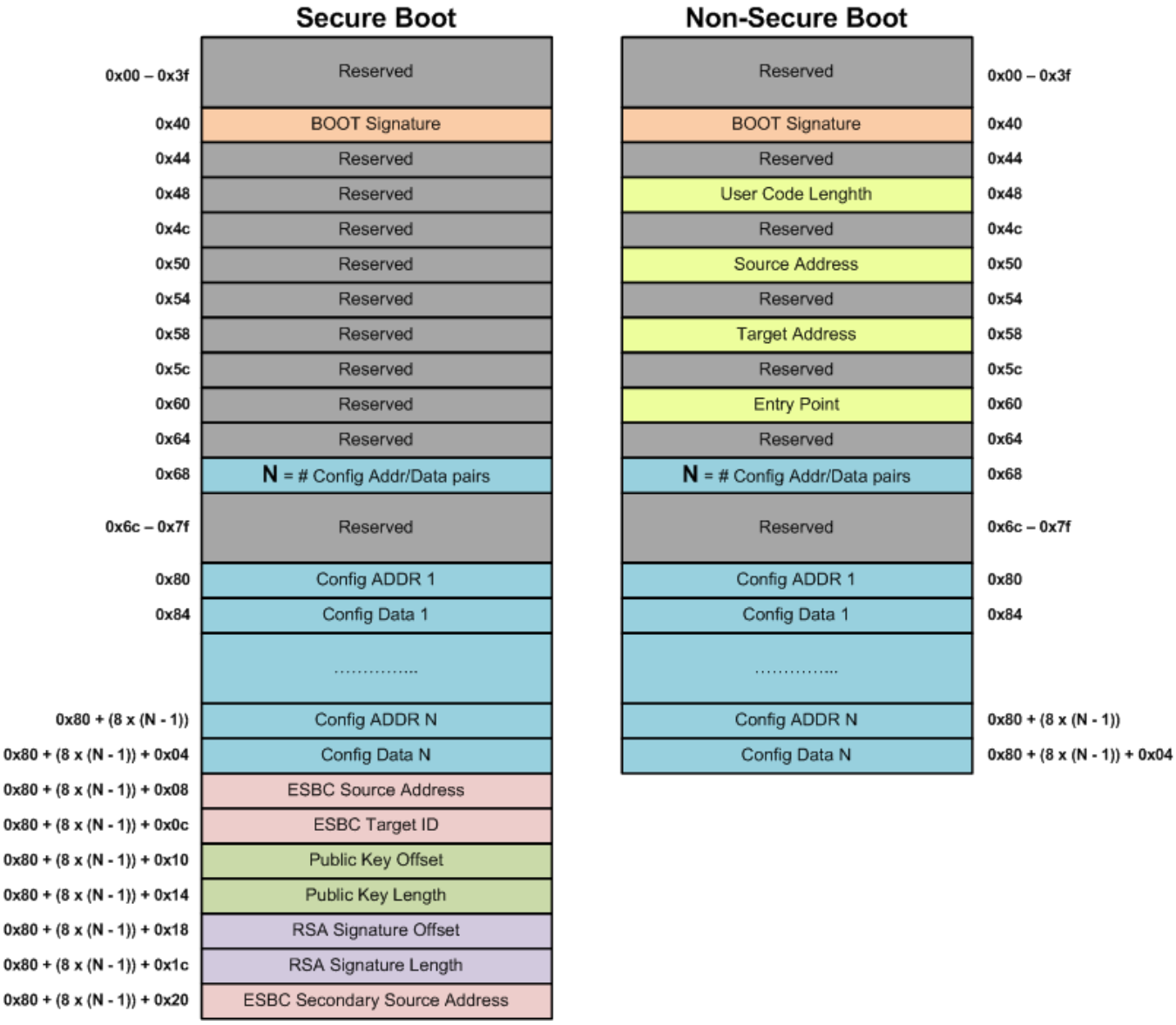


Figure 330. CF Header for C290

Table 353. CF Header Format (C290 Platform)

Offset	Data Bits [0:31]
0x00-0x3F	Reserved
0x40-0x43	**BOOT signature**. This location should contain the value 0x424f_4f54, which is the ascii code for BOOT. The boot loader code searches for this signature. If the value in this location does not match the BOOT signature or the SD/MMC card does not contain a valid user code, the boot loader code disables the eSDHC and issues a hardware reset request of the SoC by setting RSTCR[HRESET_REQ].

Table continues on the next page...

Table 353. CF Header Format (C290 Platform) (continued)

Offset	Data Bits [0:31]
0x44-0x47	Reserved
0x48-0x4b	**User's code length**. Number of bytes in the user's code to be copied. This must be a multiple of the SD/MMC card's block size (and the user's code zero-padded if necessary to achieve that length). User's code length <= 2 GBytes. This is treated as reserved in Secure Boot Mode
0x4c-0x4f	Reserved
0x50-0x53	**Source Address**. Contains the starting address of the user's code as an offset from the SD/MMC card starting address. In Standard Capacity SD/MMC Cards, the 32-bit Source Address specifies the memory address in byte address format. This must be a multiple of the SD/MMC card's block size. In High Capacity SD Cards (>2 GBytes), the 32-bit Source Address specifies the memory address in block address format. Block length is fixed to 512 bytes as per the SD High Capacity specification . This is treated as reserved in Secure Boot Mode .
0x54-0x57	Reserved
0x58-0x5b	**Target Address**. Contains the target address in the system's local memory address space in which the user's code is copied to. This is a 32-bit effective address. The core is configured in such a way that the 36-bit real address is equal to this (with 4 most significant bits zero) . This is treated as reserved in Secure Boot Mode
0x5c-0x5f	Reserved
0x60-0x63	Entry Point. Contains the jump address in the system's local memory address space into the user's code first instruction to be executed. This is a 32-bit effective address. The core is configured in such a way that the 36-bit real address is equal to this (with 4 most significant bits zero). This is treated as reserved in Secure Boot Mode.
0x64-0x67	Reserved
0x68-0x6b	**N. Number of Configuration Address/Data pairs**. Must be 1<=N<=44 if running in secure mode. Must be 1<=N<=48 if running in legacy mode. (but is recommended to be as small as possible).

Table continues on the next page...

Table 353. CF Header Format (C290 Platform) (continued)

Offset	Data Bits [0:31]
0x6c-0x7f	Reserved
0x80-0x83	Configuration Address 1
0x84-0x87	Configuration Data 1
0x88-0x8b	Configuration Address 2
0x8c-0x8f	Configuration Data 2
.....	.....
0x80 + 8*(N-1)	Configuration Address N
0x80 + 8*(N-1)+4	Configuration Data N
0x80 + 8*(N-1)+8	**ESBC Source Address**
0x80 + 8*(N-1) + 0x0C	**ESBC Location Identifier (Target ID)** The TargetID should be same as cfg_rom_loc.
0x80 + 8*(N-1)+0x10	**SRK Table/ Public Key Offset** from BOOT Signature (Actual Offset – 0x40)
0x80 + 8*(N-1) + 0x14	**Public Key Lenght/ SRK Table Structure**. First four bits contains information about srk_table_flag. Next 12 bits contains srk_sel i.e. key, to be selected for use. And final 16 bits contains num_srk i.e. number of keys available.
0x80 + 8*(N-1) + 0x18	**RSA CF Header Signature Offset** from BOOT Signature (Actual Offset – 0x40)
0x80 + 8*(N-1) + 0x1C	**RSA CF Header Signature Length**
0x80 + 8*(N-1) + 0x20	**ESBC secondary image location** (Secondary Image source address).

Table 354. Signature (C290 Platform)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CF Header.

Table 355. SRK Table (C290 Platform)

Offset	Data Bits [0:31]
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

10.3.2.8 CSF Header Data Structure Definition

The CSF Header provides the ISBC with most of the information needed to validate the image.

P1010/ 9131/ 9132 Platforms

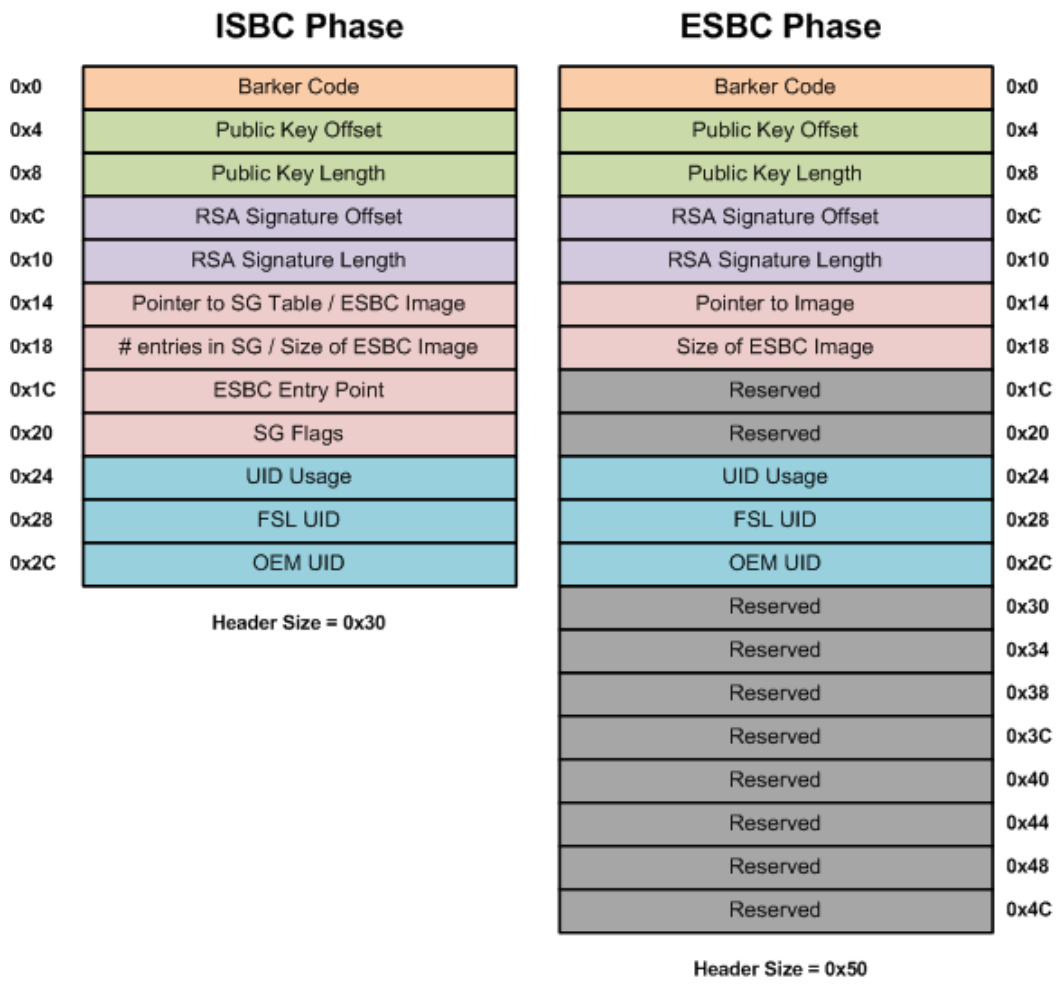


Figure 331. CSF Header for P1010/ 9131/ 9132 (ISBC and ESBC Phase)

Table 356. CSF Header Format (P1010/ 9131/ 9132 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	**Barker code**. This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.
0x04-0x07	**Public key offset**. This location contains an offset in bytes of the public key from the start of CSF header. Using this offset and the public key length, the public key is read.
0x08-0x0b	**Public key length in bytes**. (Value populated here should be twice of Modulus size). Supported sizes are 256, 512 or 1024 bytes (2 * 1024, 2 * 2048 , 2 * 4096 bits).

Table continues on the next page...

Table 356. CSF Header Format (P1010/ 9131/ 9132 Platforms) (continued)

Offset	Data Bits [0:31]
0x0c-0x0f	**RSA Signature offset**. This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x10-0x13	**RSA Signature length in bytes**.
0x14-0x17	**For ISBC Phase**: Based on the Scatter Gather flag in CSF header, this location can either be treated as **Pointer to Scatter Gather table or the address of ESBC image**. **For ESBC Phase**: This location is treated as **address of image**(linux/bootscript/rootfs/dtb) to be validated.
0x18-0x1b	**For ISBC Phase**: Based on the Scatter gather flag in CSF Header, this location can either be treated as **number of entries in SG table or ESBC image size** in bytes. **For ESBC Phase**: Size of image to be validated.
0x1c-0x1f	**For ISBC Phase**: **ESBC entry point**. ISBC transfers control to this location upon successful validation of ESBC image(s). **For ESBC Phase**: Reserved.
0x20-0x23	**Reserved** .(Earlier this field was SG Flag. SG flag is always assumed to be 1)
0x24-0x27	**Unique ID Usage**. UIDs present in the CSF Header are compared to the corresponding UIDs in the SFP, and are included in the ESBC validation. 0x0000 - No UIDs are present in CSF header 0x0001 - FSL_UID and OEM_UID are present in CSF header 0x0002 - Only OEM_UID present in CSF header 0x0004 - Only FSL_UID present in CSF header
0x28-0x2b	**NXP unique ID**. A unique 32 bit value, which is specific to NXP. This value is compared with the FSL ID in Secure Fuse Processor 's FSL-ID registers

Table continues on the next page...

Table 356. CSF Header Format (P1010/ 9131/ 9132 Platforms) (continued)

Offset	Data Bits [0:31]
0x2c-0x2f	**OEM unique ID**. A unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID in Secure Fuse Processor 's OEM-ID registers
0x30-0x4f	For ISBC Phase: Not Applicable For ESBC Phase: Reserved

Table 357. Scatter Gather Table Format (P1010/9131/9132 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1. **NOTE** For high capacity SD/MMC cards(>2G), this must be a multiple of the card's block size (and the user's code zero-padded if necessary to achieve that length).
0x04-0x07	Target where the ESBC Image 1 can be found.
0x08-0x0b	Source Address of ESBC Image 1 **NOTE** For high capacity SD/MMC cards(>2G), the address is in block address format. Block length is fixed to 512 bytes as per the SD High Capacity specification.
0x0c-0x0f	Destination Address of ESBC Image 1 If booting from NOR, the image is not required to be copied and this filed is programmed as 0xFFFFFFFF. For other devices (NAND, SPI, SD/MMC) this field contains the address at which the ESBC image will be copied.
0x10-0x13	Length of ESBC image 2
0x14-0x17	Target of ESBC Image 2
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	Destination Address of ESBC Image 2
.	
.	
.	
0x70-0x73	Length of ESBC image 8

Table continues on the next page...

Table 357. Scatter Gather Table Format (P1010/9131/9132 Platforms) (continued)

Offset	Data Bits [0:31]
0x74-0x77	Target of ESBC Image 8
0x78-0x7b	Source Address of ESBC Image 8
0x7c-0x7f	Destination Address of ESBC Image 8

Table 358. Signature (P1010/ 9131/ 9132 Platforms)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

Table 359. Public key (P1010/ 9131/ 9132 Platforms)

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

C290 Platform

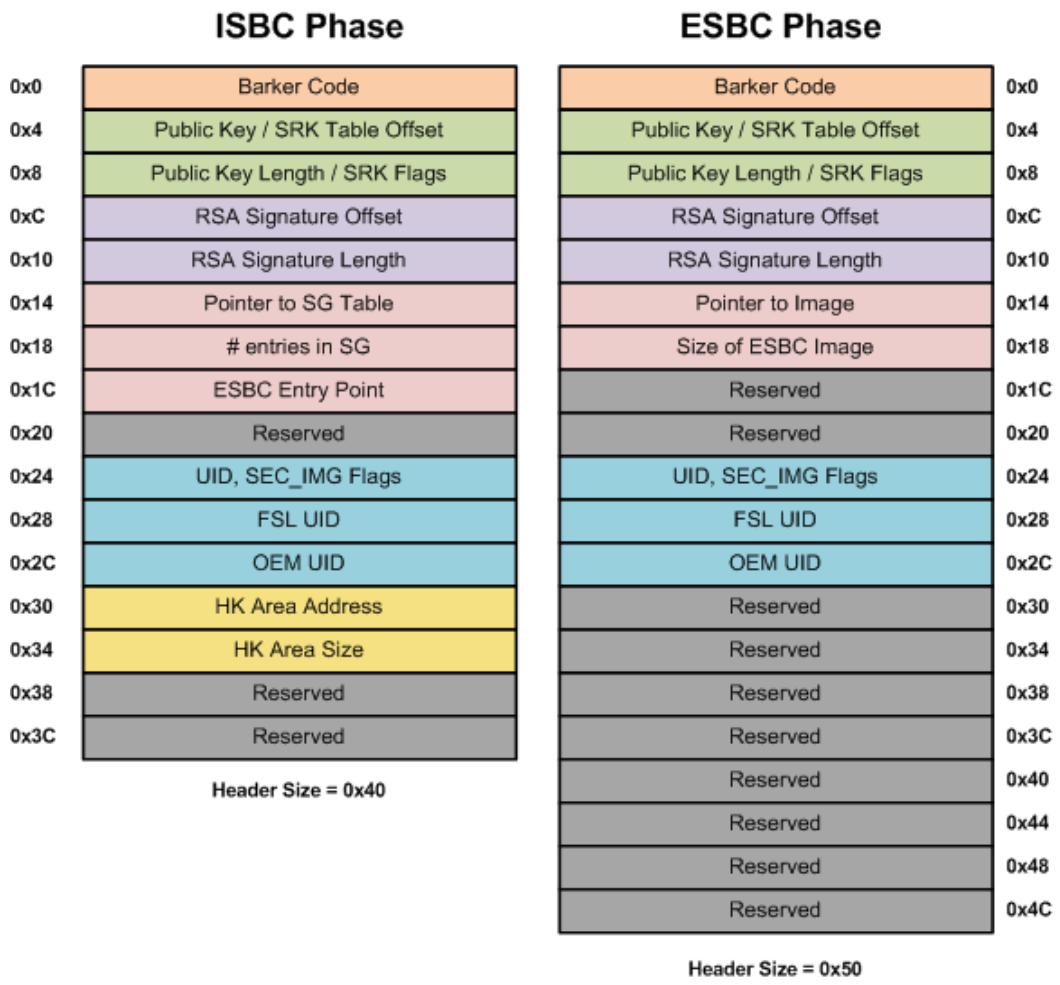


Figure 332. CSF Header for C290 (ISBC and ESBC Phase)

Table 360. CSF Header Format (C290 Platform)

Offset	Data Bits [0:31]
0x00-0x03	**Barker code**. This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.
0x04-0x07	If the srk_table_flag is not set : • **Public key offset**: This location contains an address which is the offset of the public key from the start of CSF header. Using this offset and the public key length, the public key is read. If srk_table_flag is set: • **Srk table offset**: This location contains an address which is the offset of the srk table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.

Table continues on the next page...

Table 360. CSF Header Format (C290 Platform) (continued)

Offset	Data Bits [0:31]
0x08	**Srk table flag**. This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.
0x09-0x0b	If the srk_table_flag is not set : • **Public key length**: This location contains the length of the public key in bytes. If srk_table_flag is set: • 0x09 – **Key Number** from srk table which is to be used for verification. • 0x0a-0x0b – **Number of entries** in srk table. Minimum number of entries in table = 1, Maximum = 4.
0x0c-0x0f	**RSA Signature offset**. This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x10-0x13	**RSA Signature length in bytes**.
0x14-0x17	**For ISBC Phase**: **SG Table offset** This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries is SG Table, the SG table is read. **For ESBC Phase**: **Address of the image to be validated**.
0x18-0x1b	**For ISBC Phase**: **Number of entries in SG Table** (Earlier ,Based on the Scatter gather flag in CSF Header, this location can either be treated as number of entries in SG table or ESBC image size in bytes.). **For ESBC Phase**: **Size of image** to be validated
0x1c-0x1f	**For ISBC Phase**: **ESBC entry point**. ISBC transfers control to this location upon successful validation of ESBC image(s). **For ESBC Phase**: Reserved
0x20-0x23	**Reserved** .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.)
0x24	For ISBC Phase: Reserved For ESBC Phase: Reserved

Table continues on the next page...

Table 360. CSF Header Format (C290 Platform) (continued)

Offset	Data Bits [0:31]
0x25	**For ISBC Phase**: **Secondary Image flag** Indicates if user has a secondary image available in case of failures in validating primary iamge.Valid in case of primary Images's Header. **For ESBC Phase**: Reserved
0x26-0x27	**Unique ID Usage**. UIDs present in the CSF Header are compared to the corresponding UIDs in the SFP, and are included in the ESBC validation. 0x00 - No UIDs are present in CSF header 0x01 - FSL_UID and OEM_UID are present in CSF header 0x02 - Only OEM_UID present in CSF header 0x04 - Only FSL_UID present in CSF header
0x28-0x2b	**NXP unique ID**. A unique 32 bit value, which is specific to NXP. This value is compared with the FSL ID in Secure Fuse Processor 's FSL-ID registers
0x2c-0x2f	**OEM unique ID**. A unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID in Secure Fuse Processor 's OEM-ID registers
0x30-0x33	**For ISBC Phase**: **Housekeeping area address** This is address of start of a memory which can be accessed by devices on SOC bus. (DDR, L3 cache configured as SRAM). The area should have been pre-configured by user through PBL commands or configuration header. **For ESBC Phase**: Reserved
0x34-0x37	**For ISBC Phase**: **Size of the housekeeping area** Size of the pre-configured memory which can be used by Boot Rom Code. **For ESBC Phase**: Reserved.
0x38-0x3f	Reserved
0x40-0x4f	For ISBC Phase: Not Applicable For ESBC Phase: Reserved

Table 361. Scatter Gather Table Format (C290 Platform)

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1. **NOTE** For high capacity SD/MMC cards(>2G), this must be a multiple of the card's block size (and the user's code zero-padded if necessary to achieve that length).
0x04-0x07	Target where the ESBC Image 1 can be found.
0x08-0x0b	Source Address of ESBC Image 1 **NOTE** For high capacity SD/MMC cards(>2G), the address is in block address format. Block length is fixed to 512 bytes as per the SD High Capacity specification.
0x0c-0x0f	Destination Address of ESBC Image 1 If booting from NOR, the image is not required to be copied and this filed is programmed as 0xFFFFFFFF. For other devices (NAND, SPI, SD/MMC) this field contains the address at which the ESBC image will be copied.
0x10-0x13	Length of ESBC image 2
0x14-0x17	Target of ESBC Image 2
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	Destination Address of ESBC Image 2
.	
.	
.	
0x70-0x73	Length of ESBC image 8
0x74-0x77	Target of ESBC Image 8
0x78-0x7b	Source Address of ESBC Image 8
0x7c-0x7f	Destination Address of ESBC Image 8

Table 362. Signature (C290 Platform)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

Table 363. Public key (C290 Platform)

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

Table 364. SRK Table (C290 Platform)

Offset	Data Bits [0:31]
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

10.3.2.9 ISBC Validation Error Codes

There are many reasons the IBR could fail to validate the ESBC. Technicians with debug access can check the GUTS Internal BootROM Error Code Register (IBRECR: CCSRBAR + 0xe0f90) register to obtain an error code

P1010/ 9131/ 9132 platforms

Table 365. ISBC Validation Failures (P1010/ 9131/ 9132 platforms)

Value	Code	Definition
0x1	ERROR_CRITICAL	Error due to critical exceptions
0x2	ERROR_BAD_ADDRESS	IBR accessed any unmapped address.
0x3	ERROR_MISC	Miscellaneous Exception
0x100	ERROR_CORE_NON_ZERO	Core is not boot core i.e core0
0x101	ERROR_STATE_NOT_CHECK	SNVS not in CHECK state
0x102	ERROR2_STATE_NOT_CHECK	SEC_MON State Machine not in CHECK state, when trying to transition it to Trusted/Non Secure/Soft Fail state

Table continues on the next page...

Table 365. ISBC Validation Failures (P1010/ 9131/ 9132 platforms) (continued)

Value	Code	Definition
0x103	ERROR_TARGET_NOT_SUPPORTED_IN_NON_SECURE	The target is not supported in legacy mode
0x201	ERROR_BOOT_SIG_MISMATCH	Boot signature mismatch in CF header
0x202	ERROR_CF_HASH_COMPARE_EM	CF header signature failure
0x204	ERROR_CF_HEADER_KEY_LEN	Wrong key length in CF header
0x205	ERROR_CF_HEADER_SIG_LEN	Wrong signature length in CF header
0x206	ERROR_CF_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Key length not twice of signature length
0x207	ERROR_CF_HEADER_USER_CODE_LEN	Wrong user code length in CF header
0x208	ERROR_CF_HEADER_CMDS_EXCED_LIMIT	More than allowed configuration commands in CF header
0x209	ERROR_ESBCHDRLOC_NOT_ALIGNED	ESBC header address not aligned to 512 byte boundary in case of SD/MMC
0x20a	ERROR_CF_HASH_COMPARE_KEY	Key hash failure in CF header
0x301	ERROR_ESBC_HDR_LOC	ESBC header not within 3.5G boundary
0x302	ERROR_ESBC_HEADER_BARKER	Barker code error in ESBC header
0x303	ERROR_ESBC_HEADER_KEY_LEN	Wrong key length in ESBC header
0x304	ERROR_ESBC_HEADER_SIG_LEN	Wrong signature length in ESBC header
0x305	ERROR_ESBC_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Key length not twice of signature length in ESBC header
0x306	ERROR_ESBC_HEADER_SG_TABLE_ADDR_NULL	SG table address is NULL
0x307	ERROR_ESBC_HEADER_SG_TABLE_ADDR_NOT_IN_3_5G	SG table address not within 3.5G boundary
0x308	ERROR_ESBC_HEADER_KEY_MOD_1	Public key Modulus most significant bit not set
0x309	ERROR_ESBC_HEADER_KEY_MOD_2	Public key Modulus in header not odd
0x30a	ERROR_ESBC_HEADER_SIG_KEY_MOD	Signature not less than modulus

Table continues on the next page...

Table 365. ISBC Validation Failures (P1010/ 9131/ 9132 platforms) (continued)

Value	Code	Definition
0x30b	ERROR_ESBC_HEADER_SG_ENTRIES_NULL	SG entries address is NULL
0x30c	ERROR_ESBC_HEADER_SG_ENTRIES_NOT_IN_3_5G	SG entries not within 3.5G boundary
0x30d	ERROR_ESBC_HEADER_SG_ESBC_EP	ESBC entry point not within one of the SG entries
0x30e	ERROR_HASH_COMPARE_KEY	Key hash failure in ESBC header
0x30f	ERROR_HASH_COMPARE_EM	Signature failure in ESBC header
0x310	ERROR_SSM_CHECKSTS	SNVS not in CHECK state
0x311	ERROR_SSM_TRUSTSTS	SNVS not in TRUSTED state
0x312	ERROR_FSL_UID	FSL UIDs present in ESBC header and SFP don't match
0x313	ERROR_OEM_UID	OEM UIDs present in ESBC header and SFP don't match
0x315	ERROR_SGL_ENTIRES_NOT_SUPPORTED	Number of entries in SG table exceeds maximum limit i.e 8
0x316	ERROR_SG_ENTRY_NOT_ALIGNED	SG entries address not aligned in case of SD/MMC
0x317	ESBCHDR_PKEY_NOT_ALIGNED	Key offset not aligned in case of SD/MMC
0x318	ESBCHDR_SIGN_NOT_ALIGNED	Signature offset not aligned in case of SD/MMC
0x401	ERROR_NAND_READ_TIMEOUT	Timeout while reading from NAND device.
0x403	ERROR_NAND_FTOER	FTOER error in NAND EVTER during Boot Done
0x404	ERROR_NAND_ECCER	ECCER error in NAND EVTER during Boot Done
0x405	ERROR_NAND_BBT_FULL	Bad Block Table is full
0x500	ERROR_ESDHC_CARD_DETECT_FAIL	No card present or card not inserted properly in the sdhc slot
0x501	ERROR_ESDHC_UNUSABLE_CARD	The card is not in usable state
0x502	ERROR_ESDHC_COMMUNICATION_ERROR	Communication Error
0x503	ERROR_ESDHC_READ_UNALIGNED	The offset being read is not aligned with the block size
0x600	ERROR_ESPI_READID	The Flash ID is invalid.
0x601	ERROR_ESPI_BOOT_SIGN	Unable to read Boot SIGN in both 24 bit and 16 bit mode.

Table continues on the next page...

Table 365. ISBC Validation Failures (P1010/ 9131/ 9132 platforms) (continued)

Value	Code	Definition
0x602	ERROR_ESPI_READ_TIMEOUT	Timeout while reading data from SPI.

C29x platforms

When the device is booted with the assistance of code from the Internal Boot ROM, including normal boot from SPI or SD and secure boot from any interface, the errors are logged in the GUTS registers IBRECR1 or IBRECR2. These can be read via debugger to determine the reason for failure.

Errors in the system can be of following types:

1. Core Exceptions
2. System State Failures
3. Header Checking Failures
   a. CF Header Checking Failures
   b. General CSF (ESBC) Header Checking Failures
   c. Key/Signature/UID related errors
4. Verification Failures
5. Driver errors
6. SEC errors

Internal BootROM Error Code Register 1 (IBRECR1)	CCSRBAR + 0xe0f90 Error code for validation of primary image and all core exceptions, system state failures and CF Header checking failures.
Internal BootROM Error Code Register 2 (IBRECR2)	CCSRBAR + 0xe0f94 Error code for validation of secondary image. Core Exceptions, System State Failures and CF Header Checking Failures cannot occur during verification of the secondary/ alternate image, and are not observed in this register.

Table 366. Core Exceptions (C29x platform)

Value	Code	Definition
0x1	ERROR_MACHINECHECK	Machine check Exception
0x2	ERROR_DSI	DSI Exception
0x3	ERROR_ISI	ISI Exception
0x4	ERROR_CRITICAL	Critical Exception
0x5	ERROR_ALIGN	Alignment Exception
0x6	ERROR_PROG	Program Exception
0x13	ERROR_DATA_TLB	Data TLB Miss

Table continues on the next page...

Table 366. Core Exceptions (C29x platform) (continued)

Value	Code	Definition
0x14	ERROR_INST_TLB	Instruction TLb Miss
0x20	ERROR_MISC	Any other exception

Table 367. System State Failures (C29x platform)

Value	Code	Definition
0x100	ERROR_CORE_NON_ZERO	ISBC is not running on CPU0
0x101	ERROR_STATE_NOT_CHECK	SEC_MON State Machine not in CHECK state at start of ISBC. Some Security violation could have occurred.
0x102	ERROR2_STATE_NOT_CHECK	SEC_MON State Machine not in CHECK state, when trying to transition it to Trusted/Non Secure/Soft Fail state
0x103	ERROR_SSM_TRUSTSTS	SEC_MON State Machine not in TRUSTED state at end of ISBC.

Table 368. CF Header Checking Failures for Non Secure Boot (C29x platforms)

Value	Code	Definition
0x201	ERROR_BOOT_SIG_MISMATCH	Boot signature mismatch in CF header
0x208	ERROR_CF_HEADER_CMDS_EXCEED_LIMIT	More than allowed configuration commands in CF header

Table 369. CF Header Checking Failures for Secure Boot (C29x platforms)

Value	Code	Definition
0x201	ERROR_BOOT_SIG_MISMATCH	Boot signature mismatch in CF header
0x202	ERROR_CF_HASH_COMPARE_EM	CF header signature failure
0x204	ERROR_CF_HEADER_KEY_LEN	Wrong key length in CF header
0x205	ERROR_CF_HEADER_SIG_LEN	Wrong signature length in CF header
0x206	ERROR_CF_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Key length not twice of signature length
0x207	ERROR_CF_HEADER_USER_CODE_LEN	Wrong user code length in CF header
0x208	ERROR_CF_HEADER_CMDS_EXCEED_LIMIT	More than allowed configuration commands in CF header
0x20a	ERROR_CF_HASH_COMPARE	Key hash failure in CF header

Table continues on the next page...

Table 369. CF Header Checking Failures for Secure Boot (C29x platforms) (continued)

Value	Code	Definition
0x20b	ERROR_CF_INVALID_SRK_TBL	Number of entries field in CSF Header is 0 or greater than 4(This is when srk_flag in header is 1)
0x20c	ERROR_CF_INVALID_KEY_NUM	Key number to be used from srk table is not present in table. (This is when srk_flag in header is 1)
0x20d	ERROR_CF_KEY_REVOKED	Key selected from srk table has been revoked(This is when srk_flag in header is 1)
0x20e	ERROR_CF_INVALID_SRK_ENTRY_KEYLEN	Key length specified in one of the entries in srk table is not one of the supported values (This is when srk_flag in header is 1)

Table 370. General CSF (ESBC) Header Checking Failures (C29x platforms)

Value	Code	Definition
0x301	ERROR_ESBC_HDR_LOC	ESBC header location is not in 3.5G space
0x302	ERROR_ESBC_HEADER_BARKER	Barker code in the header is incorrect.
0x303	ERROR_ESBC_HEADER_SG_ENTRIES_NOT_IN_3_5G	SG table/ESBC image address (header address + image offset in sg table) is beyond 3.5G
0x304	ERROR_ESBC_HEADER_SG_ESBC_EP	ESBC entry point in header not within ESBC address range
0x305	ERROR_SGL_ENTIRES_NOT_SUPPORTED	Number of entries in SG table exceeds maximum limit i.e 8
0x306	ERROR_ESBC_HEADER_HKAREA_LEN_ZERO	Houskeeping area not provided in header
0x307	ERROR_ESBC_HEADER_HKAREA_NOT_IN_3_5G	House keeping area not in 3.5G boundary
0x308	ERROR_ESBC_HEADER_HKAREA_LEN_INSUFFICIENT	Housekeeping area length provided is not sufficient.
0x309	ERROR_SG_TABLE_NOT_IN_3_5	SG Table is not in 3.5G boundary
0x310	ERROR_ESBC_HEADER_HKAREA_NOT_4K_ALIGNED	House keeping area is not aligned to 4K boundary
0x311	ERROR_SGL_ENTRIES_SIZE_ZERO	SG table has entry with size zero.

Table 371. Key/Signature/UID related errors (C29x platforms)

Value	Code	Definition
0x320	ERROR_ESBC_HEADER_KEY_LEN	Length of public key in header is not one of the supported values.
0x321	ERROR_ESBC_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key is not twice the length of the RSA signature

Table continues on the next page...

Table 371. Key/Signature/UID related errors (C29x platforms) (continued)

Value	Code	Definition
0x322	ERROR_ESBC_HEADER_KEY_MOD_1	Most significant bit of modulus in header is zero.
0x323	ERROR_ESBC_HEADER_KEY_MOD_2	Modulus in header is even number
0x324	ERROR_ESBC_HEADER_SIG_KEY_MOD	Signature value is greater than modulus in header
0x325	ERROR_FSL_UID	FSL_UID in ESBC Header did not match the FSL_UID in SFP if fsl uid flag Is 1
0x326	ERROR_OEM_UID	OEM_UID in ESBC Header did not match the OEM_UID in SFP if oem uid flag is 1.
0x327	ERROR_INVALID_SRK_NUM_ENTRY	Number of entries field in CSF Header is > 4(This is when srk_flag in header is 1)
0x328	ERROR_INVALID_KEY_NUM	Key number to be used from srk table is not present in table. (This is when srk_flag in header is 1)
0x329	ERROR_KEY_REVOKED	Key selected from srk table has been revoked(This is when srk_flag in header is 1)
0x32a	ERROR_INVALID_SRK_ENTRY_KEYLEN	Key length specified in one of the entries in srk table is not one of the supported values (This is when srk_flag in header is 1)
0x32b	ERROR_SRK_TBL_NOT_IN_3_5	SRK Table is not in 3.5G boundary (This is when srk_flag in header is 1)
0x32c	ERROR_KEY_NOT_IN_3_5G	Key is not in 3.5G boundary

Table 372. Verification Failures (C29x platforms)

Value	Code	Definition
0x340	ERROR_HASH_COMPARE_KEY	Super Root Key Hash Comparison failure. Mismatch in the hash of the public key/srk table as present in the header with the value in the SRK HASH fuse.
0x341	ERROR_HASH_COMPARE_EM	RSA signature check failure. Signature provided by you in the header doesn't match with the signature of the ESBC image generated by ISBC. The ESBC image loaded by you may be different than the image used while generating the signature(using CST)

Table 373. Driver errors (C29x platforms)

Value	Code	Definition
0x401	ERROR_NAND_READ_TIMEOUT	Timeout while reading from NAND device.
0x403	ERROR_NAND_FTOER	FTOER error in NAND EVTER during Boot Done

Table continues on the next page...

Table 373. Driver errors (C29x platforms) (continued)

Value	Code	Definition
0x404	ERROR_NAND_ECCER	ECCER error in NAND EVTER during Boot Done
0x405	ERROR_NAND_BBT_FULL	Bad Block Table is full
0x500	ERROR_ESDHC_CARD_DETECT_FAIL	eSDHC is unable to detect any SD/SDHC/MMC card.
0x501	ERROR_ESDHC_UNUSABLE_CARD	There is no response from the card on sending the command to get OCR value or the controller is unable to determine the card type.
0x502	ERROR_ESDHC_COMMUNICATION_ERROR	Error in setting the block size or reading a block.
0x503	ERROR_ESDHC_READ_UNALIGNED	The offset being read is not aligned with the block size.
0x600	ERROR_ESPI_READID	The Flash ID is invalid.
0x601	ERROR_ESPI_BOOT_SIGN	Unable to read Boot SIGN in both 24 bit and 16 bit mode.
0x602	ERROR_ESPI_READ_TIMEOUT	Timeout while reading data from SPI.

Table 374. SEC Failures (C29x platforms)

Value	Code	Definition
0x700	ERROR_SEC_ENQ	Error when enqueuing to SEC
0x701	ERROR_SEC_DEQ	Sec Block returned some error when dequeuing from it.
0x702	ERROR_SEC_DEQ_TO	Timeout when trying to deq from SEC
0x1	ERR_SELF_TEST_FAILED	Self Test executed on RNG returned Failure
0x10000	ERR_RNG_INSTANTIATE_FAILED	RNG Instantiation failed.

10.3.2.10 Address map used for the demo

The addresses below are effective addresses as mapped by u-boot.

P1010/ 9131/ 9132 /C290 platforms

Table 375. Memory map for P1010/ 9131/ 9132 /C290 platforms

Definition (Chain of trust)	Definition (Chain of trust with Confidentiality)	Address P1010	Address BSC9131	Address BSC9132	Address C290	Space reserved on Flash
cf_hdr.out	cf_hdr.out	0xee000000	0x8e000000	0x88000000	0xec000000	0x2000

Table continues on the next page...

Table 375. Memory map for P1010/ 9131/ 9132 /C290 platforms (continued)

Definition (Chain of trust)	Definition (Chain of trust with Confidentiality)	Address P1010	Address BSC9131	Address BSC9132	Address C290	Space reserved on Flash
esbc_hdr.out	esbc_hdr.out	0xee002000	0x8e002000	0x88002000	0xec002000	0x1E000
hdr_bs.out	hdr_bs.out	0xee020000	0x8e020000	0x88020000	0xec020000	0x2000
bootscript	bootscript	0xee022000	0x8e022000	0x88022000	0xec022000	0x1E000
hdr_dtb.out		0xee040000	0x8e040000	0x88040000	0xec040000	0x2000
DTB	DTB / DTB(encapsulated)	0xee042000	0x8e042000	0x88042000	0xec042000	0x3E000
hdr_linux.out		0xee080000	0x8e080000	0x88080000	0xec080000	0x2000
uImage	uImage / uImage(encapsulated)	0xee082000	0x8e082000	0x88082000	0xec082000	0x77E000
hdr_rootfs.out		0xee800000	0x8e800000	0x88800000	0xec800000	0x2000
rootfs	rootfs / rootfs(encapsulated)	0xee802000	0x8e802000	0x88802000	0xec802000	0x1EFE000
u-boot env	u-boot env	0xeff20000	0x8ff20000	0x8ff20000	0xeff20000	0x20000
u-boot.bin	u-boot.bin	0xeff40000	0x8ff40000	0x8ff40000	0xeff40000	0xC0000

NOTE

While running for chain of trust with confidentiality:

- **P1010**: Flash (DTB, uImage, rootfs) at above specified locations and then replace them with (DTB(encapsulated), uImage(encapsulated), rootfs(encapsulated)).
- **BSC9131**: Flash (DTB, uImage, rootfs) at above specified locations and then replace them with (DTB(encapsulated), uImage(encapsulated), rootfs(encapsulated)).
- **BSC9132**: There are two choices either Flash (DTB, uImage, rootfs) at above specified locations and then replace them with (DTB(encapsulated), uImage(encapsulated), rootfs(encapsulated)) or Flash (DTB(encapsulated), uImage(encapsulated), rootfs(encapsulated)) at location within **8B000000 - 8FF5FFFF** as this location is free for use.
- **C290**: There are two choices either Flash (DTB, uImage, rootfs) at above specified locations and then replace them with (DTB(encapsulated), uImage(encapsulated), rootfs(encapsulated)) or Flash (DTB(encapsulated), uImage(encapsulated), rootfs(encapsulated)) at location within **EE700000 - EFF5FFFF** as this location is free for use.
- To use 512KB u-boot change, address from eff40000 to eff80000

10.3.3 Service Processor (SP) Based Platforms

10.3.3.1 Secure Boot Introduction

There are three steps in the boot flow:

1. Service Processor Boot ROM

   SP provides pre-boot initialization and secure boot capabilities. The on-chip SP Boot ROM offers read-only, non-volatile storage for the Boot ROM code, including the internal secure boot code (ISBC) sub-routine for image validation. This Boot ROM is an integral part in the booting of the SOC in non-secure and secure boot modes.

2. GPP Boot ROM

   The on-chip GPP Boot ROM executes when the GPP cores are released from reset and is reponsible for passing control to next step in the boot flow i.e. the boot-loader validated by the SP.

3. Boot Loader

   The Boot Loader might further contain the External Secure Boot Code (ESBC) sub-routine for validation of next level images.

This document is intended for end-users to demonstrate the image validation process which happens in ISBC and ESBC.

The image validation can be split into stages, where each stage performs a specific function and validates the subsequent stage before passing control to that stage.

The Root of Trust is already established in the ISBC code residing in the Boot ROM which validates the Boot Loader 1.

Boot Loader 1 performs minimal SoC configuration before validating the Next Executable(s) which is/are known as ESBC image(s).

The flow includes validation of all ESBC images by a previously validated image before its execution to form a **Chain of TRUST**. The reference ESBC code also contains the functionality to form a **Chain of TRUST with confidentiality** where the next level images are kept on flash after encryption.

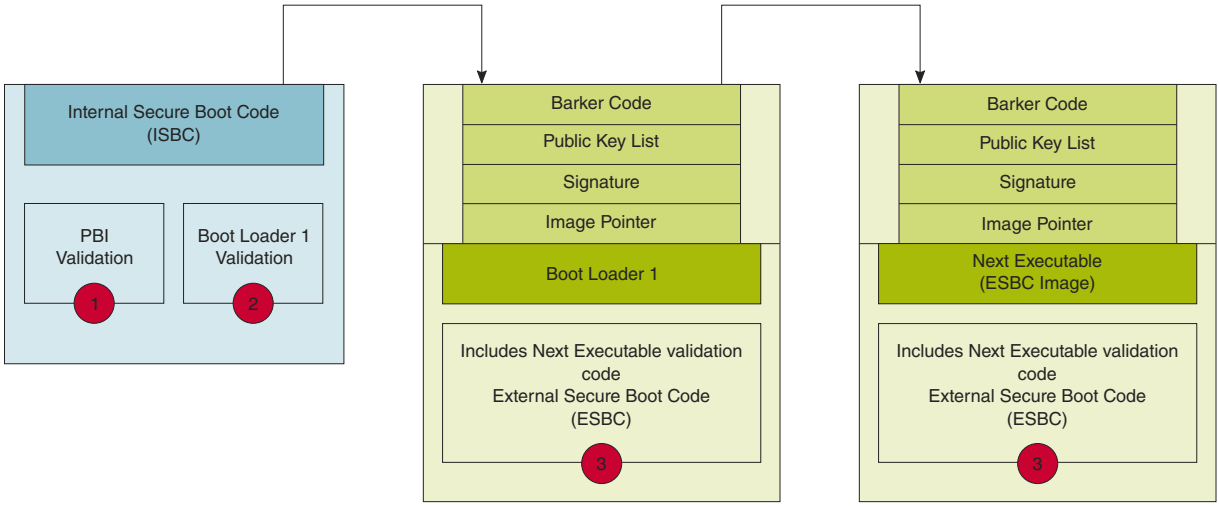


Figure 333. Chain of Trust

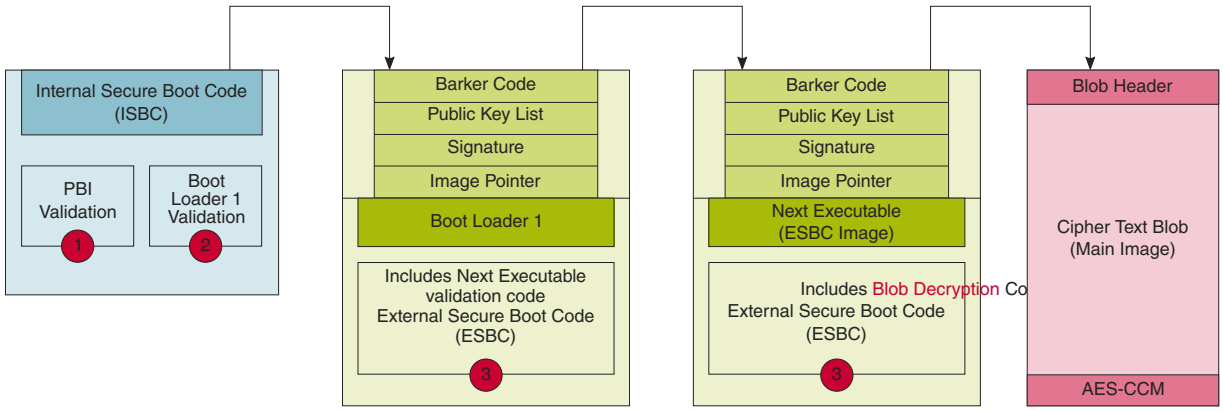


Figure 334. Chain of Trust with confidentiality

The validated ESBC image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Cryptographic blob mechanism is used to establish Chain of trust with confidentiality.

The above is explained in detail in coming sections.

As depicted in figure(s) above, there are three types of images which need to be validated as part of Secure Boot.

1. PBI image by ISBC
2. Boot Loader 1 image by ISBC
3. Next level image(s) by ESBC

Typically ESBC images would include:

Boot Loader 2 - n	In case Boot Loader is split in to multiple stages (Typical example is in case of NAND, SD Boot in which there is a mini-boot loader loaded on OCRAM which is Boot Loader stage 1 verified by ISBC. Boot Loader 2 is loaded on DDR, which must be validated by the Boot Loader 1.
MC/AIOP images	Management Complex images
LINUX	The operating system to be executed on the SoC.

10.3.3.1.1 Secure Boot process

Secure boot process uses a digital signature validation routine already present in ISBC residing in SP Boot ROM. This routine performs validation using HW bound RSA public key to decrypt the signed hash and compare it to a freshly calculated hash over the same system image. If the comparison passes, the image can be considered as authentic.

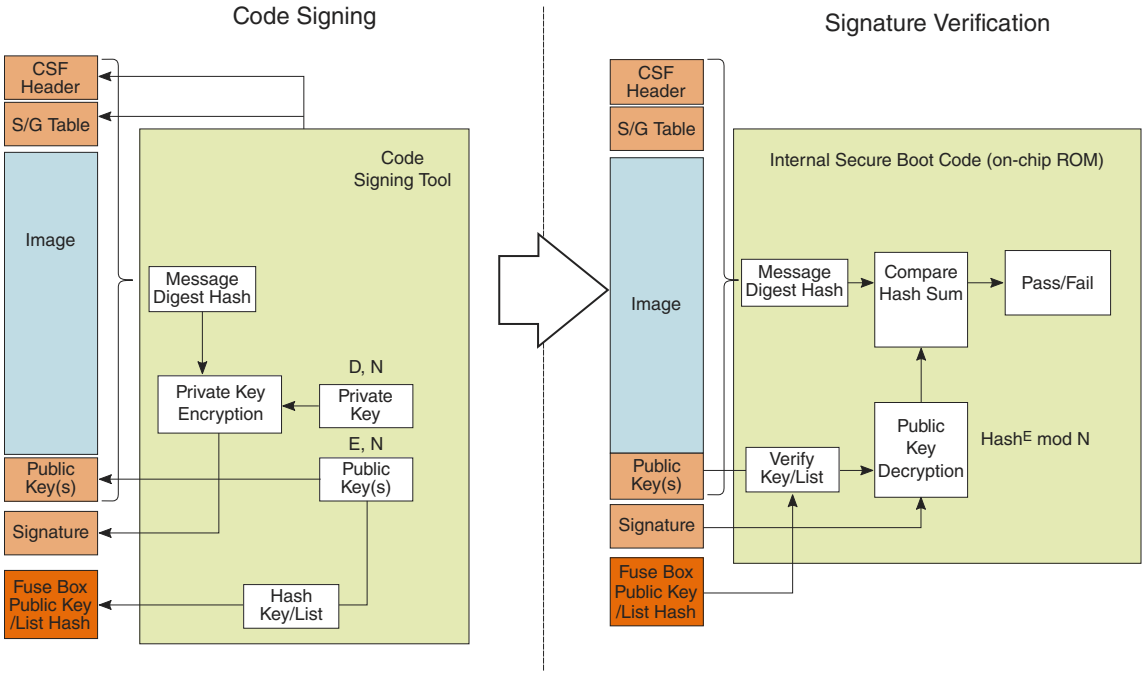


Figure 335. Secure Boot Process

As shown in the left side of the figure, the Code Signing Tool adds the following:

CSF Header	Command Sequence File Header This header provides the ISBC with flags, pointers, offsets, and lengths necessary to perform image validation.
S/G Table	Scatter Gather Table Optional (N/A for some stages which support only single image) Allows support for multiple non-contiguous images.
Public Key list	SRK (Super Root Key) Table One or more public keys is appended to the image. The CSF header indicates which of the keys is to be used in signature validation.
Signature	The SHA-256 hash of the CSF header + S/G table + Image + Public Key(s), encrypted with a RSA private key corresponding to one of the public keys in the key list.

As part of the code signing process, the CST also supports:

Generating RSA public and private key pairs	The RSA private key is exported for the OEM to store securely The CST also supports using public & private keys input by the OEM
Hashing the public key or public key list	This hash becomes the Super Root Key Hash which is stored in the SFP

At a high level, the secure boot process runs code signing in reverse.

1. The ISBC locates the CSF header and S/G table to further locate the image, public key list, and signature
2. The public key (list) is hashed and compared to the SRKH
3. If the public key is good, it is used to decrypt the signature (recover the hash)
4. The CSF header + S/G table + Image + Public key list are hashed, with the result compared to above. If the two hashes match, image is considered to be authentic.

10.3.3.1.1.1 Super Root Key (SRK)

These are RSA public and private key pairs. Private keys are used to sign the images and public keys are used to validate the image during ISBC and ESBC phase.

Public keys are embedded in the header and the hash of SRK table is fused in SRKH register of SFP.

These are Hardware Bound Keys, once the hash is fused the public private key pair can not be modified.

Keys of sizes 1k, 2k and 4k are supported in NXP Secure Boot Process.

It is the OEM's responsibility to tightly control access to the RSA private signature key. If this key is ever exposed, attackers will be able to generate alternate images that will pass secure boot.

If this key is ever lost, the OEM will be unable to update the image.

Key Revokation	Trust Architecture provides support for revoking the RSA public keys used by the ISBC to verify the ESBC. The RSA public keys used for this purpose are called super root keys. The SRK table supports maximum of 8 public keys and user has the option to revoke up to 7 keys. During secure boot, the ISBC checks the key number indicated in the CSF header against the revocation fuses in the SFP's OEM Security Policy Register (SFP_OSPR). If the key is revoked, the image validation fails.

10.3.3.2 ISBC Phase

At reset, Service Processor core is released and begins executing instructions from reset vector address 0x0 which is mapped to Internal Boot ROM. The Internal Boot ROM contains the code known as Internal Secure Boot Code (ISBC). The main steps in ISBC flow are defined below.

10.3.3.2.1 ISBC for PBI validation

1. **Sec_Mon check**: Confirms that the Sec_Mon is in the Check state. If not, it writes a 'fail' bit in a Sec_Mon control register, leading to a state transition.
2. **PBI command check**: Verify that the first PBI command is 'LOAD SEC HDR'. If not found, an error is raised.
3. **Valid header check**: Check for a valid preamble and correct B0/1 flag set as 0 in the header. If not, an error is raised.
4. **CSF parsing and public key check**; If ISBC finds a valid CSF header, it parses the CSF header to locate the public key from SRK (Super Root Key) table to be used to validate the code. There can be a table of maximum 8 public keys present in the header. The Secure Fuse Processor doesn't actually store a public key, it stores a SHA-256 hash of the table. If the hash of the SRK table fails to match the stored hash, secure boot fails.
5. **Signature validation**: With the validated public key, ISBC decrypts the digital signature stored with the CSF header. The ISBC then uses the PBI length field in the RCW to calculate a hash over all PBI commands (CSF header is also a part of PBI commands) along with the SRK table. Optional flags in the CSF header tell the ISBC whether the FSL Unique ID and the OEM Unique ID (in the Secure Fuse Processor) are to be checked or not. Including these IDs allows the image to be bound to a single platform. If the decrypted hash and generated hash don ' t match, secure boot fails.
6. **Sec_Mon Transition**: If ISS (Increment Security State) flag is set in the header, transition the SNVS state from Check to Trusted.

NOTE

1. PBI commands in Secure Boot must have a command '**Load Boot 1 CSF Header Ptr**' to inform the ISBC about location of CSF Header for BOOT1 image (ESBC).
2. Boot1 image and header must be placed on an XIP memory before execution of next phase (ISBC validation of Boot1/ESBC). If these images are placed on memories like NAND/SD/eMMC, then they must be copied to an XIP memory like OCRAM, DDR via PBI commands.

10.3.3.2.2 ISBC for Boot1 (Boot Loader 1) validation

1. **Valid header check**: Check for a valid preamble and correct B0/1 flag set as 1 in the header. If not, an error is raised.
2. **CSF parsing and public key check**: If ISBC finds a valid CSF header, it parses the CSF header to locate the public key from SRK (Super Root Key) table to be used to validate the code. There can be a table of maximum 8 public keys present in the header. The Secure Fuse Processor doesn't actually store a public key, it stores a SHA-256 hash of the table. If the hash of the SRK table fails to match the stored hash, secure boot fails.
3. **Signature validation**: With the validated public key, ISBC decrypts the digital signature stored with the CSF header. The image information is stored in a SG (scatter gather) table with support of up to 8 discrete images. The ISBC calculates a hash over the CSF header, SRK table, SG table and all entries in SG table (i.e. images). Optional flags in the CSF header tell the ISBC whether the FSL Unique ID and the OEM Unique ID (in the Secure Fuse Processor) are to be checked or not. Including these IDs allows the image to be bound to a single platform. If the decrypted hash and generated hash do not match, secure boot fails.
4. **Entry Point check**: One final check is performed by the ISBC. This check confirms that the Entry Point to be updated in Boot Location Pointer falls within one of the SG entries which have been validated by the ISBC.
5. **Sec_Mon Transition**: If ISS (Increment Security State) flag is set in the header, transition the SNVS state from Check to Trusted or Trusted (if transitioned in PB phase) to Secure.

NOTE

1. After End of ISBC, Entry Point parsed from header is written to Boot LOC PTR register.
2. GPP is waken up.
3. SP goes to sleep.

There are many reasons the ISBC could fail to validate the PBI or Boot1. Technicians with debug access can check the DCFG SCRATCHRW3 register to obtain an error code. For a list of error codes refer ISBC Validation Error Codes.

10.3.3.3 ESBC Phase

Unlike the ISBC, which is in an internal ROM and therefore unchangeable, the ESBC is reference code, and can be changed by OEMs. The remainder of this section is the description of a reasonable secure boot chain of trust based on NXP's reference software for secure boot. The reference ESBC code is also part of the Boot 1 image validated by ISBC and would be used to validate further ESBC images like MC, AIOP, and LINUX etc.

NXP provided ESBC consists of standard u-boot which has been signed using a private key. If the Boot Mode is secure, user can't reach to uboot prompt as the environment variable **bootdelay** is defined to 0.

There is default boot command for secure boot in the environment which executes on auto boot. This default **bootcmd** validates a file called boot script and on successful validation execute the commands in the boot script.

There are many reasons ESBC could fail to validate Client images or boot script. The error status message along with the code is printed on the u-boot console. For a list of error codes refer ESBC Validation Error Codes.

Users are free to use NXP ESBC as it is provided or to use it as reference to modify their own secure boot system.

To establish the Secure Boot Chain of Trust, some U-Boot Commands have been added in the ESBC Code which will be discussed in detail in coming sections.

10.3.3.3.1 esbc_validate command

```
esbc_validate img_hdr [pub_key_hash]
```

Input arguments:

img_hdr – Location of CSF header of the image to be validated

pub_key_hash – hash of the public key used to verify the image. This is optional parameter. If not provided, code makes the assumption that the key pair used to sign the image is same as that used with ISBC. So the hash of the key in the header is checked against the hash available in SRK fuse for verification.

Description:

The command would do the following:

Perform CSF header validation on the address passed in the image header. During parsing of the header, image address in stored in an environment variable which is later used in source command in default secure boot command.

Signature checks on the image.

10.3.3.3.2 esbc_halt command

```
esbc_halt (no arguments)
```

Description:

This command puts core in spin loop.

10.3.3.3.3 blob enc command

```
blob enc <src location> <dst location> <length> <key_modifier address>
```

Input Arguments:

src location	Address of the image to be encapsulated
dst location	Address where the blob will be created
length	Size of the image to be encapsulated
key_modifier address	Address where a random number 16 bytes long (key modifier) is placed

Description:

This command would create a cryptographic blob of the image placed at src location and place the blob at dst location.

10.3.3.3.4 blob dec command

```
blob dec <src location> <dst location> <length> <key_modifier address>
```

Input Arguments:

src location	Address of the image blob to be decapsulated
dst location	Address where the decapsulated image will be placed
length	Expected Size of the image after decapsulation
key_modifier address	Address where a random number 16 bytes long(key modifier) is placed

Description:

This command would decapsulate the blob placed at src location and place the decapsulated data of expected size at dst location.

10.3.3.3.5 Boot Script

Boot script is a U-Boot script image which contains u-boot commands. ESBC would validate this boot script before executing commands in it.

1. Boot script can have any commands which u-boot supports. No checking is done on the allowed commands in boot script. Since it is validated image, assumption is that commands in boot script would be correct.
2. If some basic scripting error is done in boot script like unknown command, missing arguments, the required usage of that command and core is put in infinite loop.
3. After execution of commands in boot script, if control reaches back in u-boot, error message would be printed on u-boot console and core would be put in spin loop by command esbc_halt.
4. Scatter gather images are not supported with validate command.
5. If ITS fuse is blown, any error in verification of the image would result in system reset. The error would be printed on console before system goes for a reset.

Where to place the boot script?

ESBC u-boot expects the boot script to be loaded in flash as specified in Address Map used for demo on page 2112. ESBC u-boot code assumes that the public/private key pair used to sign the boot script is same as that was used while signing the u-boot image. If user used different key pair to sign the image, hash of the N and E component of the key pair should be defined in macro:

```
CONFIG_BOOTSCRIPT_KEY_HASH
```

10.3.3.3.5.1 Chain of Trust

Boot script contains information about the next level images, e.g. MC, LINUX etc. ESBC validates these images as per their public keys. MC is started with validated MC images if required and finally bootm command is executed to pass control to LINUX image.

Users are free to use NXP ESBC as it is provided or to use it as reference to modify their own secure boot system. Figure below shows the Chain of Trust established for validation with this ESBC.

Sample Boot Script

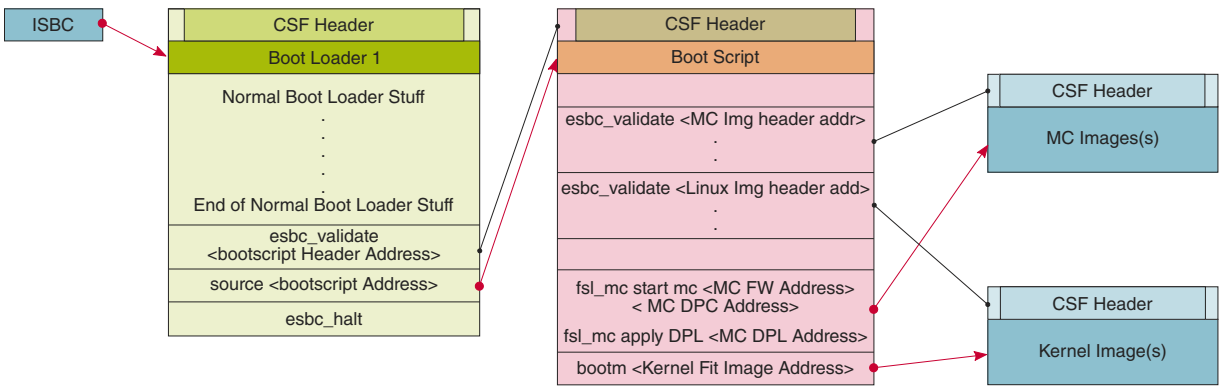


Figure 336. Secure Boot Flow (Chain of Trust)

```
# Get Images and Headers on DDR
.
.
.
# Validate the Images. (<pub_key_hash> is optional)
esbc_validate <Image1 Header Address> <pub_key_hash>
esbc_validate <Image2 Header Address> <pub_key_hash>
.
.
.
```

```
# Start MC with validated images
fsl_mc start mc <MC FW Address> < MC DPC Address>
fsl_mc apply DPL <MC DPL Address>

# Boot the Linux
bootm <Kernel Fit Image Address>
```

10.3.3.3.5.2 Chain of Trust with confidentiality

To establish chain of trust with confidentiality, cryptographic blob mechanism can be used. In this chain of trust, validated image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Two bootscripts are to be used. First encap bootscripts is used which creates a blob of the next level images(e.g. MC, LINUX etc.) and saves them on flash. After this the system is booted after replacing the encap bootscript with decap bootscript which decapsulates the blobs and start MC and LINUX.

Sample Encap Boot Script

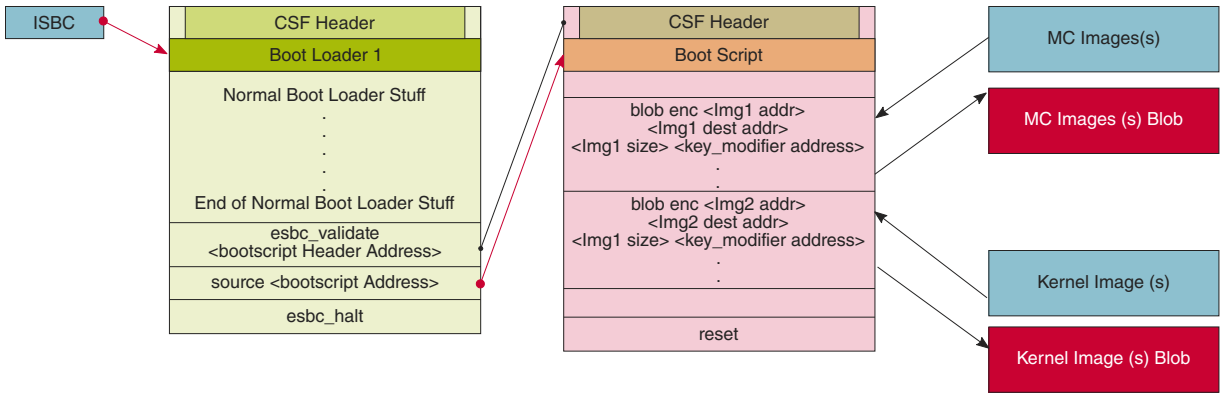


Figure 337. Chain of Trust with Confidentiality (Encapsulation)

```
# Get Images on DDR
.
.
.
# Create the Blobs
blob enc <Img1 addr> <Img1 dest addr> <Img1 size> <key_modifier address>
blob enc <Img2 addr> <Img2 dest addr> <Img2 size> <key_modifier address>
blob enc <Img3 addr> <Img3 dest addr> <Img3 size> <key_modifier address>
.
.
.

Save The Blobs created on Flash
.
.
.

# End of Encap Boot Script (This is one time only and must be replaced with decap Boot Script)
```

Sample Decap Boot Script

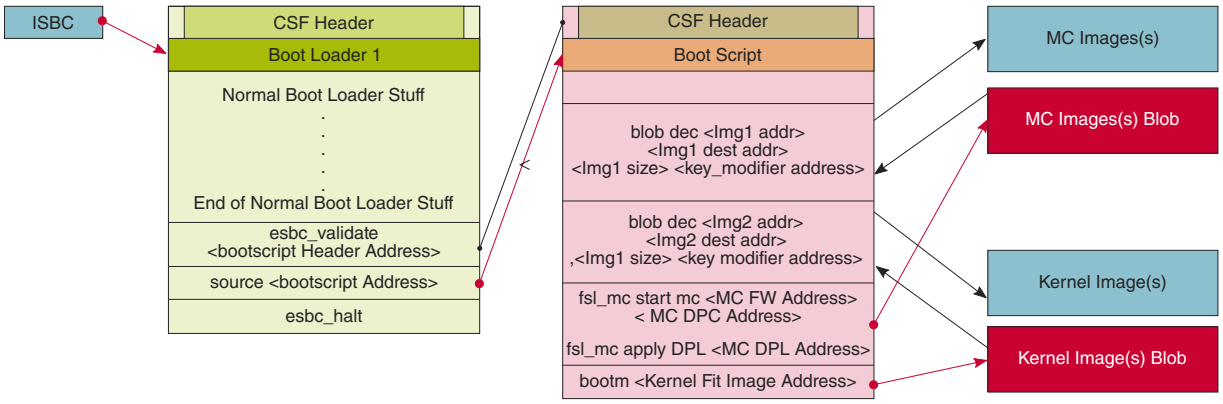


Figure 338. Chain of Trust with Confidentiality (Decapsulation)

```
# Get Images Blobs on DDR
.
.
.
# Decap the Blobs to get the actual images
blob dec <Img1 blob addr> <Img1 dest addr> <expected Img1 size> <key_modifier address>
blob dec <Img2 blob addr> <Img2 dest addr> <expected Img2 size> <key_modifier address>
blob dec <Img3 blob addr> <Img3 dest addr> <expected Img3 size> <key_modifier address>
.
.
.
# Start MC with validated images
fsl_mc start mc <MC FW Address> < MC DPC Address>
fsl_mc apply DPL <MC DPL Address>

# Boot the Linux
bootm <Kernel Fit Image Address>
```

10.3.3.4 Next Executable Phase

The bootloader (ESBC) finishes the platform initialization and passed control to the Linux image. The boot-chain can be further extended to be able to sign application which would be running on Linux prompt. Further RTIC can be integrated to verify memory regions using Security Engine (SEC) during run time.

10.3.3.5 ISBC Key Extension (IE)

The ISBC Key Extension feature allows the user to extend the number of keys available for signature validation. To use this feature, ISBC validates a table of **public** keys (IE keys) along with the ESBC image using one of the SRKs (Super Root Keys). These validated IE keys later become available for ESBC to use to validate downstream images e.g. Linux, MC etc.

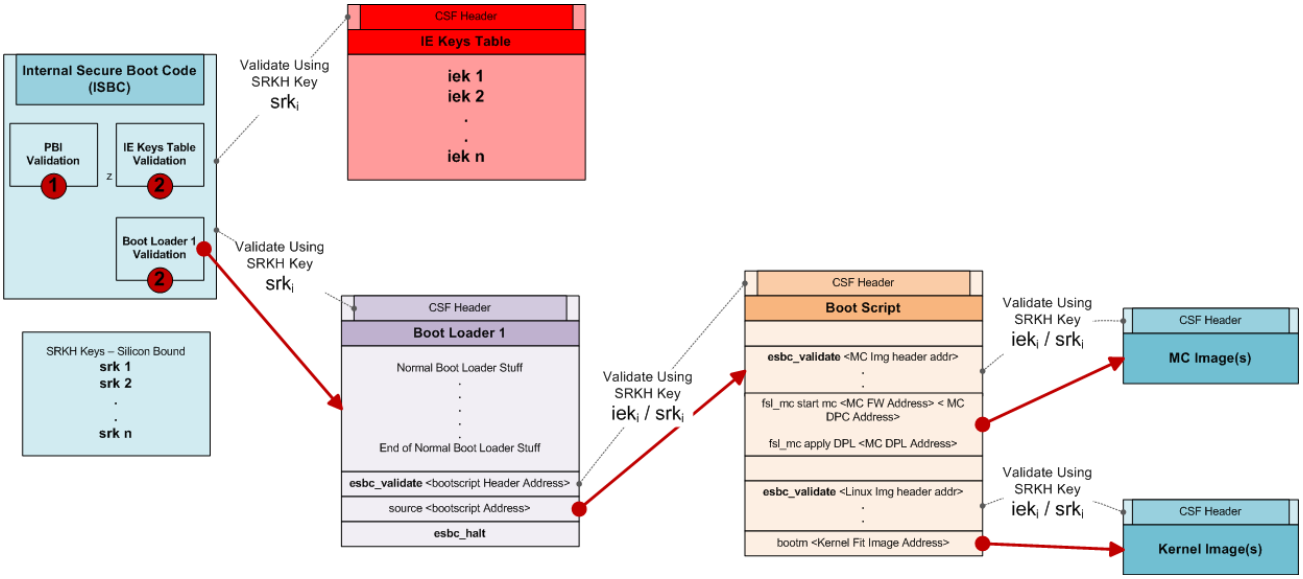


Figure 339. Secure Boot Flow (with IE)

As shown in the Secure Boot Flow (with IE),

1. ISBC code validates IE Table and Bootloader (ESBC) using one of the SRKs (Silicon Bound).
2. After IE Table is successfully validated, it becomes available for use by future images.[29](SRKs are present in SRK Table which is supplied with each image that is signed and validated using them. They are silicon bound as their hash, SRKH, is present in SoC's fuses.)
3. ESBC validates further images using either SRKs or IE Keys, which in turn can validate other images using these keys. Which key to use to verify image is decided based on CSF Header of the image to be validated.

The advantage of IE Keys is that

- Different images can be signed and validated using different IE keys. This is not possible with SRKs as they are significantly less in number than IE Keys.
- IE Key table can be changed as opposed to SRKs which are fixed as they are tied to SRKH blown in fuses.

10.3.3.5.1 IE Table Format

Table 376. Table: IE Table format

Offset (In Bytes)	Description
0x0 - 0x3	Keys Revoked. Each bit represents a key. If the bit is set, it indicates that this key in the table has been revoked and cannot be used for verification purposes. Bit 0 represents Key 1, Bit 1 represent Key 2 ..., Bit 31 represents Key 32.
0x04-0x07	Total number of keys present in the table (Max 32).
0x08-0x0b	Key 1 length (In bytes. Max 1024)

Table continues on the next page...

[29] For allowing ESBC image to use IE Table, it is neccesary for the IE Table to be present in an XIP memory. This is trivial in case of XIP memories such as NOR. In case on Non - XIP memories, IE table needs to be copied along with CSF Header to an XIP memory.

Table 376. Table: IE Table format (continued)

Offset (In Bytes)	Description
0x0c-0x40b	Key 1 value (Zero padded if less than 1024 bytes)
0x40c-0x40f	Key 2 length
0x410-0x80f	Key 2 value
.....	.....
0x8 + 0x404 * (n - 1) - 0x8 + 0x404 8 (n - 1) + 0x3	Key N length (N <= 32)
0x8 + 0x404 * (n - 1) + 0x4 - 0x8 + 0x404 * (n) - 0x1	Key N value

10.3.3.5.2 Enabling IE via CST Tool

The following are the steps that needs to be executed during **run-time** to enable use of IE feature during secure boot:

1. During PBI Phase, Scratch registers 13 and 14 (in DCFG Block) needs to be populated with the address of IE Table (to be loaded later). Scratch register 14 contains the higher 32 bits of address of IE Table and Scratch register 13 contains the lower 32 bits of address (SoC supports 48 bit address space).
2. Next, during the ISBC runtime, the IE table needs to be validated along with the ESBC image. For this, the address of this IE Table is added as the first entry in SG Table of CSF header.
3. Now ESBC can use either one of the IE Keys or the SRK Keys to validate next images. Which key to use, depends on the various field of CSF header (CSF Header Structure Definition on page 2096) of the image to be validated
   a. If IE Flag is not set, then one of the SRKs is used to verify the image. The key number is taken from the "Key No. for verification" flag. All SRKs would be present in the SRK Table, whose offset is embedded in CSF header.
   b. If IE Flag is set, then one of the IE Keys is used to verify the image. The key number is taken from the "IE Key Select" flag. The key is gathered from IE Table, the address of which was already populated in Scratch registers 13 and 14.

The following are the steps that needs to be executed during **build-time** to incorporate IE Table in CSF header and sign images using IE Keys using CST tool :

1. Using uni_sign, sign ESBC image and IE Table (uni_sign also generates IE Table using IE keys). uni_sign also prints the IE Table address.
2. Using uni_pbi generate PBI commands which also includes the commands to write the address of IE Table given by uni_sign to Scratch registers 13 and 14.
3. Sign next level images using one of the IE Keys / SRKs.

10.3.3.5.2.1 uni_sign on ESBC

Sample Input File for uni_sign

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
```

```
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
-------------------------------------------------
# Entry Point/Image start address field in the header.[Mandatory]
# (default=ADDRESS of first file specified in images)
# Address can be 64 bit
ENTRY_POINT=30100000
-------------------------------------------------
# Specify the Key Information.
# PUB_KEY [Mandatory] Comma Seperated List
# Usage: <srk1.pub> <srk2.pub> .....
PUB_KEY=srk.pub
# KEY_SELECT [Mandatory]
# USAGE (for TRUST 3.x): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=srk.pri
# IE_KEY [Mandatory] Comma Seperated List
#IE_REVOC [Optional] Comma Seperated List
IE_KEY=iekey1.pub,iekey2.pub,iekey3.pub
#ESBC_HDRADDR [Mandatory] 32bit Address of ESBC Header
#Used to Calculate IE Table Address
ESBC_HDRADDR=30020000
-------------------------------------------------
# Specify IMAGE, Max 8 images are possible.
# DST_ADDR is required only for Non-PBL Platform. [Mandatory]
# USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
# Address can be 64 bit
IMAGE_1={u-boot-dtb.bin,30100000,ffffffff}
IMAGE_2={,,}
IMAGE_3={,,}
IMAGE_4={,,}
IMAGE_5={,,}
IMAGE_6={,,}
IMAGE_7={,,}
IMAGE_8={,,}
-------------------------------------------------
# Specify OEM AND FSL ID to be populated in header. [Optional]
# e.g FSL_UID_0=11111111
FSL_UID_0=
FSL_UID_1=
OEM_UID_0=
OEM_UID_1=
OEM_UID_2=
OEM_UID_3=
OEM_UID_4=
-------------------------------------------------
# Specify the output file names [Optional].
# Default Values chosen in Tool
OUTPUT_HDR_FILENAME=hdr_uboot.out
IMAGE_HASH_FILENAME=
RSA_SIGN_FILENAME=
-------------------------------------------------
# Specify The Flags. (0 or 1) - [Optional]
MP_FLAG=0
ISS_FLAG=1
LW_FLAG=0
-------------------------------------------------
```

```
# Specify VERBOSE as 1, if you want to Display Header Information [Optional]
VERBOSE=0
```

Table 377. Description of Fields in Input File

Field in Input File	Description
IE_KEY	Comma seperated list of names of files containing public keys (IE Keys)
IE_REVOC	Comma seperated list of key number that are to be revoked from IE Table
ESBC_HDRADDR	32 Bit address where generated header shall be placed

The above file is an example of input file for running uni_sign on ESBC image to enable use of IE Table. Presence of field 'IE_KEY' indicates to the CST tool to embed IE Table in the output header file. Field 'ESBC_HDRADDR' is needed to calculate address of IE Table. It states the address at which the output of uni_sign (output header file) will be placed on memory map. 'PUB_KEY=srk.pub' and 'PRI_KEY=srk.pri' indicates that SRK Key needs to be used to sign both ESBC and IE Table.

Note: Here ESBC_HDRADDR needs to be 32 bit address as ISBC can only access 32 bit addresses.

Output of uni_sign

```
$ ./uni_sign --verbose input_files/uni_sign/<platform>/ie_keys/input_uboot_nor_secure


    #-----------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #-----------------------------------------------------#

============================================================
This tool includes software developed by OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
============================================================

Input File is input_files/uni_sign/<platform>/ie_keys/input_uboot_nor_secure

-----------------------------------------------
-    Dumping the Header Fields
-----------------------------------------------
- SRK Information
-     SRK Offset : 200
-     Number of Keys : 1
-     Key Select : 1
-     Key List :
-        Key1 srk.pub(100)
- UID Information
-     UID Flags = 00
-     FSL UID = 00000000_00000000
-     OEM UID0 = 00000000
-     OEM UID1 = 00000000
-     OEM UID2 = 00000000
-     OEM UID3 = 00000000
-     OEM UID4 = 00000000
- FLAGS Information
-     MISC Flags = 60
-         ISS = 1
```

```
-         MP = 0
-         LW = 0
-         B01 = 1
- Image Information
-     SG Table Offset : 800
-     Number of entries : 2
-     Entry Point : 30100000
-     Entry 1 : ie_table.out (Size = 00000c14 SRC = 30020a00 DST = ffffffff)
-     Entry 2 : u-boot-dtb.bin (Size = 00090c16 SRC = 30100000 DST = ffffffff)
- RSA Signature Information
-     RSA Offset : 1800
-     RSA Size : 80
----------------------------------------------

Image Hash:
fa2a5b7ba63375f18c94f2911687750e0e49c27eaf0c313617ed3ebfc006c6e6
IE Table Absolute Address: 30020a00


**********************************************
* Header File is with Signature appended
**********************************************

Header File Created: hdr_uboot.out

SRK (Public Key) Hash:
c9f90c9762e1693804421a4bd8193735b38ea03b83303d9529f7b1b5d2e4688f
     SFP SRKHR0 = c9f90c97
     SFP SRKHR1 = 62e16938
     SFP SRKHR2 = 04421a4b
     SFP SRKHR3 = d8193735
     SFP SRKHR4 = b38ea03b
     SFP SRKHR5 = 83303d95
     SFP SRKHR6 = 29f7b1b5
     SFP SRKHR7 = d2e4688f
```

As shown, the output prints the absolute address of IE Table. This address is equal to 'ESBC_HDRADDR + offset of IE Table within output header file' as IE Table is embedded in the output header file by uni_sign.

Note: uni_sign needs to be executed with --verbose option to get the 'IE Table Absolute Address'

Partial view of hdr_uboot.out generated by uni_sign (In Hex)

```
......
......
0000800: 140c 0000 0000 0000 000a 0230 ffff ffff  ...........0....
0000810: 160c 0900 0000 0000 0000 1030 ffff ffff  ...........0....
......
......
0000a00: 0000 0000 0300 0000 0001 0000 cf9d 8fd0  ................
0000a10: 57a4 c7be e519 6ae1 b4db 2e97 8c79 9d3f  W.....j......y.?
0000a20: 61a2 5538 f1af 58c0 31cf 3484 a54d 343a  a.U8..X.1.4..M4:
0000a30: 58ec 02ec e5e2 153f 9843 b0e0 7c0b f9b2  X......?.C..|...
0000a40: 76fa f87d 8780 98bd cf87 6079 3b34 87ad  v..}......`y;4..
0000a50: c9d3 6fe9 71fd 884d f531 be13 2d83 dfaa  ..o.q..M.1..-...
0000a60: 0ca1 0f9f 8ae1 5312 457a 251d 3fd2 0127  ......S.Ez%.?..'
0000a70: de51 3f3e 2da7 ae17 b203 42db a495 6c7a  .Q?>-.....B...lz
0000a80: d2b9 5671 bb77 b32d 2a62 9045 0000 0000  ..Vq.w.-*b.E....
```

```
......
......
```

As shown above, at offset 0x800 in the hdr_uboot.out, SG Table is present. There are two SG Entries present there. The first entry is the IE Table (at address 0x30020a00). The second entry is the u-boot image. Also, at 0xa00 offset of hdr_uboot.out, the IE Table is present with 3 Keys.

10.3.3.5.2.2 uni_pbi

Sample Input File for uni_pbi

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

--------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
--------------------------------------------------
# Specify the Key Information.
# PUB_KEY [Mandatory] Comma Seperated List
# Usage: <srk1.pub> <srk2.pub> .....
PUB_KEY=srk.pub
# KEY_SELECT [Mandatory]
# USAGE (for TRUST 3.x): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=srk.pri
#IE_TABLE_ADDR [optional] Used by Uboot to get IE Table Address
#Can be 64 bit
IE_TABLE_ADDR=580020a00
--------------------------------------------------
# For PBI Signing
# Name of RCW + PBI file [Mandatory]
RCW_PBI_FILENAME= rcw.bin
# Address of ISBC (Boot1) CSF Header [Mandatory]
BOOT1_PTR=30020000
--------------------------------------------------
# Specify OEM AND FSL ID to be populated in header. [Optional]
# e.g FSL_UID_0=11111111
FSL_UID_0=
FSL_UID_1=
OEM_UID_0=
OEM_UID_1=
OEM_UID_2=
OEM_UID_3=
OEM_UID_4=
--------------------------------------------------
# Specify the output file names [Optional].
# Default Values chosen in Tool
OUTPUT_HDR_FILENAME=rcw_sec.bin
IMAGE_HASH_FILENAME=
RSA_SIGN_FILENAME=
--------------------------------------------------
# Specify The Flags. (0 or 1) - [Optional]
MP_FLAG=0
ISS_FLAG=1
```

```
LW_FLAG=0
-------------------------------------------------
# Specify VERBOSE as 1, if you want to Display Header Information [Optional]
VERBOSE=0
```

Table 378. Description of Fields in Input File

Field in Input File	Description
IE_TABLE_ADDR	64 bit address of IE Table

As shown above, in the input file to uni_pbi, the field 'IE_TABLE_ADDR' is needed. This field indicates the address of IE Table, the address which is written to Scratch registers 13 and 14.

Note: This address (0x580020a00) is different from the address generated from uni_sign (0x30020a00) (though both address maps to same address. Refer to the memory map of the respective SoC). This is because ISBC can only access 32 bit addresses and u-boot uses only 48 bit addresses for NOR mapping because u-boot doesn't make MMU Table entries for the 32 bit addresses of NOR. So uni_sign needs 32 bit address so that ISBC can access this address to validate IE Table and uni_pbi needs 48 bit address so that u-boot can access the address written in the Scratch register 13 and 14 to fetch the keys in IE Table.

Output of uni_pbi

```
$ ./uni_pbi input_files/uni_pbi/<platform>/ie_keys/input_pbi_nor_secure


    #-----------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #-----------------------------------------------------#

============================================================
This tool includes software developed by OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
============================================================

Input File is input_files/uni_pbi/<platform>/ie_keys/input_pbi_nor_secure


***********************************************
* Header File is with Signature appended
***********************************************

Header File Created: rcw_sec.bin

SRK (Public Key) Hash:
c9f90c9762e1693804421a4bd8193735b38ea03b83303d9529f7b1b5d2e4688f
     SFP SRKHR0 = c9f90c97
     SFP SRKHR1 = 62e16938
     SFP SRKHR2 = 04421a4b
     SFP SRKHR3 = d8193735
     SFP SRKHR4 = b38ea03b
     SFP SRKHR5 = 83303d95
     SFP SRKHR6 = 29f7b1b5
     SFP SRKHR7 = d2e4688f
```

Partial view of rcw_sec.bin generated by uni_pbi (In Hex)

```
0000000: 55aa 55aa 0000 1080 2038 3038 3800 3838  U.U..... 8088.88
0000010: 0000 0000 0000 0000 0000 0000 0000 2000  .............. .
0000020: 0000 2000 0000 0000 802d e102 8025 0000  .. ......-...%..
0000030: 0000 0000 0000 0000 0b0e 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0070 0200 0000 0000 0000 2a41 0000 0000  .p........*A....
0000080: 0000 0000 0000 0000 b8b3 1bdf 0000 2080  .............. .
0000090: 1219 2001 0002 0000 0101 0020 0000 0000  .. ........ ....
00000a0: 0008 0000 8000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 2280 0000 0230 3002 e030 000a 0280   .."....00..0....
00000f0: 3402 e030 0500 0000 0404 e030 0000 0000  4..0.......0....
0000100: 0004 e030 0000 1030 0000 0833 0000 0000  ...0...0...3....
0000110: 0005 0833 d704 0000 2006 e030 0000 00a0  ...3.... ..0....
.....
.....
```

As shown above, the highlighted part are the PBI commands inserted in the rcw_sec.bin to write IE Table address (64 bits) to Scratch Registers 13 and 14.

10.3.3.5.2.3 uni_sign on next level images

Sample Input File for uni_sign for signing Linux

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

ESBC=1
-------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
-------------------------------------------------
# Specify the Key Information.
# PUB_KEY [Mandatory] Comma Seperated List
# Usage: <srk1.pub> <srk2.pub> .....
PUB_KEY=iekey3.pub
# IE_KEY_SEL [Mandatory]
# USAGE (for TRUST 3.x): (between 1 to 32)
IE_KEY_SEL=3
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=iekey3.pri
-------------------------------------------------
# Specify IMAGE, Max 8 images are possible.
# DST_ADDR is required only for Non-PBL Platform. [Mandatory]
# USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
# Address can be 64 bit
IMAGE_1={kernel.itb,a1100000,ffffffff}
```

```
-------------------------------------------------
# Specify OEM AND FSL ID to be populated in header. [Optional]
# e.g FSL_UID_0=11111111
FSL_UID_0=
FSL_UID_1=
OEM_UID_0=
OEM_UID_1=
OEM_UID_2=
OEM_UID_3=
OEM_UID_4=
-------------------------------------------------
# Specify the output file names [Optional].
# Default Values chosen in Tool
OUTPUT_HDR_FILENAME=hdr_kernel.out
IMAGE_HASH_FILENAME=
RSA_SIGN_FILENAME=
-------------------------------------------------
# Specify VERBOSE as 1, if you want to Display Header Information [Optional]
VERBOSE=0
```

Table 379. Description of Fields in Input File

Field in Input File	Description
IE_KEY_SEL	Number of the key in IE Table that is to be used to validate image

As shown above, in the input file to uni_sign, the field 'IE_KEY_SEL' indicates that a Key from IE Table is being used and it also indicates which key number from IE Table needs to be used to validate this image during run-time. For build-time, CST tool needs the IE private key. This is indicated via the field 'PRI_KEY'.

Output of uni_sign

```
$ ./uni_sign input_files/uni_sign/<platform>/ie_keys/input_kernel_secure


    #-------------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #-------------------------------------------------------#

============================================================
This tool includes software developed by OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
============================================================

Input File is input_files/uni_sign/<platform>/ie_keys/input_kernel_secure

SRK/Public Key Hash not calculated.. IE = 1

***********************************************
* Header File is with Signature appended
***********************************************

Header File Created: hdr_kernel.out
SRK (Public Key) Hash Not Available
```

As shown above, SRK Hash is not calculated as IE Key is being used.

10.3.3.6 Product Execution

This section presents the steps to be followed in order to properly run the software product according to its intended use and functionalities.

Steps in the demo would be:

1. ISBC code would validate PBI and Boot Loader 1.
2. On Successful validation, PBI commands would be executed by SP BootROM.
3. Boot Loader 1 execution will begin on GPP.
4. The ESBC code in Boot Loader 1 would validate and execute and bootscript.[30]
5. The boot script would contain the commands to validate and execute next level images as described in Boot Script on page 2071Boot Script on page 2071.

10.3.3.6.1 Environment for Secure Boot

There are 2 ways in which secure boot can be initiated:

1. Set SB_EN bit in RCW to 1.
2. Programming the ITS fuse in SFP.

In a manufacturing environment, it is recommended that all fuses be programmed at once, including the ITS and OEM Section Write Protect bits. In a prototyping environment, it may be preferable to leave ITS and Write Protect unprogrammed (relying on RCW to initiate secure boot) until the developer has confidence in the secure boot process.

10.3.3.6.2 SDK images required for demo

Given Below are the images required which are built as part of the SDK Release:

1. RCW along with PBI commands
2. Boot Loader Image(s[31])
3. Kernel Fit Image
4. MC/AIOP image(s)[32]

10.3.3.6.3 CST tool for signing images

CST tool used for signing the images is provided as a package with yocto and is built for host. It can be run from your host machine.

Install path for CST binaries in yocto:

```
tmp/sysroots/x86_64-linux/usr/bin/cst/
```

In the Yocto environment, the user needs to use below commands to rebuild CST:

1. Modify the CST source code if needed
2. bitbake cst-native –c cleanall
3. bitbake cst-native

[30] In case the boot loader is split into two parts, the validation and execution of boot script would happen in the final boot loader i.e Boot Loader2. Boot Loader 1 will validate and transfer control to Boot Loader 2.
[31] In Case the Boot Loader is split into parts.
[32] MC/AIOP loading is optional and can be skipped.

NOTE

1. Sample Input Files are provided in the CST package for signing the various images and creating the headers as per the memory map described in Address Map used for demo on page 2112
2. The input files also contain the names of the images to be signed and names of output header file. User must place the images with the same name as specified in input file in the CST directory.
3. If a custom name/address is desired, then the same must be specified in the input file before executing the CST tool.

Table 380. CST Input File Information

Image	CST Tool	Image Name in Input File	Output Name in Input File
RCW with PBI commands	uni_pbi	rcw.bin	rcw_sec.bin
U-Boot	uni_sign (ESBC=0)	u-boot-dtb.bin	hdr_uboot.out
BootScript	uni_sign (ESBC=1)	bootscript	hdr_bs.out
Linux Kernel Image	uni_sign (ESBC=1)	kernel.itb	hdr_kernel.out
Management complex Firmware(MC)	uni_sign (ESBC=1)	mc.itb	hdr_mc.out
MC DPL	uni_sign (ESBC=1)	dpl.itb	hdr_dpl.out
MC DPC	uni_sign (ESBC=1)	dpc.dtb	hdr_dpc.out
PPA	uni_sign (ESBC=1)	ppa.itb	hdr_ppa.out

10.3.3.6.4 Chain of Trust

This section presents the steps needed to be followed in order to execute Chain of Trust.

10.3.3.6.4.1 Other images required for demo

Apart from SDK images described above, the following images are also required:

1. rcw_sec.bin (Contains the CSF Header for PBI commands. This is created using **uni_pbi** CST Tool)
2. CSF header for Boot Loader 1. (Created using **uni_sign** CST tool with **ESBC = 0**)
3. CSF header of Kernel Image (Created using **uni_sign** CST tool with **ESBC = 1**)
4. Boot Script
5. CSF header of Boot Script (Created using **uni_sign** CST tool with **ESBC = 1**)
6. Optional: (Created using **uni_sign** CST tool with **ESBC = 1**)
   a. CSF header(s) for MC/AIOP image(s).
   b. CSF header(s) for Boot Loader 2..n.

10.3.3.6.4.2 Boot Script and signing images

CSF header needs to be generated for all the images. More details on the commands provided by CST can be found in Code Signing Tool on page 2116

Signing images using same key pair

1. Generate the public/private key pair to be used for signing the image

```
./gen_keys 1024
```

   Key pair - public key file - srk.pub and private key in srk.priv would be generated.

2. Copy all the required images in the CST directory with the names as specified in the Input Files (Table 380. CST Input File Information on page 2085)
3. Create CSF header for PBI and U-Boot Image.

```
./uni_pbi input_files/uni_pbi/<platform>/input_pbi_nor_secure
```

```
./uni_sign input_files/uni_sign/<platform>/input_uboot_nor_secure
```

   Executing the above command would also output a 256 bit hash of the SRK Table. This has to be programmed in the SRKF fuses in SFP. The hash can also be obtained later on by using the command

```
./uni_sign --hash <input_file>
```

NOTE

a. The steps in demo are described for NOR Boot. The input files chosen above have the corresponding address from NOR Flash.

b. For the ESBC images, the images are first copied to DDR and then validated. Thus the input files and corresponding output headers would have addresses of DDR and the input files are independent of the boot source.

4. Create CSF header for Linux Kernel Image.

```
./uni_sign input_files/uni_sign/<platform>/input_kernel_secure
```

5. Create header for PPA using PPA input file as ./uni_sign input_files/uni_sign/<platform>/input_ppa_secure
6. Create CSF header(s) for MC Image(s) if required.

```
./uni_sign input_files/uni_sign/<platform>/input_mc_secure
./uni_sign input_files/uni_sign/<platform>/input_dpl_secure
./uni_sign input_files/uni_sign/<platform>/input_dpc_secure
```

7. Create the Boot Script (Chain of Trust on page 2072) using the following steps:

   a. Create a text file bootscript.txt with following commands:

NOTE

- The commands to get the images on DDR would change if the headers and images are placed on some other device like NAND or SD.
- The other commands for validation and booting remain same irresepctive of the device.
- The address and size of image(s) are as specified in Address Map used for demo on page 2112

```
#Get All ESBC Headers on DDR
#LINUX Kernel Header
cp.b 0x580d80000 0xa0d80000 0x4000;
#DPL Header
cp.b 0x580d40000 0xa0d40000 0x4000;
```

```
#Get All ESBC Images on DDR
#LINUX Kernel Image
cp.b 0x581100000 0xa1100000 0x2800000;
#DPL Image
cp.b 0x580700000 0xa0700000 0x100000;
```

```
###Validating DPL###
esbc_validate 0xa0d40000;
setenv dpl_img $img_addr;

fsl_mc apply DPL $dpl_img

###Validate and Boot Kernal Image###
echo "Validating Kernel"
esbc_validate 0xa0d80000
bootm $img_addr;
```

b. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript.txt
bootscript
```

8. Create CSF header for Boot Script.

```
./uni_sign input_files/uni_sign/<platform>/input_bootscript_secure
```

Signing images using different key pair

U-Boot and PBI must be signed with the same key pair as they are both validated by ISBC. For all ESBC images a different key pair can be used if desired.

NOTE

If boot script is also signed with a different key, remember to define the macro "CONFIG_BOOTSCRIPT_KEY_HASH" with the hash of the key used to sign the boot script in file:

```
include/config_fsl_chain_trust.h
```

1. Generate the public/private key pair to be used for signing the image

```
./gen_keys 1024 -p isbc.priv -k isbc.pub
```

Key pair - public key file - isbc.pub and private key in isbc.priv would be generated.

2. Copy all the required images in the CST directory with the names as specified in the Input Files (Table 380. CST Input File Information on page 2085)
3. Create CSF header for PBI and U-Boot Image.

Change PRI_KEY and PUB_KEY to isbc.priv and isbc.pub respectively in the input files.

```
./uni_pbi input_files/uni_pbi/<platform>/input_pbi_nor_secure
```

```
./uni_sign input_files/uni_sign/<platform>/input_kernel_secure
```

Executing the above command would also output a 256 bit hash of the SRK table. This has to be programmed in the SRKF fuses in SFP. The hash can also be obtained later on by using the command

```
./uni_sign --hash <input_file>
```

NOTE

a. The steps in demo are described for NOR boot. The input files chosen above have the corresponding address from NOR flash.

b. For the ESBC images, the images are first copied to DDR and then validated. Thus the input files and corresponding output headers would have addresses of DDR and the input files are independent of the boot source.

4. Repeat step 1 to generate more key pair(s) for other ESBC images

```
./gen_keys 1024 -p bootscript.priv -k bootscript.pub
./gen_keys 1024 -p kernel.priv -k kernel.pub
./gen_keys 1024 -p mc.priv -k mc.pub
./gen_keys 1024 -p dpc.priv -k dpc.pub
./gen_keys 1024 -p dpl.priv -k dpl.pub
```

5. Create CSF header for Linux Kernel Image.

   Change PRI_KEY and PUB_KEY to kernel.priv and kernel.pub respectively in the input files.

```
./uni_sign input_files/uni_sign/<platform>/input_kernel_secure
```

   The key hash obtained is <kernel_key_hash>

6. Create header for PPA using PPA input file as ./uni_sign input_files/uni_sign/<platform>/input_ppa_secure

7. Create CSF header(s) for MC image(s) if required.

   Change PRI_KEY and PUB_KEY to corresponding key names in the input files.

```
./uni_sign input_files/uni_sign/<platform>/input_mc_secure
./uni_sign input_files/uni_sign/<platform>/input_dpl_secure
./uni_sign input_files/uni_sign/<platform>/input_dpc_secure
```

   The key hash(s) obtained will be <mc_key_hash>, <dpl_key_hash> and <dpc_key_hash>

8. Create the Boot Script (Chain of Trust on page 2072) using the following steps:

   a. Create a text file bootscript.txt with following commands:

NOTE

- The commands to get the images on DDR would change if the headers and images are placed on some other device like NAND or SD.
- The other commands for validation and booting remain same irresepctive of the device.
- The address and size of image(s) are as specified in Address Map used for demo on page 2112

```
#Get All ESBC Headers on DDR
#LINUX Kernel Header
cp.b 0x580d80000 0xa0d80000 0x4000;
#DPL Header
cp.b 0x580d40000 0xa0d40000 0x4000;
```

```
#Get All ESBC Images on DDR
#LINUX Kernel Image
cp.b 0x581100000 0xa1100000 0x2800000;
#DPL Image
cp.b 0x580700000 0xa0700000 0x400000;
```

```
###Validating DPL###
echo "Validating DPL"

esbc_validate 0xa0d40000 <dpl_key_hash>
setenv dpl_img $img_addr;

fsl_mc apply DPL $dpl_img

###Validate and Boot Kernal Image###
echo "Validating Kernel"
esbc_validate 0xa0d80000 <kernel_key_hash>
bootm $img_addr;
```

b. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript.txt
bootscript
```

9. Create CSF header for Boot Script.

   Change PRI_KEY and PUB_KEY to bootscript.priv and bootscript.pub respectively in the input files.

```
./uni_sign input_files/uni_sign/<platform>/input_bootscript_secure
```

10.3.3.6.4.3 Running Secure Boot (Chain of Trust)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
   a. **Flow A**

      Program the ITS fuse.

   b. **Flow B**

      For protyping phase, don't blow the ITS fuse, Secure Boot can be enabled by RCW with SB_EN = 1.
2. Blow other required fuses(OTPMK and SRK Hash[33]) in the SFP in silicon. For more details regarding fuse blowing, CCS and Reset Pause, refer to *Platform Reference Manual* and *Trust Architecture User Guide*.

[33] Blowing of OTPMK is essential to run secure boot for both Production (Flow A) and Prototyping/ Development (Flow B).

For SRK Hash,in Development Mode (Flow B), there is a workaround to avoid blowing fuses. The SoC can be put in a **Reset Pause** state. This will pause the Reset State Machinery after RCW Loading. Then CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then get the system out of Reset Pause State.

NOTE

SRK hash in the fuse should be same as the hash of the key pair being used to sign the PBI and U-Boot image.

For testing purpose, the SRK hash can be written in the mirror registers.

gen_otpmk_drbg utility in CST can be used to generate otpmk key.

3. Flash all the generated images at locations as described in the address map(Address Map used for demo on page 2112).
   a. **Flow A** - All the images would have to be flashed at the current bank addresses.
   b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.

NOTE

The U-Boot commands to flash the images as per the address map and CCS info are described in Useful Commands on page 2113.

4. Give a power on cycle to the board.
   a. For Flow A and Flow B (If Secure Boot images flashed on default bank)
      - On power on, ISBC code in SP Boot ROM would validate the PBI image followed by validation of Boot Loader1 (U-Boot)
      - ESBC code in Boot Loader 1 image would further validate the ESBC images (BootScript, LINUX, MC etc.)
      - MC and LINUX would be started.
   b. For Flow B (If Secure Boot images flashed on alternate bank), the user must first do the switch settings[34] for booting from alternate bank and also to enable reset pause.
      - On power on after the correct switch setting, Reset State Machinery will be paused after RCW loading.
      - Write the SRKH to SFP mirror registers and get the system out of Reset Pause via CCS.
      - Secure Boot flow as mentioned above would execute.

10.3.3.6.5 Chain of Trust with confidentiality

This section presents the steps to be followed in order to execute chain of trust with confidentiality.

The demo would be divided into two parts:

1. Creating /encrypting images in form of blobs.
2. Decrypting the images, and booting from decrypted images.

The execution steps remain same as specified above in Product Execution on page 2084. In first phase the Boot Script would contain the commands to encrypt and create blobs of the images. After that the Boot Script is replaced and in second phase the Boot Script would contain commands to decrypt the blobs to get back the images and boot LINUX, AIOP using these images.

10.3.3.6.5.1 Other images required for demo

Apart from SDK images described above, the following images are also required:

1. rcw_sec.bin (contains the CSF header for PBI commands. This is created using **uni_pbi** CST tool)
2. CSF header for Boot Loader 1. (Created using **uni_sign** CST tool with **ESBC = 0**)
3. Encap Boot Script

[34] This may also be done via writing to FPGA registers from the U-Boot Prompt of U-Boot runing in Non-Secure Mode on Bank0. Please refer the Platform FPGA guide for the same.

4. Decap Boot Script
5. CSF header of Encap Boot Script (Created using **uni_sign** CST tool with **ESBC = 1**)
6. CSF header of Decap Boot Script (Created using **uni_sign** CST tool with **ESBC = 1**)
7. **[Optional]** CSF header(s) for Boot Loader 2..n.(Created using **uni_sign** CST tool with **ESBC = 1**)

10.3.3.6.5.2 Boot Script and signing images

NOTE

- The commands to get the images on DDR would change if the images are placed on some other device like NAND or SD.
- The other commands for encryption, decryption and booting remain same irresepctive of the device.
- The address and size of image(s) are as specified in Address Map used for demo on page 2112

Encap Boot Script

1. Create a bootscript_en.txt file with following commands:
2. For device specific bootscript_en.txt refer in section (LS1088: Secure Boot Demo) Address Map used for demo on page 2112

```
#Sample Encap Bootscript
#Makes blob of Linux, MC,DPL,DPC images and replace original images
#with blobs
#Addr for DRAM to use temporarily
setenv dram_temp "0x80000000"
setenv dram_temp2 "0xa0000000"
#Write Key
mw 0x90000000 0x1234567
mw 0x90000004 0x1234567
mw 0x90000008 0x1234567
mw 0x9000000c 0x1234567
#Key Identifier Addr
setenv key_id_addr "0x90000000"
#Encap MC
cp.b 0x580300000 $dram_temp2  0x3c0000
blob enc $dram_temp2 $dram_temp 0x3c0000 $key_id_addr
erase 0x580300000 +0x400000
cp.b $dram_temp 0x580300000 0x400000
#Encap DPL
cp.b 0x580700000 $dram_temp2  0xc0000
blob enc $dram_temp2 $dram_temp 0xc0000 $key_id_addr
erase 0x580700000 +0x100000
cp.b $dram_temp 0x580700000 0x100000
#Encap DPC
cp.b 0x580800000 $dram_temp2  0xc0000
blob enc $dram_temp2 $dram_temp 0xc0000 $key_id_addr
erase 0x580800000 +0x100000
cp.b $dram_temp 0x580800000 0x100000
#Encap Kernel
cp.b 0x581100000 $dram_temp2  0x27c0000
blob enc $dram_temp2 $dram_temp 0x27c0000 $key_id_addr
erase 0x581100000 +0x2800000
cp.b $dram_temp 0x581100000 0x2800000
```

3. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript_en.txt
bootscript_encap
```

Decap Boot Script

1. Create a bootscript_de.txt file with following commands:
2. For device specific bootscript_de.txt refer to section (LS1088: Secure Boot Demo) Address Map used for demo on page 2112

```
# Sample Decap Boot Script
#Decrypt blob of Linux, MC,DPL,DPC images to get original images
#Write Key - Same as key for Enc
mw 0x90000000 0x1234567
mw 0x90000004 0x1234567
mw 0x90000008 0x1234567
mw 0x9000000c 0x1234567
#Key Identifier Addr
setenv key_id_addr "0x90000000"
#Addr for DRAM to use temporarily
setenv dram_temp "0x80000000"
#Decrypt MC
cp.b 0x580300000 $dram_temp 0x400000
blob dec  $dram_temp 0xa0300000 0x3c0000 $key_id_addr
#Decrypt DPL
cp.b 0x580700000 $dram_temp 0x100000
blob dec $dram_temp 0xa0700000 0xc0000 $key_id_addr
#Decrypt DPC
cp.b 0x580800000 $dram_temp 0x100000
blob dec $dram_temp 0xa0800000 0xc0000 $key_id_addr
#Decrypt Kernel
cp.b 0x581100000 $dram_temp 0x2800000
blob dec $dram_temp 0xa1100000 0x27c0000 $key_id_addr
#Start MC
fsl_mc start mc 0xa0300000 0xa0800000;
fsl_mc apply DPL 0xa0700000
#Start Kernel
bootm 0xa1100000;
```

3. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript_de.txt
bootscript_decap
```

Signing the images

1. Generate the public/private key pair to be used for signing the image

```
./gen_keys 1024
```

   Key pair - public key file - srk.pub and private key in srk.priv would be generated.

2. Copy all the required images in the CST directory with the names a specified in the Input Files Address Map used for demo on page 2112

3. Create CSF header for PBI and U-Boot Image.

```
./uni_pbi input_files/uni_pbi/<platform>/<device>[35]/input_pbi_nor_secure
```

```
./uni_sign input_files/uni_sign/<platform>/<device>[36]/input_uboot_secure
```

Executing the above command would also output a 256 bit hash of the SRK table. This has to be programmed in the SRKF fuses in SFP. The hash can also be obtained later on by using the command

```
./uni_sign --hash <input_file>
```

4. Create CSF headers for both the Boot Scripts

Change the image name and output header name in the input file for the two bootscripts.

```
./uni_sign input_files/uni_sign/<platform>/<device>/input_bootscript_secure
```

10.3.3.6.5.3 Running Secure Boot (Chain of Trust with confidentiality)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
   a. **Flow A**

      Program the ITS fuse.

   b. **Flow B**

      For protyping phase, don't blow the ITS fuse, Secure Boot can be enabled by RCW with SB_EN = 1.

2. Blow other required fuses(OTPMK and SRK hash[37]) in the SFP in silicon. For more details regarding fuse blowing, CCS and Reset Pause, refer to Platform Reference Manual and Trust Architecture User Guide.

> **NOTE**
>
> SRK hash in the fuse should be same as the hash of the key pair being used to sign the PBI and U-Boot image.
>
> For testing purpose, the SRK hash can be written in the mirror registers.
>
> gen_otpmk_drbg utility in CST can be used to generate otpmk key.

3. Flash all the generated images at locations as described in the address map (Address Map used for demo on page 2112).
   - rcw_sec.bin
   - U-Boot binary
   - CSF header for U-Boot
   - Kernel fit image

---

[35] Device folder is not required in case of NOR device, for others use device name eg. qspi.

[36] Device folder is not required in case of NOR device, for others use device name eg. qspi.

[37] Blowing of OTPMK is essential to run secure boot for both Production (Flow A) and Prototyping/ Development (Flow B).

For SRK Hash,in Development Mode (Flow B), there is a workaround to avoid blowing fuses. The SoC can be put in a **Reset Pause** state. This will pause the Reset State Machinery after RCW Loading. Then CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then get the system out of Reset Pause State.

- MC, DPL, DPC images
- bootscript_encap
- CSF header for bootscript_encap

a. **Flow A** - All the images would have to be flashed at the current bank addresses.

b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.

NOTE

The U-Boot commands to flash the images as per the address map and CCS info are described in Useful Commands on page 2113.

4. Give a power on cycle to the board.
   a. For Flow A and Flow B (If Secure Boot images flashed on default bank)
      - On power on, ISBC code in SP Boot ROM would validate the PBI image followed by validation of Boot Loader1 (U-Boot)
      - ESBC code in Boot Loader 1 image would further validate the Boot Script.
      - Boot Script would encapsulate the Kernel and MC images and store the blobs at the desired locations.
   b. For Flow B (If Secure Boot images flashed on alternate bank), the user must first do the switch settings[38] for booting from alternate bank and also to enable Reset Pause.
      - On power on after the correct switch setting, Reset State Machinery will be paused after RCW loading.
      - Write the SRKH to SFP mirror registers and get the system out of reset pause via CCS.
      - Secure Boot flow as mentioned above would execute.
5. Replace the encap bootscript and its CSF header with decap bootscript and the CSF header of the decap bootscript respectively.
6. Give a power on cycle to the board.
   a. For Flow A and Flow B (If Secure Boot images flashed on default bank)
      - On power on, ISBC code in SP Boot ROM would validate the PBI image followed by validation of Boot Loader1 (U-Boot)
      - ESBC code in Boot Loader 1 image would further validate the Boot Script.
      - Boot Script would decapsulate the Kernel and MC blobs and store the images on DDR.
      - MC and LINUX Kernel would be started.
   b. For Flow B (If Secure Boot images flashed on alternate bank), the user must first do the switch settings[39] for booting from alternate bank and also to enable Reset Pause.
      - On power on after the correct switch setting, Reset State Machinery will be paused after RCW Loading.
      - Write the SRKH to SFP mirror registers and get the system out of reset pause via CCS.
      - Secure Boot flow as mentioned above would execute.

[38] This may also be done via writing to FPGA registers from the U-Boot Prompt of U-Boot runing in Non-Secure Mode on Bank0. Please refer the Platform FPGA guide for the same.
[39] This may also be done via writing to FPGA registers from the U-Boot Prompt of U-Boot runing in Non-Secure Mode on Bank0. Please refer the Platform FPGA guide for the same.

10.3.3.7 PBI Structure

	Fields	Offset	Size (In 32-bit word)
RCW	Preamble (RCW)	0x00	1
	Load RCW Command	0x04	1
	RCW words	0x08 – 0x87	32
	RCW Checksum	0x88	1
PBI Commands	Load Security Header CSF Header	0x8c 0x90 – 0xdf	1 20
	Load Boot 1CSF Header Boot 1 pointer	0xe0 0xe4	1 1
	Other PBI Commands	0xe8	N
	STOP Command (With/ Without CRC)	0xe8 + (4*N)	2
SRK Table	SRK Table	0x90 + SRK table offset in CSF Header	(No. of Keys * Key length)
RSA signature	Signature	0x90 + Sign offset in CSF Header	Sign Length

RCW

Preamble The preamble is always the first element in a PBI Image. It contains a standard pattern that identifies the memory location as the beginning of a valid PBI Image. The preamble is a 4-byte pattern defined as “0xaa55aa55”.

Load RCW Command The next word is load RCW command. This command loads the 1024 bit Reset Configuration Word from the interface specified by Power-on-Reset (POR) Configuration strapping pins. It has two formats:

1. Load RCW with Checksum (0x10): Read Reset Configuration Word performs simple 32-bit checksum, and update RCW registers.
2. Load RCW without Checksum (0x11): Read Reset Configuration Word and update RCW registers without performing checksum. The version without the checksum includes padding with zeroes in the place of the checksum value.

RCW Words 1024 RCW bits i.e. 32 words of 32 bit.

RCW Checksum It is calculated as a 32-bit unsigned integer summation of the RCW Preamble, the Load RCW with checksum command, and each of the 32 words (32-bit) of the RCW. A simple 32-bit checksum is used for the validation of the command.

```
checksum(RCW_WORD[]){
    unsigned_32 sum = 0xAA55AA55 + 0x80100000 + Load RCW Command;
    for(i=0; i<32; i++)
        sum+=RCW_WORD[i];
```

```
    return (sum);
}
```

NOTE

Checksum will have to be updated by CST tool as the fields like SB_EN, PBI_LEN in the RCW words are changed.

PBI Commands	**Load Security Header**	This command loads information required for authentication of the PBI image. The security header includes pointers to an SRK key table and RSA signatures as well as other flags and IDs. The CSF Header is part of the command. Please refer the CSF header structure in .
	Load Boot 1 CSF Header	This command loads a pointer to a CSF Header used for authentication of the Boot 1 Secondary Program Loader. This 32-bit value used by the Boot 0 ISBC and is required for secure boot.
	Other PBI commands	Other PBI commands input by user.
	STOP Command	This command ends the PBI sequence and has two variants (with and without CRC). The CRC check value covers all commands from the first command after the RCW up to and including this CRC and Stop command, regardless of whether any are skipped by Jump commands during execution. In Stop command without CRC it ends the PBI sequence immediately. It does not include a CRC value, but it instead has a 32-bit padding with zeroes so that it is the same size as the Stop with CRC command.

NOTE

CST tool updates the PBI commands by adding Load Security Header command and Load Boot 1 Security Header command. So, CRC must also be updated.

SRK Table	Table of Public Keys used in Secure Boot Validation. It is kept at an offset from CSF Header. The offset is specified in the CSF Header.
RSA Signature	RSA signature calculated over all PBI commands and SRK Table. It is kept at an offset from CSF Header. The offset is specified in the CSF Header.

10.3.3.8 CSF Header Structure Definition

Table 381. Trust Architecture and SFP Information

SoC	Trust Arch. Version	SFP Version	POVDD	DRVR		OTPMK	
				Algo (CST)	Register to check Hamming Error	Algo (CST)	Register to check Hamming Error
LS1088A	3.1	3.5	1.89 V	A2	SFP Secret Value Hamming	2	SFP Secret Value Hamming

Table continues on the next page...

Table 381. Trust Architecture and SFP Information (continued)

LS2088A (LS2 Rev2)	3.1	3.5	1.89 V	A2	Error Status Register (SFP_SVHESR)	2	Error Status Register (SFP_SVHESR)

Figure 340. CSF Header Structure

Table 382. CSF Header Structure (ISBC Trust 3.0)

Offset	Description
0x00	**Barker Code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)

Table continues on the next page...

Table 382. CSF Header Structure (ISBC Trust 3.0) (continued)

Offset		Description
0x04		**SRK table offset** This location contains an address which is the offset of the SRK table from the start of CSF header. Using this offset and the number of entries is SRK table, the SRK table is read. * Description of fields in SRK Table is mentioned below.
0x08	0x08	**No. of keys** This field specifies the no. of keys in the SRK table
	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** Reserved **MP :** Execute Manufacturing Protection Routine **ISS :** Increment Security State; indicates whether the ISBC should increment the SNVS SSM upon successful verification **B01 :** Identifies whether this is the CSF header of a boot 0 image (PBI) or a boot 1 image (SPL) **LW :** Leave Writeable; when set; ISBC doesn't set the SFP Write Disable
0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved
	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x14		**RSA signature length** This location contains the length of the RSA Signature in bytes.
0x18		**SG table offset** This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries is SG Table, the SG table is read. * Description of fields in SG table is mentioned below/.

Table continues on the next page...

Table 382. CSF Header Structure (ISBC Trust 3.0) (continued)

Offset	Description
0x1C	**No. of entries** This field specifies the number of entries present in SG table.
0x20	**Entry point (32 bit)** ISBC transfers control to this location upon successful validation of ESBC image(s).
0x24	FSL UID 0
0x28	FSL UID 1
0x2c	OEM UID 0
0x30	OEM UID 1
0x34	OEM UID 2
0x38	OEM UID 3
0x3c	OEM UID 4
0x40	Reserved
0x44	Reserved
0x48	Reserved
0x4C	Reserved

Table 383. CSF Header Structure (ISBC Trust 3.1)

Offset		Description
0x00		**Barker Code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)
0x04		**SRK table offset** This location contains an address which is the offset of the SRK table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.
0x08	0x08	**No. of keys** This field specifies the no. of keys in the SRK Table

Table continues on the next page...

Table 383. CSF Header Structure (ISBC Trust 3.1) (continued)

	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** ISBC Extension (Reserved) **MP :** Execute Manufacturing Protection Routine **ISS :** Increment Security State; indicates whether the ISBC should increment the SNVS SSM upon successful verification **B01 :** identifies whether this is the CSF header of a boot 0 image (PBI) or a boot 1 image (SPL) **LW :** Leave Writeable; when set; ISBC doesn't set the SFP Write Disable
0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved
	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x14		**RSA signature length** This location contains the length of the RSA Signature in bytes.
0x18		**SG table offset** This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries is SG Table, the SG table is read.
0x1C		**No. of entries** This field specifies the number of entries present in SG table.
0x20		**Entry point (64 bit)** ISBC transfers control to this location upon successful validation of ESBC image(s).
0x28		FSL UID 0
0x2c		FSL UID 1
0x30		OEM UID 0

Table continues on the next page...

Table 383. CSF Header Structure (ISBC Trust 3.1) (continued)

0x34	OEM UID 1
0x38	OEM UID 2
0x3c	OEM UID 3
0x40	OEM UID 4
0x44	Reserved
0x48	Reserved
0x4C	Reserved

Table 384. CSF Header Structure (ESBC Trust 3.0 and Trust 3.1)

Offset		Description
0x00		**Barker Code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)
0x04		**SRK table offset** This location contains an address which is the offset of the SRK table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.
0x08	0x08	**No. of keys** This field specifies the no. of keys in the SRK Table
	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** ISBC Extension Flag
0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved

Table continues on the next page...

Table 384. CSF Header Structure (ESBC Trust 3.0 and Trust 3.1) (continued)

	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x14		**RSA signature length** This location contains the length of the RSA signature in bytes.
0x18		**Image address (64 bit)**
0x20		**Image Size**
0x24		IE Key Select
0x28		FSL UID 0
0x2c		FSL UID 1
0x30		OEM UID 0
0x34		OEM UID 1
0x38		OEM UID 2
0x3c		OEM UID 3
0x40		OEM UID 4
0x44		Reserved
0x48		Reserved
0x4c		Reserved

Table 385. SRK Table Structure

Offset	Description
0x00	SRK 0 Length (Length of Modulus or Exponent; Mod length always equals Exp length)
0x04	SRK 0 Value (Modulus)
0x04 + K	SRK 0 Value (Exponent)
0x04 + 2K	SRK 0 (Padding; 8Kb - (Exp+Mod))
0x04 *1 + (10 * 1)K	SRK 1 Length (Length of Modulus or Exponent; Mod length always equals Exp length)
0x04 * 2 + (10 *1) K	SRK 1 Value (Modulus)

Table continues on the next page...

Table 385. SRK Table Structure (continued)

Offset	Description
0x04 * 2+ (10 * 1) + 1k	SRK 1 Value (Exponent)
0x04 * 2 + (10 * 1) + 2K	SRK 1 (Padding; 8Kb - (Exp+Mod))
0x04 * (N-1) + (10 *(N-1))K	SRK N Length (Length of Modulus or Exponent; Mod length always equals Exp length)
0x04 * N + (10 *(N-1))K	SRK N Value (Modulus)
0x04 * N + (10 *(N-1)) + 1K	SRK N Value (Exponent)
0x04 * N + (10 *(N-1)) + 2K	SRK N (Padding; 8Kb - (Exp+Mod))

Table 386. SG Table Structure

Offset	Description
0x00	Length
0x04	Reserved
0x08	SRC Address Low
0x0C	SRC Address High

10.3.3.9 Secure Boot Specific RCW Fields

This section describes the various fields in RCW which are relevant to the ISBC code executed in the Service Processor Boot ROM.

SB_EN Secure Boot Enable

Bit(s): 266

- 0 Secure Boot is not enabled[40]
- 1 Secure Boot is enabled

PBI_LENGTH Pre-Boot Initialization Length

Bit(s): 287-276

Size in words of the PBI commands.

SDBGEN Secure Debug Enable

Bit(s): 288

Secure Debug (CoreSight SPIDEN) is enabled after RCW is loaded if this RCW bit is set and the 'Intent to Secure' fuse value is cleared.

- 0 Secure debug is not enabled
- 1 Secure debug is enabled if the ITS fuse is not burned to asserted

GPIO_LED_EN GPIO LED Enable

Bit(s): 311

If the OEM chooses to implement a LED to indicate secure boot failure, the LED will be connected to a GPIO. The SP Boot ROM code sequence turns on the LED (if RCW[GPIO_LED_EN] = 1) by configuring

[40] Secure Boot is enabled if either this RCW bit is set or the Intent to Secure fuse value is set.

one GPIO direction (GPDIR) register bit as an output and writing the corresponding output in a GPIO block data (GPDAT) register.

GPIO_LED_NUM GPIO Number for LED

Bit(s): 310-304

If GPIO_LED_EN is set, these bits specify the GPIO number to which LED is connected.

- 0x1f - 0x00 : GPIO_1
- 0x3f - 0x20 : GPIO_2
- 0x5f - 0x40 : GPIO_3
- 0x7f - 0x60 : GPIO_4

NOTE

The GPIO output assigned to the LED is driven high to whatever VDD voltage is supplied by the integrated device for the chosen GPIO output.Since GPIO pins at the time of SoC reset are initially configured as inputs, and since there will be some indeterminate period of time from the assertion of SoC reset to when the GPIO pin is configured by SP Boot ROM as an output, the GPIO pin chosen must be terminated with a weak pulldown to ground.

10.3.3.10 ISBC Error Codes

Error Handling in Production Environemnt (ITS = 1)

- Error Code would be logged in DCFG SCARTCH register.
- SNVS would be transitioned to **soft fail** state.
- Activate the LED. If the OEM chooses to implement a LED to indicate secure boot failure, the LED will be connected to a GPIO. The information of GPIO is specified via bits in RCW.

  GPIO_LED_EN Bit(s): 311

  The SP Boot ROM code sequence turns on the LED (if RCW[GPIO_LED_EN] = 1) by configuring one GPIO direction (GPDIR) register bit as an output and writing the corresponding output in a GPIO block data (GPDAT) register.

  GPIO_LED_NUM Bit(s): 310-304

  If GPIO_LED_EN is set, these bits specify the GPIO number to which LED is connected.

  - 0x1f - 0x00 : GPIO_1
  - 0x3f - 0x20 : GPIO_2
  - 0x5f - 0x40 : GPIO_3
  - 0x7f - 0x60 : GPIO_4

- Soft Reset would be issued
- Cores would then enter infinite loop (If Reset is disabled)[41]

Error Handling in Development Environemnt (ITS = 0, SB_EN = 1)

- Error Code would be logged in DCFG SCARTCH register.
- SNVS would be transitioned to **non-secure** state.

[41] To debug the root cause of failure and view the error code, Reset has to be disabled on the SoC.

- Further actions depends on the type of failure

Fatal Errors Core is put in infinite Loop

Non-Fatal Error Application Software is allowed to execute

Error Codes

The Error codes reported by SP Boot ROM can be categorized in following sections:

1. Core Exceptions
2. Device Errors
3. RCW/PBI Errors
4. Validation Errors

Table 387. ISBC Error Codes

When Error Generated	Error Code	Value	Description
Core Exceptions			
Random	ERROR_UNDEFINED_INSTRUCTION	0x1	Occurs if neither the processor nor any attached co-processor recognizes the currently executing instruction.
Random	ERROR_SWI	0x2	Software Interrupt is a user-defined interrupt instruction. It allows a program running in User mode, for example, to request privileged operations that run in Supervisor mode.
Random	ERROR_PREFETCH_ABORT	0x3	Occurs when the processor attempts to execute an instruction that has been prefetched from an illegal address.
Random	ERROR_DATA_ABORT	0x4	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
Random	ERROR_IRQ	0x5	Occurs when the processor external interrupt request pin is asserted (LOW) and IRQ interrupts are enabled.
Random	ERROR_FIQ	0x6	Occurs when the processor external fast interrupt request pin is asserted (LOW) and FIQ interrupts are enabled.
Device Errors – I2C			
Random	ERROR_I2C_TIMEOUT	0x11	
Random	ERROR_I2C_RESTART	0x12	
Random	ERROR_I2C_NODEV	0x13	
Random	ERROR_I2C_NOT_IDLE	0x14	
Random	ERROR_I2C_NOT_BUSY	0x15	
Random	ERROR_I2C_INVALID_OFFSET	0x16	

Table continues on the next page...

Table 387. ISBC Error Codes (continued)

When Error Generated	Error Code	Value	Description
Random	ERROR_I2C_NO_WAKEUP_INIT	0x17	
Random	ERROR_I2C_NO_WAKEUP_READ	0x18	
Random	ERROR_I2C_NOACK	0x19	
Random	ERROR_READ_TIMEOUT	0x1a	
Random	ERROR_SLAVE_ADDR_TIMEOUT	0x1b	
Random	ERROR_MEM_ADDR_TIMEOUT	0x1c	
Device Errors – ESDHC			
Random	ERROR_ESDHC_CARD_DETECT_FAIL	0x31	
Random	ERROR_ESDHC_UNUSABLE_CARD	0x32	
Random	ERROR_ESDHC_COMMUNICATION_ERROR	0x33	
Random	ERROR_ESDHC_BLOCK_LENGTH	0x34	
Device Errors – QSPI			
Random	ERROR_QSPI_INVALID_OFFSET	0x41	
Phase – "**RCW**"			
RCW Phase	ERROR_PREAMBLE	0x50	Preamble not found.
RCW Phase	ERROR_RCW_CMD_NOT_FOUND	0x51	RCW command not found
RCW Phase	ERROR_RCW_CHECKSUM_MISMATCH	0x52	Checksum mismatch in RCW
RCW Phase	ERROR_RCW_SRC_INVALID	0x58	RCW_SRC is not a valid source
RCW Phase	ERROR_RCW_REQ_NOT_SET	0x59	RCW_REQ bit never set by Reset state machine (RSM)
RCW Phase	ERROR_PBI_REQ_NOT_SET	0x60	PBI_REQ bit never set (by RSM)
Phase = **PBI**			
PBI Phase	ERROR_SEC_CAAM_INIT	0x61	CAAM init failed (Would rarely occur)
PBI Phase	ERROR_SEC_CAAM_NOT_FOUND	0x62	CAAM block not found in case of secure boot
PBI Phase	ERROR_PBI_SRC_NOT_SAME_AS_RCW_SRC	0x64	Mismatch between RCW_SRC and PBI_SRC fields
PBI Phase	ERROR_PBI_LENGTH	0x65	PBI Length defined in RCW[PBI_LEN] field is invalid
PBI Phase	ERROR_PBI_LAST _CMD_NOT_STOP	0x66	STOP or CRC&STOP not found at the end in the specified PBI Length.

Table continues on the next page...

Table 387. ISBC Error Codes (continued)

When Error Generated	Error Code	Value	Description
PBI Phase	ERROR_PBI_ COMMAND_UNKNOWN	0x67	An invalid command parsed by PBI Parser
PBI Phase	ERROR_CAAM_SELF_TEST	0x6a	CAAM self-test failed
PBI Phase	ERROR_PBI_ COPY_INVALID_ SRC_TYPE	0x70	Copy command, src field doesn't match the RCW_SRC field
PBI Phase	ERROR_PBI_ COPY_INVALID_ DST_ADDR	0x71	Copy command, dest field isn't 0x00
PBI Phase	ERROR_PBI_ COPY_INVALID_SRC_ADDR_ SRC_ADDR	0x72	SRC Address is invalid (ROM/ OCRAM reserved for SP)
PBI Phase	ERROR_PBI_CCSR_BYTE_COUNT	0x74	Byte count in CCSR Write not valid
PBI Phase	ERROR_PBI_CCSR_4_BYTE_ALLIGN ED	0x75	Offset is not 4 byte aligned
PBI Phase	ERROR_PBI_CCSR_OFFSET_INVALID	0x76	Offset is invalid i.e less than allowed CCSR Base 0x0100_0000
PBI Phase	ERROR_PBI_ACSR_INVALID_ADDRE SS	0x78	Source address in ACSR invalid (invalid addresses - OCRAM or ROM address)
PBI Phase	ERROR_PBI_ACSR_BYTE_COUNT	0x79	Byte count in ACSR write command not valid
PBI Phase	ERROR_PBI_ACSR_WINDOW_NOT_S ET	0x7a	ATU Window is not configured
PBI Phase	ERROR_PBI_ACSR_OFFSET_ALLIGN ED	0x7b	ACSR offset is invalid and trying to write to Reserved space on OCRAM.
PBI Phase	ERROR_PBI_ALTCFG_WNDW_INVALI D	0x7c	ATU Window is invalid
PBI Phase	ERROR_PBI_JUMP_OUT_LENGTH	0x80	Offset specified in JUMP command doesn't lie in PBI length range
PBI Phase	ERROR_PBI_JUMP_4_BYTE_ALLIGNE D	0x81	Offset specified in JUMP command is not 4 byte aligned
PBI Phase	ERROR_PBI_JUMP_OFFSET_0	0x82	Offset specified in JUMP command is 0
PBI Phase	ERROR_PBI_LOADC_4_BYTE_ALLIGN ED	0x84	Address specified in LOAD Condition command is not 4 byte alligned
PBI Phase	ERROR_PBI_JUMPC_OUT_LENGTH	0x88	Offset specified in JUMP command doesn't lie in PBI length range
PBI Phase	ERROR_PBI_JUMPC_4_BYTE_ALLIGN ED	0x89	Offset specified in JUMP Conditional command is not 4 byte aligned
PBI Phase	ERROR_PBI_JUMPC_CONDITION_NO T_SET	0x8a	Jump conditional command encountered before condition is set using Load Condition

Table continues on the next page...

Table 387. ISBC Error Codes (continued)

When Error Generated	Error Code	Value	Description
PBI Phase	ERROR_PBI_CRC_MISMATCH	0x90	CRC Mismatch
PBI Phase	ERROR_PBI_POLL	0x91	Poll Timeout
PBI Phase	ERROR_PBI_POLL_4_BYTE_ALLIGNED	0x92	Address being polled is not 4 byte aligned
PBI Phase	ERROR_PBI_BOOT1_CSF_INVALID_ADDR	0x94	Address of CSF Header is not valid.
PBI Phase	ERROR_PBI_BOOT1_CSF_ALLIGNED	0x95	Address of CSF Header is not 4 byte aligned
Phase = **Verify** (System State Errors (Secure boot))			
Before PBI verification	ERROR_STATE_NOT_CHECK	0xf0	SEC_MON State Machine not in CHECK state at start of ISBC in primary flow. Some Security violation could have occurred.
Before PBI verification	ERROR_STATE_NOT_CHECK_TRUSTED	0xf1	SEC_MON State Machine not in CHECK/ Trusted state at start of ISBC in secondary flow.
Phase = **Verify** (Secure Boot Fatal errors)			
Verify PBI	ERROR_PBI_COMMANDS_NOT_FOUND	0xf4	Not having PBI commands in RCW is error scenario for secure boot
Verify PBI	ERROR_SEC_HDR_NOT_FOUND	0xf5	Error if security header command not found in RCW. Expected location of Security Header command • After Preamble for hard coded RCW • After preamble and rcw for other RCW sources
Phase = **Verify** (Secure Boot Fatal (Header parsing errors))			
Verify PBI	ERROR_HEADER_LOC	0xf8	Header Location is invalid
Verify PBI	ERROR_HEADER_BARKER	0xf9	Barker code in the header is incorrect.
Verify PBI	ERROR_HEADER_INVALID	0xfa	Flag B01 in the header identifies this as SPL Header.
Phase = **Verify** (Secure Boot Non Fatal (Key/UID related errors))			
Verify PBI	ERROR_INVALID_SRK_ENTRY_KEYLEN	0x210	Length of public key specified in one of the entries in srk table is not one of the supported values. (1k, 2k or 4k)
Verify PBI	ERROR_ KEY_LEN_ NOT_TWICE_SIG_LEN	0x211	Public key is not twice the length of the RSA signature
Verify PBI	ERROR_ KEY_MOD_1	0x212	Most significant bit of modulus in header is zero.

Table continues on the next page...

Table 387. ISBC Error Codes (continued)

When Error Generated	Error Code	Value	Description
Verify PBI	ERROR_ KEY_MOD_2	0x213	Modulus in header is even number
Verify PBI	ERROR_ SIG_KEY_MOD	0x214	Signature value is greater than modulus in header
Verify PBI	ERROR_ INVALID_SRK_NUM_ENTRY	0x215	Number of entries field in CSF Header is > 8(This is when srk_flag in header is 1)
Verify PBI	ERROR_ INVALID_KEY_NUM	0x216	Key number to be used from srk table is not present in table.(This is when srk_flag in header is 1)
Verify PBI	ERROR_ KEY_REVOKED	0x217	Key selected from srk table has been revoked(This is when srk_flag in header is 1)
Verify PBI	ERROR_ FSL_UID	0x220	FSL_UID in ESBC Header did not match the FSL_UID in SFP if fsl uid flag Is 1
Verify PBI	ERROR_ OEM_UID0	0x221	OEM_UID0 in ESBC Header did not match the OEM_UID0 in SFP if oem uid0 flag is 1.
Verify PBI	ERROR_ OEM_UID1	0x222	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify PBI	ERROR_ OEM_UID2	0x223	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify PBI	ERROR_ OEM_UID3	0x224	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify PBI	ERROR_ OEM_UID4	0x225	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Phase = **Verify** (Header Verification Failure) Secure Boot Non Fatal			
Verify PBI	ERROR_ HASH_COMPARE_KEY	0x240	Super Root Key Hash Comparison failure. Mismatch in the hash of the public key/srk table as present in the header with the value in the SRK HASH fuse.
Verify PBI	ERROR_ HASH_COMPARE_EM	0x241	RSA signature check failure. Signature provided by you in the header doesn't match with the signature of the ESBC image generated by ISBC. The ESBC image loaded by you may be different than the image used while generating the signature(using CST)
Phase = **Verify** (Secure Boot Fatal (Header parsing errors))			
Verify Boot1	ERROR_HEADER_LOC	0x100f8	Header Location is invalid
Verify Boot1	ERROR_HEADER_BARKER	0x100f9	Barker code in the header is incorrect.
Verify Boot1	ERROR_HEADER_INVALID	0x100fa	Flag B01 in the header identifies this as SPL Header.

Table continues on the next page...

Table 387. ISBC Error Codes (continued)

When Error Generated	Error Code	Value	Description
Phase = Verify (Secure Boot Fatal (SG Table related errors))			
Verify Boot1	ERROR_SG_ENTRY_POINT	0x10200	Entry point is not within any of SG Entries
Verify Boot1	ERROR_SG_NUM_ENTRY	0x10201	No. of entries in SG Table is 0 or >8
Verify Boot1	ERROR_SG_SIZE_ZERO	0x10202	A SG entry has size 0
Phase = **Verify** (Secure Boot Non-Fatal (Key/UID related errors))			
Verify Boot1	ERROR_INVALID_SRK_ENTRY_KEYLEN	0x10210	Length of public key specified in one of the entries in srk table is not one of the supported values. (1k, 2k or 4k)
Verify Boot1	ERROR_ KEY_LEN_ NOT_TWICE_SIG_LEN	0x10211	Public key is not twice the length of the RSA signature
Verify Boot1	ERROR_ KEY_MOD_1	0x10212	Most significant bit of modulus in header is zero.
Verify Boot1	ERROR_ KEY_MOD_2	0x10213	Modulus in header is even number
Verify Boot1	ERROR_ SIG_KEY_MOD	0x10214	Signature value is greater than modulus in header
Verify Boot1	ERROR_ INVALID_SRK_NUM_ENTRY	0x10215	Number of entries field in CSF Header is > 8(This is when srk_flag in header is 1)
Verify Boot1	ERROR_ INVALID_KEY_NUM	0x10216	Key number to be used from srk table is not present in table.(This is when srk_flag in header is 1)
Verify Boot1	ERROR_ KEY_REVOKED	0x10217	Key selected from srk table has been revoked(This is when srk_flag in header is 1)
Verify Boot1	ERROR_ FSL_UID	0x10220	FSL_UID in ESBC Header did not match the FSL_UID in SFP if fsl uid flag Is 1
Verify Boot1	ERROR_ OEM_UID0	0x10221	OEM_UID0 in ESBC Header did not match the OEM_UID0 in SFP if oem uid0 flag is 1.
Verify Boot1	ERROR_ OEM_UID1	0x10222	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify Boot1	ERROR_ OEM_UID2	0x10223	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify Boot1	ERROR_ OEM_UID3	0x10224	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify Boot1	ERROR_ OEM_UID4	0x10225	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Phase = **Verify** (Header Verification Failure) Secure Boot Non-Fatal			

Table continues on the next page...

Table 387. ISBC Error Codes (continued)

When Error Generated	Error Code	Value	Description
Verify Boot1	ERROR_ HASH_COMPARE_KEY	0x10240	Super Root Key Hash Comparison failure. Mismatch in the hash of the public key/srk table as present in the header with the value in the SRK HASH fuse.
Verify Boot1	ERROR_ HASH_COMPARE_EM	0x10241	RSA signature check failure. Signature provided by you in the header doesn't match with the signature of the ESBC image generated by ISBC. The ESBC image loaded by you may be different than the image used while generating the signature(using CST)
Verify Boot1	ERROR_PRIVATE_KEY_DERIVATION	0x10250	Error in Private key derivation when enabling Manufacturing Protection.

10.3.3.11 ESBC Error Codes

Table 388. ESBC Validation Failures

Value	Code	Definition
0x4	ERROR_ESBC_CLIENT_HEADER_BARKER	Wrong barker code in header
0x8	ERROR_ESBC_CLIENT_HEADER_KEY_LEN	Wrong public key length in header
0x10	ERROR_ESBC_CLIENT_HEADER_SIG_LEN	Wrong signature length in header
0x20	ERROR_ESBC_CLIENT_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key length not twice of signature length
0x40	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_1	Public key Modulus most significant bit not set
0x80	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_2	Public key Modulus in header not odd
0x100	ERROR_ESBC_CLIENT_HEADER_SIG_KEY_MOD	Signature not less than modulus
0x400	ERROR_ESBC_CLIENT_HASH_COMPARE_KEY	Public key hash comparison failed
0x800	ERROR_ESBC_CLIENT_HASH_COMPARE_EM	RSA verification failed

Table continues on the next page...

Table 388. ESBC Validation Failures (continued)

Value	Code	Definition
0x10000	ERROR_ESBC_CLIENT_HEADER_SG	No SG support
0x20000	ERROR_ESBC_WRONG_CMD	Failure in command/Unknown command/Wrong arguments of boot script.
0x40000	ERROR_ESBC_MISSING_BOOTM	Bootm command missing from boot script.

10.3.3.12 Address Map used for demo

The Base addresses for device/flash are:

NOR (Bank 0) 0x5_8000_0000 (0x3000_0000[42])

NOR (Alt Bank 4) 0x5_8400_0000 (0x3400_0000)

Table 389. Address Map Used For Demo

Offset on Flash/Device	Address on DDR (To Copy the Image)	Size (Reserved)	Definition
0x0	<>	0x10_0000 (1 MB)	rcw_sec.bin
0x010_0000	<>	0x10_0000 (1 MB)	Boot Loader (U-Boot.bin)
0x020_0000	<>	0x10_0000 (1 MB)	Boot Loader Environment
0x030_0000	<>	0x40_0000 (4 MB)	MC FW
0x070_0000	0xA070_0000	0x10_0000 (1 MB)	MC DPL
0x080_0000	<>	0x10_0000 (1 MB)	MC DPC
0x0A0_0000	<>	0x10_0000 (1 MB)	PPA
0x0C0_0000	<>	0x4000(16K)	hdr_uboot.out
0x0C4_0000	0xA0C4_0000	0x4000 (16K)	hdr_ppa.out
0x0C8_0000	0xA0C8_0000	0x4000(16K)	hdr_mc.out
0x0CC_0000	0xA0CC_0000	0x4000(16K)	hdr_dpc.out
0x0D0_0000	0xA0D0_0000	0x4000(16K)	hdr_bs.out
0x0D4_0000	0xA0D4_0000	0x4000 (16K)	hdr_dpl.out
0x0D8_0000	0xA0D8_0000	0x4000 (16 K)	hdr_kernel.out
0x0E0_0000	0xA0E0_0000	0x2_0000 (128 KB)	BootScript
0x110_0000	0xA110_0000	0x280_0000 (40 MB)	Kernel Fit Image

[42] There is an address in 32 bit address space and 64 bit address space. Please refer the Platform Address map for details.

10.3.3.13 Useful Commands

This section contains some useful commands for loading images via U-Boot (As per memory map defined in this document) and CCS commands to load the SRK Hash in shadow registers and get the system out of Reset Pause State in Development Mode.

NOTE

The CCS commands to connect and configure config chain might change with Board/ Silicon revision. See the Board Manual/ CCS Guide in case of issues with CCS commands.The commands provided below are for reference only.

NOTE

For permanently blowing the values (OTPMK, SRK HASH etc.) in SFP, refer to the details in *Trust Architecture User Guide*. A brief summary of the steps is described below:

1. Ensure that PROG_SFP/POVDD(Table 381. Trust Architecture and SFP Information on page 2096) signal is correctly asserted. This is usually controlled via a switch or a jumper. (Refer Board schematic/manual for the same)
2. Write the required fuse values to the SFP mirror registers.
3. To permamnetly blow the fuses, write to 'PROGFB' in SFP Instruction Register (SFP_INGR).

   Make sure that the values written in SFP mirror registers are correct before blowing the fuses as once blown, the fuse values cannot be changed.

   - To check if OTPMK is blown or not on the Silicon, check the bit 'OTPMK_ZERO' in the SECMON_HPSR register. If the bit is set, it means OTPMK is zero i.e. OTPMK needs to be blown.
   - The OTPMK value can be generated using 'gen_otpmk_drbg' tool provided in CST.
   - After writing the values in SFP mirror registers, check the hamming error register. Ensure that the value is 0x0 i.e. no error is reported.(Table 381. Trust Architecture and SFP Information on page 2096)
     - It is reported in SFP Secret Value Hamming Error Status Register (SFP_SVHESR).

U-Boot Commands to Flash Images on NOR Bank 4

```
setenv dir <tftp folder path>

setenv cprcw 'protect off all ; tftp 0x80000000 $dir/rcw_sec.bin; echo Copying rcw ; erase
0x584000000 +$filesize; cp 0x80000000 0x584000000 $filesize ; cmp.b 0x80000000
0x584000000 $filesize;'
setenv cpuboothdr 'tftp 0x80000000 $dir/hdr_uboot.out; echo Copying uboot header ; erase
0x584C00000 +$filesize; cp 0x80000000 0x584C00000 $filesize; cmp.b 0x80000000
0x584C00000 $filesize;'
setenv cpubootsecure 'tftp 0x80000000 $dir/u-boot-dtb.bin; erase 0x584100000 +$filesize; echo
Copying Uboot ; cp 0x80000000 0x584100000 $filesize; echo Comparing Uboot ; cmp.b 0x80000000
0x584100000 $filesize;'

setenv cpppasecure 'tftp 0x80000000 $dir/ppa.itb; erase 0x584A00000 +$filesize; echo Copying PPA ;
cp 0x80000000 0x584A00000 $filesize; echo Comparing PPA ; cmp.b 0x80000000 0x584A00000 $filesize;'
setenv cpppahdr 'tftp 0x80000000 $dir/hdr_ppa.out; erase 0x584C40000 +$filesize; echo Copying PPA
Header ; cp 0x80000000 0x584C40000 $filesize; echo Comparing PPA header ; cmp.b 0x80000000
0x584C40000 $filesize;'

setenv cpbootscripthdr 'tftp 0x80000000 $dir/hdr_bs.out; echo Copying bootscript header ; erase
0x584D00000 +$filesize; cp 0x80000000 0x584D00000  $filesize; cmp.b 0x80000000
0x584D00000  $filesize;'
```

```
setenv cpbootcriptsecure 'tftp 0x80000000 $dir/bootscript; erase 0x584E00000 +$filesize; echo
Copying Bootscript ; cp 0x80000000 0x584E00000 $filesize; echo Comparing Bootscript; cmp.b
0x80000000 0x584E00000 $filesize;'

setenv cpkernelsecure 'tftp 0x80000000 $dir/kernel.itb; erase 0x585100000 +$filesize; echo Copying
Kernel ; cp.b 0x80000000 0x585100000 $filesize; echo Comparing Kernel; cmp.b 0x80000000
0x585100000 $filesize;'
setenv cpkernelhdr 'tftp 0x80000000 $dir/hdr_kernel.out; erase 0x584D80000 +$filesize; echo Copying
Kernel ; cp.b 0x80000000 0x584D80000  $filesize; echo Comparing Kernel; cmp.b 0x80000000
0x584D80000 $filesize;'

setenv cpmc 'tftp 0x80000000 $dir/mc.itb; erase 0x584300000 +$filesize; echo Copying MC ; cp.b
0x80000000 0x584300000  $filesize; echo Comparing MC; cmp.b 0x80000000  0x584300000  $filesize;'
setenv cpdpc 'tftp 0x80000000 $dir/dpc.dtb; erase 0x584800000 +$filesize; echo Copying DPC ; cp.b
0x80000000 0x584800000 $filesize; echo Comparing DPC; cmp.b 0x80000000  0x584800000 $filesize;'
setenv cpdpl 'tftp 0x80000000 $dir/dpl.dtb; erase 0x584700000 +$filesize; echo Copying DPL ; cp.b
0x80000000 0x584700000 $filesize; echo Comparing DPL; cmp.b 0x80000000  0x584700000 $filesize;'
setenv cpmchdr 'tftp 0x80000000 $dir/hdr_mc.out; erase 0x584C80000  +$filesize; echo Copying MC
HDR; cp.b 0x80000000 0x584C80000   $filesize; echo Comparing MC HDR; cmp.b 0x80000000
0x584C80000 $filesize;'
setenv cpdpchdr 'tftp 0x80000000 $dir/hdr_dpc.out; erase 0x584CC0000  +$filesize; echo Copying DPC
HDR; cp.b 0x80000000 0x584CC0000   $filesize; echo Comparing DPC HDR; cmp.b 0x80000000
0x584CC0000 $filesize;'
setenv cpdplhdr 'tftp 0x80000000 $dir/hdr_dpl.out; erase 0x584D40000  +$filesize; echo Copying DPL
HDR ; cp.b 0x80000000 0x584D40000 $filesize; echo Comparing DPL HDR; cmp.b 0x80000000
0x584D40000   $filesize;'

setenv runsecuboot 'run cprcw ; run cpuboothdr; run cpubootsecure; run cpbootscripthdr; run
cpbootcriptsecure; '
setenv runsecmc 'run cpppasecure;run cpppahdr; run cpmc;run cpdpc; run cpdpl; run cpmchdr; run
cpdpchdr; run cpdplhdr;'
setenv runseckernel 'run cpkernelsecure; run cpkernelhdr;'

run runsecuboot
run runsecmc
run runseckernel
```

Enabling Reset Pause State[43]

LS2 QDS Board	Switch: **SW2.7 – 0**
	U-Boot Command: **i2c mw 0x66 0x66 0x7f 1**
LS2 RDB Board	Switch (Rev B): **SW3.8 – 0**
	Switch (Rev C to Rev E): **SW4.8 – 0**
	U-Boot Command: None (**No register available in FPGA**)
	To boot from vbank4 change **SW9[3:5] to 100**

CCS Commands to Write SRK Hash and Get System Out of Reset Pause State

```
(RSP has two states : RSP1 and RSP2. Initially we are in RSP1 state)
delete all;config cc cwtap;show cc
# use config cc cwtap:<tap_id> if using remote board
ccs::config_chain {<platform> sap2}
#display ccs::get_config_chain
puts "Entering RSP2: "
ccs::write_mem <SAP CHAIN_POS> 0x7 0x001000D0 0x4 0x0 0x800
```

[43] Please refer the board schematic/guide for exact switches on the board.

```
#Write Key In sfp  (Reversed byte order of key hash)
#Example Key Hash: c9f90c9762e1693804421a4bd8193735b38ea03b83303d9529f7b1b5d2e4688f
# Key Hash must be written after byte swapping as SRKH is Little Endian
ccs::write_mem <SAP CHAIN_POS> 0x1E80254 4 0 0x970CF9C9;
ccs::write_mem <SAP CHAIN_POS> 0x1E80258 4 0 0x3869E162;
ccs::write_mem <SAP CHAIN_POS> 0x1E8025c 4 0 0x4B1A4204;
ccs::write_mem <SAP CHAIN_POS> 0x1E80260 4 0 0x353719D8;
ccs::write_mem <SAP CHAIN_POS> 0x1E80264 4 0 0x3BA08EB3;
ccs::write_mem <SAP CHAIN_POS> 0x1E80268 4 0 0x953D3083;
ccs::write_mem <SAP CHAIN_POS> 0x1E8026c 4 0 0xB5B1F729;
ccs::write_mem <SAP CHAIN_POS> 0x1E80270 4 0 0x8F68E4D2;

puts "Exiting RSP: "
ccs::write_mem 2 0x7 0x001000D0 0x4 0x0 0x400;
```

10.3.3.14 Troubleshooting

	Symptoms	Reasons and/or Recommended actions
1.	No print on UART console.	Check the status register of sec mon block. Refer to the details of the register from the Reference Manual. Bits OTPMK_ZERO, OTMPK_SYNDROME and PE should be 0 otherwise there is some error in the OTPMK fuse blown by you. If OTMPK fuse is correct (see Step 1), check the DCFG SCRATCHRW3 register for error code. For a list of error codes see ISBC Error Codes on page 2104 If **Error code = 0** then check the Security Monitor state in HPSR register of Sec Mon. Sec Mon in Check State (0x9) If ITS fuse = 1, then it means ISBC code has reset the board. This may be due to the following reasons “ Hash of the public key used to sign the ESBC u-boot doesn’t match with the value in SRK Hash Fuse Or Signature verification of the image failed Sec Mon in Trusted State (0xd) or Non Secure State (0xb) Check the entry point field in the CSF header. If entry point is correct, ensure that u-boot image has been signed with the correct input file.
2.	Instead of linux prompt, you get a u-boot command prompt.	You have not booted in secure boot mode. You never get a u-boot prompt in secure boot flow. You would reach this stage if ITS = 0 and you are running normal uboot.
3	u-boot hangs or board resets	Some validation failure occurred in u-boot. Error code and description would be printed on u-boot console. See ESBC Error Codes on page 2111for more details on errors.

10.3.4 Code Signing Tool

To assist with signing of various images and creation of CSF header, NXP offers a Code Signing Tool (CST). It is generally expected that the CST signs images in an offline process

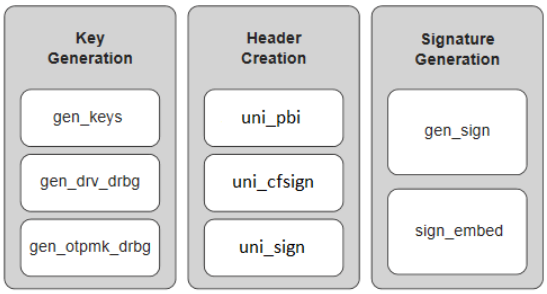


Figure 341. Tool in CST Package

CST tool installation steps

Step 1 Yocto installs the cst package at the following location:

tmp/sysroots/x86_64-linux/usr/bin/cst

OR

In the Yocto environment, the user can use below commands to rebuild cst:

1. bitbake cst-native –c cleanall
2. bitbake cst-native

Step 2 cd ***tmp/sysroots/x86_64-linux/usr/bin/cst***

Note: LD_LIBRARY_PATH should be set to the library path in yocto workspace. <project_folder_path>/tmp/sysroots/x86_64-linux/usr/lib

10.3.4.1 Key generation

The CST begins by generating a RSA public and private key pair using OPENSSL APIs. The key pair consists of 3 parts - N, E and D.

N - Modulus

E - Encryption exponent

D - Decryption exponent

Public Key - It is a combination of E and N components.

Private Key - It is a combination of D and N components.

The application allows the user to feed 3 key sizes for generating keys. The sizes allowed are - 1024 bits, 2048 bits and 4096 bits.

It is the OEM's responsibility to tightly control access to the RSA private signature key. If this key is ever exposed, attackers will be able to generate alternate images that will pass secure boot. If this key is ever lost, the OEM will be unable to update the image

10.3.4.1.1 gen_keys

This utility generates a RSA public and private key pair using OPENSSL APIs. The key pair consists of 3 parts: N, E and D.

N – Modulus

E – Encryption exponent

D – Decryption exponent

Public Key - It is a combination of E and N components.

Private Key - It is a combination of D and N components.

It is the OEM's responsibility to tightly control access to the RSA private signature key. If this key is ever exposed, attackers will be able to generate alternate images that will pass secure boot. If this key is ever lost, the OEM will be unable to update the image.

Features

- The application allows the user to generate 3 sizes keys. The sizes allowed are - 1024 bits, 2048 bits and 4096 bits.
- It generates RSA key pairs in PEM format.
- Keys are generated and stored in the files. User can provide filenames through command line option.

Usage

./gen_keys [OPTION] SIZE

SIZE refers to size of public key in bits. (Modulus size).

Sizes supported -- 1024, 2048, 4096. The generated keys would be in PEM format.

Options:

-h,--help Usage of the command

-k,--pubkey File where Public key would be stored in PEM format(default = srk.pub)

-p,--privkey File where Private key would be stored in PEM format(default = srk.priv)

Usage Example

```
$ ./gen_keys 1024

    #------------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #------------------------------------------------------#

================================================================
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
```

```
Eric Young (eay@cryptsoft.com)
============================================================

Generated SRK pair stored in :
        PUBLIC KEY srk.pub
        PRIVATE KEY srk.pri
```

```
$ ./gen_keys 4096 -k my.pub -p my.pri

    #--------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #--------------------------------------------------#

============================================================
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
============================================================

Generated SRK pair stored in :
        PUBLIC KEY my.pub
        PRIVATE KEY my.pri
```

10.3.4.1.2 gen_otpmk_drbg

This utility in the Code Signing Tool inserts hamming code in a user defined 256b hexadecimal string, or generate a 256b hexadecimal random number and inserts the hamming code in it which can be used as OTPMK value.

NOTE

For random number generation, Hash_DRBG library is used. The Hash_DRBG is an implementation of the NIST approved DRBG(Deterministic Random Bit Generator), specified in SP800-90A. The entropy source is the Linux /dev/random.

Features:

- Generates random numbers, which can be used if user defined string is not provided, to generate OTPMK value.
- Calculates and embeds the hamming code in the hexadecimal string.

Usage:

./gen_otpmk_drbg <bit_order> [string]

<bit_order> : (1 or 2) OTPMK Bit Ordering Scheme in SFP

1 : BSC913x, P1010, P3, P4, P5, C29x

2 : T1, T2, T4, B4, LSx

<string> : 32 byte string

In case string is not specified, the utility generates a 32 bytes random number and embeds hamming code in it.

Usage Example:

```
$ ./gen_otpmk_drbg 2

    #--------------------------------------------------#
    #-------         --------     --------        -------#
```

```
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #------------------------------------------------------#

Input string not provided
Generating a random string
-----------------------------------------
* Hash_DRBG library invoked
* Seed being taken from /dev/random
-----------------------------------------
OTPMK[255:0] is:
3feac02ce3583ad9077ab70f3a398cd71955f8bffa3191428cb25bb6bffb3113

 NAME    |     BITS     |    VALUE
_________|______________|___________
OTPMKR 0 | 255-224    |   3feac02c
OTPMKR 1 | 223-192    |   e3583ad9
OTPMKR 2 | 191-160    |   077ab70f
OTPMKR 3 | 159-128    |   3a398cd7
OTPMKR 4 | 127- 96    |   1955f8bf
OTPMKR 5 |  95- 64    |   fa319142
OTPMKR 6 |  63- 32    |   8cb25bb6
OTPMKR 7 |  31-  0    |   bffb3113
```

```
$ ./gen_otpmk_drbg 2 1234567856485626a6f6e617485858384772067353 4a8958c983774b848438fe

    #------------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #------------------------------------------------------#

OTPMK[255:0] is:
ce3c563856584664a6b66617485a5e3d46720673534a8958c983774b848438fe

 NAME    |     BITS     |    VALUE
_________|______________|___________
OTPMKR 0 | 255-224    |   ce3c5638
OTPMKR 1 | 223-192    |   56584664
OTPMKR 2 | 191-160    |   a6b66617
OTPMKR 3 | 159-128    |   485a5e3d
OTPMKR 4 | 127- 96    |   46720673
OTPMKR 5 |  95- 64    |   534a8958
OTPMKR 6 |  63- 32    |   c983774b
OTPMKR 7 |  31-  0    |   848438fe
```

10.3.4.1.3 gen_drv_drbg

This utility in the Code Signing Tool inserts hamming code in a user defined 64b hexadecimal string, or generate a 64b hexadecimal random number and inserts the hamming code in it which can be used as Debug Response Value.

NOTE

For random number generation, Hash_DRBG library is used. The Hash_DRBG is an implementation of the NIST approved DRBG(Deterministic Random Bit Generator), specified in SP800-90A. The entropy source is the Linux /dev/random.

Features:

- Generates random numbers, which can be used if user defined string is not provided, to generate Debug Response value.
- Calculates and embeds the hamming code in the hexadecimal string.

Usage:

./gen_drv_drbg <Hamming_algo> [string]

Hamming_algo : Platforms

A1 : T10xx, T20xx, T4xxx, P4080rev1, B4xxx

A2 : LSx

B : P10xx, P20xx, P30xx, P4080rev2, P4080rev3, P50xx, BSC913x, C29x

string : 8 byte string

In case string is not specified, the utility generates an 8 byte random number and embeds hamming code in it.

Usage Example:

```
$ ./gen_drv_drbg A2

    #-------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #-------------------------------------------------#

Input string not provided
Generating a random string
-----------------------------------------
* Hash_DRBG library invoked
* Seed being taken from /dev/random
-----------------------------------------
Random Key Genearted is:
f4bfc65e16284dbb
DRV[63:0] after Hamming Code is:
f4bfc65f16294daf
 NAME    |     BITS     |    VALUE
_________|______________|___________
DRV 0    |    63 - 32   | f4bfc65f
DRV 1    |    31 - 0    | 16294daf
```

```
$ ./gen_drv_drbg A2 1652afe595631dec

    #-------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #-------------------------------------------------#

DRV[63:0] after Hamming Code is:
1652afe495631cea
 NAME    |     BITS     |    VALUE
_________|______________|___________
DRV 0    |    63 - 32   | 1652afe4
DRV 1    |    31 - 0    | 95631cea
```

10.3.4.2 Header creation

10.3.4.2.1 uni_pbi

Options available with the uni_pbi command are as follows:

```
$ ./uni_pbi
        --verbose    Display header Info after Creation. This option is invalid for TA2 platform
        --hash       Print the SRK(Public key) hash. This option is invalid for TA2 platform
        --img_hash   Header is generated without Signature.
                     Image Hash is stored in a separate file. This option is invalid for TA2
platform
        --help       Show the Help for Tool Usage.
```

The input to this tool will be an input file specifying the platform, and based on that there are two separate behaviour of the tool:

uni_pbi for TA2.x platforms is used for the following:

- To add boot location pointer and set SB_EN and BOOT_HO value for secure boot
- (optional) To add PBI commands (ACS write commands to add u-boot-spl and its header to OCRAM from Non-XIP memory).
- (optional) To append images (uboot, bootscript and their headers) to RCW file.

Refer Hardware PreBoot Loader (PBL) Based Platforms on page 1940 for TA2.x based platforms

uni_pbi for Service processor based platforms

- uni_pbi tool is used for creating signature and header over PBI commands.

Table 390. Description of Fields in input files of both type of platforms (TA2.x and TA3.x)

Field Name	Description	Platform supported
PLATFORM	The platform for which tool is being used	TA 2.x and TA 3.x
RCW_PBI_FILENAME	Input image file name. The rcw file which has to be modified.	TA 2.x and TA 3.x
BOOT1_PTR	Address of ISBC (Boot1) CSF Header	TA 2.x and TA 3.x
OUTPUT_RCW_PBI_FILENAME	To identify the platform for which the tool is being used.. This field is optional. if not specified, it will take default name.	TA 2.x
BOOT_SRC	Only to be specified in case of **SD boot**	TA 2.x
SB_EN	Field to enable/disable secure boot, by setting SB_EN bit in rcw file to 1	TA 2.x
BOOT_HO	To put core in hold-off state to fuse key hash in case of secure boot, by settin BOOT_HO bit in rcw file to 1	TA 2.x
COPY_CMD	To add ACS write commands to write u-boot spl and is header to OCRAM. This is an optional field. If not mentioned, won't add the command.	TA 2.x
APPEND_IMAGES	To append u-boot, bootscript and their headers to the new rcw generated. It is an optional field. This is an optional field, if not specified no images will be appended.	TA 2.x

Table continues on the next page...

Table 390. Description of Fields in input files of both type of platforms (TA2.x and TA3.x) (continued)

Field Name	Description	Platform supported
KEY_SELECT	Specify the key to be used in signature generation from the SRK Table	TA 3.x
PRI_KEY	Private key filename in PEM format. Maximum 8 keys supported.	TA 3.x
FSL_UID_x	FSL UID(s) to be populated in the header	TA 3.x
OEM_UID_x	OEM UID(s) to be populated in the header	TA 3.x
OUTPUT_HDR_FILENAME	Output File name of the Header. An out put file name is generated with rcw commands appended with signed PBI commands.	TA 3.x
IMAGE_HASH_FILENAME	used with '--img_hash' option (Name of file in which Image Hash is stored)	TA 3.x
MP_FLAG	Manufacturing Protection Flag	TA 3.x
ISS_FLAG	Increment Security State Flag	TA 3.x
LW_FLAG	Leave Writeable Flag	TA 3.x
VERBOSE	Specify VERBOSE as 1, if you want to Display Header Information. Can also be done with '--verbose' option	TA 3.x
IE_TABLE_ADDR	64 bit address of IE table(to be used in case of IE key extension feature usage)	TA 3.x

Sample input files are present in the CST tool at location: input_files/uni_pbi/<platform>/

eg. input_files/uni_pbi/ls1/input_pbi_sd_secure

NOTE

In TA 3.x, SB_EN and BOOT_HO fields are by default set to 1 to enable secure boot.

NOTE

TA 3.x : LS1088, LS2088. To know platforms under TA 2.x refer Trust Architecture and SFP Information on page 1996

10.3.4.2.1.1 Sample Input File

Sample input file for TA2 based platforms

```
/*
 * Copyright 2016 NXP
 */
------------------------------------------------------------------------------
# For PBI Creation
# Name of RCW + PBI file [Mandatory]
RCW_PBI_FILENAME= u-boot-with-spl-pbl.bin
------------------------------------------------------------------------------
# Specify the output file name [Optional].
# Default Values chosen in Tool
OUTPUT_RCW_PBI_FILENAME=u-boot-with-spl-pbl-sec.bin
------------------------------------------------------------------------------
```

```
#specify the boot src
BOOT_SRC=SD_BOOT
# Specify the platform
PLATFORM=LS1020
# Specify the RCW Fields. (0 or 1) - [Optional]
SB_EN=1
BOOT_HO=1
BOOT1_PTR=10016000
--------------------------------------------------------------------------------
# Specify the PBI commands - [Optional]
# Argument: COPY_CMD = (src_offset, dest_offset, Image name)
# Split hdr_uboot_spl.out in PBI commads
COPY_CMD={ffffffff,10016000,hdr_uboot_spl.out;}
--------------------------------------------------------------------------------
# Specify the Images to be appended
# Arguments: APPEND_IMAGES=(Image name, Offset from start)
APPEND_IMAGES={u-boot-dtb.bin,00022000;}
APPEND_IMAGES={hdr_uboot.out,00122000;}
APPEND_IMAGES={hdr_bs.out, 00124000;}
APPEND_IMAGES={bootscript,00128000;}
--------------------------------------------------------------------------------
```

Sample input file for SP based platforms

```
--------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
PLATFORM=LS1088
--------------------------------------------------
# Specify the Key Information.
# PUB_KEY [Mandatory] Comma Seperated List
# Usage: <srk1.pub>, <srk2.pub> .....
PUB_KEY=srk.pub
# KEY_SELECT [Mandatory]
# USAGE (for TRUST 3.1): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Comma Seperated List for Signing
# USAGE: <srk.pri>, <srk2.pri>
PRI_KEY=srk.pri
--------------------------------------------------
# For PBI Signing
# Name of RCW + PBI file [Mandatory]
RCW_PBI_FILENAME=rcw.bin
# Address of ISBC (Boot1) CSF Header [Mandatory]
BOOT1_PTR=20c00000
--------------------------------------------------
# Specify OEM AND FSL ID to be populated in header. [Optional]
# e.g FSL_UID_0=11111111
FSL_UID_0=
FSL_UID_1=
OEM_UID_0=
OEM_UID_1=
OEM_UID_2=
OEM_UID_3=
OEM_UID_4=
--------------------------------------------------
# Specify the output file names [Optional].
# Default Values chosen in Tool
```

```
OUTPUT_HDR_FILENAME=rcw_sec.bin
IMAGE_HASH_FILENAME=
-------------------------------------------------
# Specify The Flags. (0 or 1) - [Optional]
MP_FLAG=0
ISS_FLAG=1
LW_FLAG=0
-------------------------------------------------
# Specify VERBOSE as 1, if you want to Display Header Information [Optional]
VERBOSE=0
```

10.3.4.2.2 CF Header Generation

uni_cfsign command is provided to generate cf header.

./uni_cfsign INPUT_FILE

A QORIQ device can boot securely from multiple targets like NOR flash, NAND flash , SPI flash and SD/MMC (P1010 does not support SD/MMC). For each target, an input file is built which contains the configuration words and populates the S/G table. CF header is generated with the information as described in CF Header Data Structure Definition on page 2041. Legacy mode fields contain the information related to the legacy image like source address, destination address, length and entry point. Configuration words are used to – create LAW for DDR, initialize the DDR controller. ESBC source address points to the ESBC/CSF header. CF header generation is valid only for the below mentioned platforms

- p1010
- bsc9131
- bsc9132
- c290

INPUT_FILE - Sample input files have been provided in the input_files directory in the cst tool.

- input_files/uni_cfsign/p1010/
- input_files/uni_cfsign/bsc9131/
- input_files/uni_cfsign/bsc9132/
- input_files/uni_cfsign/c290/

For cf header generation input file contains following major piece of information -

1. Target interface ID (IMAGE_TARGET)
2. Address where ESBC header will placed on the target (ESBC_HDRADDR)
3. Configuration words (address/data tuples)

The use of configuration address and data pairs for the following configuration is not allowed before the control is given to the ESBC image:

- Change of CCSRBAR
- Programming of ALTCSR
- Programming of boot page translation register
- SNVS
- CCSR Access Control
- Secure Debug Control
- MPIC
- SFP

There are certain points which should be adhered to while writing the configuration file:

1. The CF Header should be on the boot_device within first 512bytes.
2. If using the same data structure for secure and non secure booting modes, make sure that entry points are different in both cases.
3. In legacy mode with SD/MMC card in high capacity mode, the Source Address field is translated as containing the address in block address format. Whereas, in case of Secure mode even if the SD/MMC card is High capacity the public key offset, signature offsets are all treated as 32 bit values. For all the other interfaces all address fields are read as 32bit values.
4. If the cfg_rom_loc is SD/MMC, the IBR SD/MMC driver code will throw an error if the address is not aligned. For instance, the Source address must be aligned, and in case an address is constructed with the summation of base and offset the resultant address must be aligned. The alignment value is 0x200. i.e the address should be 512 byte aligned. The alignment requirement is only for SD/MMC.

Sample input file to generate CF Header is as follows –

```
--------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform - 1010/9131/9132/1040/C290
PLATFORM=C290
--------------------------------------------------
# Primary Image Header Address - [Mandatory]
ESBC_HDRADDR=CC002000
--------------------------------------------------
# Specify the file name of the keys seperated by comma.
# The number of files and key select should lie between 1 and 4 for 1040 and C290.
# For rest of the platforms only one key is required and key select should not be provided.

# USAGE (for 1010/913x): PRI_KEY = key1.pri
# USAGE (for 1040/C290): PRI_KEY = key1.pri, key2.pri, key3.pri, key4.pri

# PRI_KEY (Default private key :srk.pri) - [Optional]
PRI_KEY=srk.pri
# Please provide KEY_SELECT(between 1 to 4) (Required for 1040 and C290 only) - [Optional]
KEY_SELECT=
--------------------------------------------------
# Specify the target where image will be loaded. - [Mandatory]
# Select from - NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/NAND_8B_4K/NAND_16B_512/NAND_16B_2K/
NAND_16B_4K/SD/MMC/SPI
IMAGE_TARGET= NAND_8B_512
--------------------------------------------------
# Specify Configuration Words (Address Data Pairs), Max 1024 such pairs are supported.
# USAGE : CF_WORD = {ADDR, DATA}
CF_WORD=(0xff700d28 , 0x000ca000)
CF_WORD=(0xff700d30 , 0x80a00012)

--------------------------------------------------
# Specify the file names of cf header. (Default : cf_hdr.out) - [Optional]
OUTPUT_HDR_FILENAME=cf_hdr.out
--------------------------------------------------
# Primary Image Header Address (Required for C290 and 1040 only) - [Mandatory]
ESBC_HDRADDR_SEC_IMAGE=CFFFFFFC
-----------------------
```

NOTE

The right input file is picked as per the specified interface/platform and user can change the fields based on the requirement.

10.3.4.2.3 uni_sign

uni_sign tool can be used for the following functions :

- CSF header generation along with signature for both ISBC and ESBC phase
- CSF header generation without signature if private key is not provided

  uni_sign tool (with ESBC = 0 in input file) is used for creating signature and header over Boot1 image to be verified by ISBC

  uni_sign tool (with ESBC = 1 in input file) is used for creating signature and header over images to be verified by ESBC.

Options available with the uni_sign command are as follows:

Usage:

To view usage of tool:

```
./uni_sign
    --verbose    Display header Info after Creation
    --hash       Print the SRK(Public key) hash.
    --img_hash   Header is generated without Signature.
                 Image Hash is stored in a separate file.
    --help       Show the Help for Tool Usage.
```

Table 391. Description of Fields

Field	Field Description	Platform supported
PLATFORM	To identify the platform/SoC for which CF Header needs to be created.	All
ESBC	Don't set this flag when code signing is being performed on the image directly verified by the ISBC. For later images in the chain of trust, set this flag.	TA3.x
ENTRY_POINT	Entry Point address / Image start address field in the header.	All
PRI_KEY	Private key filename to be used for signing the image. (File has to be in PEM format) (default = srk.pri generated by gen_keys command) FILE1 [,FILE2, FILE3, FILE4]. Multiple key support for Trust Arch v2.x devices only.	All
PUB_KEY	Public key filename in PEM format. (default = srk.pub generated by gen_keys) FILE1 [,FILE2, FILE3, FILE4]. Multiple key support for Trust Arch v2.x devices only.	All
KEY_SELECT	Specify the key to be used in signature generation when more than one key has been given as input. (Default=1, first key will be selected)	All
IMAGE_1 - IMAGE_8	Create Entries for SG Table in the format { IMAGE_NAME, SRC_ADDR, DST_ADDR }	All
OEM_UID_x	OEM UID to be populated in the header.	All
FSL_UID_x	FSL UID to be populated in header.	All
HK_AREA_POINTER	House Keeping Area Starting Pointer Required by Sec (Required for Trust Arch v2.x devices only when esbc option is not provided)	TA2.x

Table continues on the next page...

Table 391. Description of Fields (continued)

Field	Field Description	Platform supported
HKAREA_SIZE	House Keeping Area Size (Required for Trust Arch v2.x devices only when esbc option is not provided)	TA2.x
OUTPUT_HDR_FILENAME	Name of the combined header binary to be created by tool	All
SG_TABLE_ADDR	Specify SG_TABLE Address where Scatter Gather table is present for 2041/3041/4080/5020/5040 when ESBC=0.	TA1.x
OUTPUT_SG_BIN	Specify the output file name of sg table.	TA1.x
IMAGE_TARGET	Specify the target where image will be loaded. Ex:NOR_8B/NOR_16B/ NAND_8B_512/NAND_8B_2K/NAND_8B_4K/ NAND_16B_512/ NAND_16B_2K/NAND_16B_4K/SD/MMC/SPI	All
SEC_IMG	Flag for Secondary Image. Required for Trust Arch v2.x devices only	TA2.x
MP_FLAG	Specify Manufacturing Protection Flag. Available for LS1 only.	All, only needed in ISBC phase
VERBOSE	Specify Verbose option. Contents of header generated will be printed.	All
IMAGE_HASH_FILENAME	used with '--img_hash' option (Name of file in which Image Hash is stored)	TA3.x
ISS_FLAG	Increment Security State Flag	TA3.x, only needed in ISBC phase
LW_FLAG	Leave Writeable Flag	TA3.x, only needed in ISBC phase
ESBC_HDRADDR	32 bit address where header generated shall be placed. Used to calculate IE key table address	TA3.x, only to be used in case of IE key extension feature usage
IE_KEY	comma separated list of files containing public keys(IE Keys)	TA3.x, only to be used in case of IE key extension feature usage
IE_REVOC	Comma seperated list of numbers that are to be revoked from IE Table	TA3.x, only to be used in case of IE key extension feature usage
IE_KEY_SEL	No of keys in IE table that is to be used to validate image	TA3.x, only to be used in case of IE key extension feature usage

Sample input files can be referred to, from input_files/uni_sign/I<platform>

For IE keys u can refer to input_files/uni_sign/l<platform>/ie_ke

TA3.x: LS2088 and LS1088. To know platforms under TA1.x and TA 2.x refer Trust Architecture and SFP Information on page 1996

10.3.4.2.3.1 Sample Input File

The input files will not have ESBC field (ESBC=0).

```
--------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
# TRUST 1.0, 1.1, 2.0, 2.1: 1010/1040/2041/3041/4080/5020/5040/9131/9132/9164/4240/C290/LS1
PLATFORM=LS2088
--------------------------------------------------
# Entry Point/Image start address field in the header.[Mandatory]
# (default=ADDRESS of first file specified in images)
# Address can be 64 bit
ENTRY_POINT=30008000
--------------------------------------------------
# Specify the Key Information.
# PUB_KEY [Mandatory] Comma Seperated List
# Usage: <srk1.pub> <srk2.pub> .....
PUB_KEY=srk.pub
# KEY_SELECT [Mandatory]
# USAGE (for TRUST 3.1): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Comma Seperated List for Signing
# USAGE: <srk.pri>, <srk2.pri>
PRI_KEY=srk.pri
--------------------------------------------------
# Specify IMAGE, Max 8 images are possible.
# DST_ADDR is required only for Non-PBL Platform. [Mandatory]
# USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
# Address can be 64 bit
IMAGE_1={u-boot.bin,30008000,ffffffff}
IMAGE_2={,,}
IMAGE_3={,,}
IMAGE_4={,,}
IMAGE_5={,,}
IMAGE_6={,,}
IMAGE_7={,,}
IMAGE_8={,,}
--------------------------------------------------
# Specify OEM AND FSL ID to be populated in header. [Optional]
# e.g FSL_UID_0=11111111
FSL_UID_0=
FSL_UID_1=
OEM_UID_0=
OEM_UID_1=
OEM_UID_2=
OEM_UID_3=
OEM_UID_4=
--------------------------------------------------
# Specify the output file names [Optional].
# Default Values chosen in Tool
OUTPUT_HDR_FILENAME=hdr_uboot.out
IMAGE_HASH_FILENAME=
```

```
RSA_SIGN_FILENAME=

------------------------------------------------
# Specify The Flags. (0 or 1) - [Optional]
MP_FLAG=0
ISS_FLAG=1
LW_FLAG=0

------------------------------------------------
# Specify VERBOSE as 1, if you want to Display Header Information [Optional]
VERBOSE=0
------------------------------------------------
# Following fields are Required for 4240/9164/1040/C290 only

# Specify House keeping Area
# Required for 42409164/1040/C290 only when ESBC flag is not set. [Mandatory]
HK_AREA_POINTER=
HK_AREA_SIZE=
------------------------------------------------
# Following field Required for 4240/9164/1040/C290 only
# Specify Secondary Image Flag. (0 or 1) - [Optional]
# (Default is 0)
SEC_IMAGE=
------------------------------------------------
# Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 - [Optional]
SG_TABLE_ADDR=
```

10.3.4.3 Signature generation

The tools in this category are provided in case the user does not want to share the Private Key with the CST Tool. The **--img_hash** option in Header creation on page 2121 Tools provides OEMs with the ability to perform code signing in a secure environment which does not run the NXP Code Signing Tool.

--img_hash option

- Generates hash file in binary format which contains SHA256 hash of the componenets required for signature.
- Generates output header binary file based on the fields specified in input file.
- Output header binary file doesn't contain signature.
- Provides flexibility to manually append signature at the end of output header file. User's can use their own custom tool to generate the signature. The signature offset chosen in the header is such that the signature can be appended at the end of the header file.
- This option does not require private key to be provided. But the corresponding Public key from the public/ private key pair must be provided to calculate correct SHA256 hash.
- The SHA256 hash generated over CF Header (in case of TA1.x platforms)) is then signed using RSA algorithm (OPENSSL APIs) with the private key . This encrypted hash is known as digital signature. This signature is placed at an offset from the CF header, which is later read by IBR.
- The SHA256 hash generated over CSF header, the Public Key, the S/G Table and the ESBC is also signed using RSA algorithm with the same private key . The signature generated is placed at an offset from the CSF header, which is again later read by IBR.

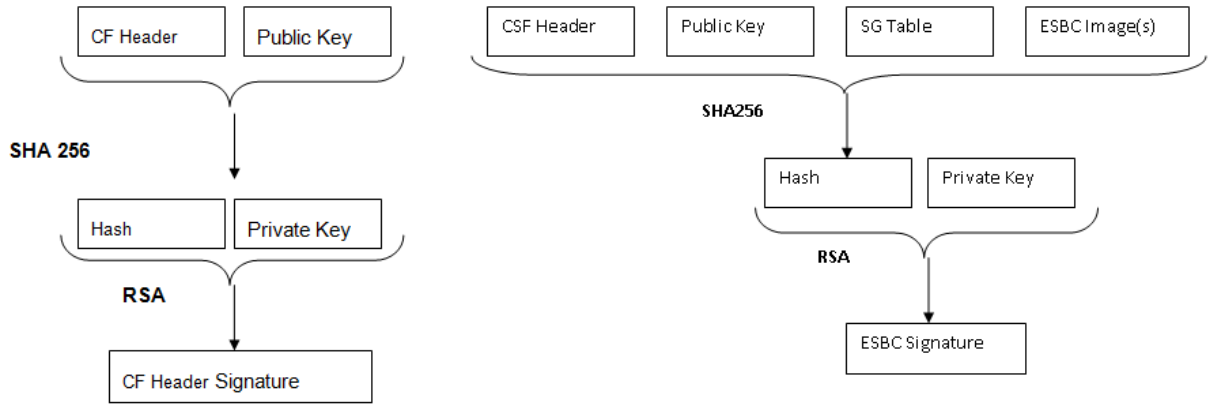


Figure 342. Dual signature generation

Usage Example

```
$ ./uni_sign --img_hash --verbose input_files/uni_sign/<platform>/input_uboot_nor_secure


    #-------------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #-------------------------------------------------------#

============================================================
This tool includes software developed by OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
============================================================

Input File is input_files/uni_sign/<platform>/input_uboot_nor_secure

-----------------------------------------------
-    Dumping the Header Fields
-----------------------------------------------
- SRK Information
-     SRK Offset : 200
-     Number of Keys : 1
-     Key Select : 1
-     Key List :
-        Key1 srk.pub(100)
- UID Information
-     UID Flags = 00
-     FSL UID = 00000000_00000000
-     OEM UID0 = 00000000
-     OEM UID1 = 00000000
-     OEM UID2 = 00000000
-     OEM UID3 = 00000000
-     OEM UID4 = 00000000
- FLAGS Information
-     MISC Flags = 60
-         ISS = 1
-         MP = 0
```

```
-         LW = 0
-         B01 = 1
- Image Information
-     SG Table Offset : 800
-     Number of entries : 1
-     Entry Point : 30008000
-     Entry 1 : u-boot.bin (Size = 000c0000 SRC = 30008000 DST = ffffffff)
- RSA Signature Information
-     RSA Offset : a00
-     RSA Size : 80
----------------------------------------------

Image Hash:
8588c174dd92f4a1b114b9029fc647e18cac4aaa46f03a6538ef20531e796e8f

***********************************************
* Image Hash Stored in File: hash.out
* Header File is w/o Signature appended
***********************************************

Header File Created: hdr_uboot.out

SRK (Public Key) Hash:
7df50d4256c4cbde4ef4ae9931042b1e44ff13aeb5107a7e0e9ee07e0fbfc236
     SFP SRKHR0 = 7df50d42
     SFP SRKHR1 = 56c4cbde
     SFP SRKHR2 = 4ef4ae99
     SFP SRKHR3 = 31042b1e
     SFP SRKHR4 = 44ff13ae
     SFP SRKHR5 = b5107a7e
     SFP SRKHR6 = 0e9ee07e
     SFP SRKHR7 = 0fbfc236
```

The tools are provided to create the signature file and embed the signature at end of header file.

10.3.4.3.1 gen_sign

This tool is provided for the user to calculate signature for a given hash using CST tool. The tool requires only the hash file and private key file from the user as input. It would generate signature file as output.

It uses RSA_sign API of openssl to calculate signature over hash provided.

Usage

./gen_sign [option] <HASH_FILE> <PRIV_KEY_FILE>

--sign_file SIGN_FILE	Provide file name for signature to be generated as operand. SIGN_FILE is generated containing signature calculated over hash provided through HASH_FILE using private key provided through PRIV_KEY_FILE. With this option HASH_FILE and PRIV_KEY_FILE are compulsory while SIGN_FILE is optional. SIGN_FILE default value is signout.
HASH_FILE	Name of hash file containing hash over signature needs to be calculated.
PRIV_KEY_FILE	Name of key file containing private key

Usage Example

After the hash file has been created as described in Signature generation on page 2129, the tool can be used as described below:

```
$ ./uni_sign --img_hash input_files/uni_sign/<platform>/input_uboot_nor_secure
.
.
.

*********************************************
* Image Hash Stored in File: hash.out
* Header File is w/o Signature appended
*********************************************

Header File Created: hdr_uboot.out

$ ./gen_sign hash.out srk.pri

    #-----------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #-----------------------------------------------------#
Signature Length = 80
Hash in hash.out is signed with srk.pri
Signature is stored in file : sign.out
```

10.3.4.3.2 sign_embed

This tool embeds signature in the header file generated using img_hash option which generates header but doesn't embed signature in the header. This option opens header file and copies signature at the end of the file.

The header file generated with 'img_hash' option has padding added till signature offset, so that signature can be directly embedded to the end of the file.

Usage

./sign_embed <hdr_file> <sign_file>

hdr_file	Name of header file in which signature needs to be embedded
sign_file	Name of sign file containing signature which needs to be embedded

Usage Example

```
$ ./sign_embed hdr_uboot.out sign.out

    #-----------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #-----------------------------------------------------#

hdr_uboot.out is appended with file sign.out (0x80)
```

NOTE

User can generate the complete header along with signature in a single step using uni_sign/uni_pbi tool without any option.

./uni_sign <input_file>

OR

User may wish to do it in three seperate steps:

1. ./uni_sign **--img_hash** <input_file> (Create header file without signature and store the Hash in a separte File)
2. ./gen_sign[44] [option] <HASH_FILE> <PRIV_KEY_FILE> (Sign the Image Hash using Private Key)
3. ./sign_embed <hdr_file> <sign_file> (Embed the signature at the end of header file)

[44] This may be done by user's own tool in case he doesn't want to share the Private Key with the CST Tool.

Chapter 11
Virtualization

11.1 Hypervisor

11.1.1 NXP Embedded Hypervisor Release Notes

NXP Embedded Hypervisor Release Notes

Software Version: 1.3

Date: 05/29/2014

Contents

1. Introduction
2. References
3. Changes
4. Limitations

---

1. Introduction

   This file documents the known limitations with the NXP Embedded Hypervisor version 1.3 release.

   Please refer to the Embedded Hypervisor Software User's Manual [2] and the Embedded Hypervisor Software Reference Manual [1] for complete details on the hypervisor software.

2. References

   a. NXP Embedded Hypervisor Software Reference Manual

   b. NXP Embedded Hypervisor Software User's Manual

3. Changes

   The following list summarizes changes in v1.3-009 of the NXP Embedded Hypervisor since release v1.3-008:

   - introducing "fast tlb1 feature" - the performance of TLB1 emulation is significantly improved by implementing a shorter code path for non-IPROT TLB1 entries. This is particularly useful for guests making heavy use of tlb1, for example when using indirect tlb1 entries. The feature is controlled by a Kconfig option and is enabled by default.
   - EPCR[DGTMI] (disable guest tlb management instructions) is now dynamically controlled
   - workaround A-006958 time-base erratum
   - workaround A-007907 l1 stash on e6500 erratum
   - workaround A-008007 t1040 pvr erratum
   - workaround A-00598 pamu erratum
   - workaround t1040 uart fifo erratum
   - other misc fixes

   The following list summarizes changes in v1.3-008 of the NXP Embedded Hypervisor since release v1.3-003:

- e6500 rev2 core support
- added emulation for PWRMGTCR0 introduced by e6500 rev2
- added Kconfig option for maximum number of supported partitions, defaulting to 24. Previous hard-coded limit was 8.
- fixed issue with guest tlb1 management structures state becoming out-of-sync. The issue triggered when a tlb1 entry was evicted.

The following list summarizes changes in v1.3-003 of the NXP Embedded Hypervisor since release v1.2-003:

- initial e6500 core support featuring MMUv2, LRAT, hardware page table walk
- initial multi-threading support on e6500 cores: each thread shows up in guest as a single threaded e6500 core
- added options in the HV config file to configure TLB behavior:
  - direct-guest-tlb-miss: tlb misses directed to the guest
  - direct-guest-tlb-mgt: tlb management instructions (tlbwe, tlbilx) are allowed in guest
- tlb0 caching is disabled on e6500 cores (support for this is not useful and would impact performance on other platforms)
- added support for the following T4xxx hw blocks: mpic, guts, ccf, cpc
- updated driver model to support multiple compatibles per driver
- added support for cache-able spin-table, epapr conformance
- added support to handle altivec exceptions
- reduced hypervisor's binary image footprint by ~400KB
- added shared l1 cache "shared resource synchronization" as defined and required in "e6500 Core RefMan, chapter 3.3.3.1"
- fixed a memory corruption happening on 64-bit targets
- fixed to work with newer gcc versions
  - r3 register was clobbered in bootstrap code thus the pointer to the device tree was lost
  - .notes section added in linker script
- fixed a long-standing l2 flush timeout issue on 64-bit targets
- many e6500 core related unit test updates
- libos: added a simple hardware threads api and example
- libos: introduced a cpu capability querying api

4. Limitations
   a. Boot Services
      - ELF file format-- the hypervisor image loader only supports ET_EXC as per the ePAPR. The ET_DYN flag is not supported.
   b. Configuration
      - The stashing may not be configured correctly if the hypervisor creates a binding between CPU and qman portal different than the one defined in the hardware device tree.
   c. Debug Services
      - Verbose log levels may not function correctly with all logging groups.
      - The GDB stub does not support using software breakpoints to debug multi-CPU partitions. 2 hardware breakpoints are supported.
      - The GDB stub does not support 'read' data breakpoints (rwatch command).

- In 'guest debug mode' debug stubs cannot share the debug resources (registers, interrupt) with the virtual CPU.
- The GDB stub does not support single stepping (stepi) over emulated instructions.

d. Tools

- the partman load command does not support compressed uImage files
- partman does not support loading images at guest physical addresses bigger than 4GB
- partman does not support loading images bigger than 3GB

11.1.2 NXP Embedded Hypervisor Software Reference Manual

11.1.2.1 Preface

About This Book

This document is a reference manual for the NXP Embedded Hypervisor software.

Audience

Developers or system architects that need to understand the interfaces and architecture of the NXP Embedded Hypervisor architecture.

Organization

A summary of the organization of this document is below:

- Introduction on page 2137 contains an introduction to the software architecture.
- The following sections describe the features and services of the hypervisor architecture available to a guest OS executing in a partition.
  - vcpu (Virtual CPU) on page 2139 describes the features of the vcpu relative to an physical CPU. The vcpu is the 'virtual' CPU in a partition upon which guest operating systems execute.
  - Partitions and partition management on page 2147 describes aspects of partitioning, including—the lifecycle of a partition, the initial state of a partition, how hardware resources are partitioned, guest device trees, and partition management.
  - ePAPR virtualization on page 2158 describes the hypervisor services compatible with the ePAPR virtualization standard. These services include—PIC, byte channels, and interpartition doorbells.
  - NXP hypervisor services on page 2171 describes NXP-specific hypervisor services available to guest operating systems. These services include—partition management, power management, and high availability services.
- The following sections describe additional features of the architecture
  - Configuration on page 2209 describes how the hypervisor is configured including the definition of partitions.
  - Debugging on page 2234 describes the debug interfaces to the hypervisor—the console and GDB stub interface.
  - Byte-Channel Serial Multiplexer Protocol on page 2237 describes the serial multiplexer protocol used by the byte-channel multiplexer.
  - Linux hypervisor management driver on page 2238 describes the hypervisor management driver API available for the Linux kernel.
  - Hypervisor Debug Stubs on page 2243 describes the programming interface available to create debug stubs that can be used to debug guest operating systems.
  - Hypervisor APIs on page 2245 describes hypervisor APIs available to debug stubs and other custom code added to the hypervisor
  - Hypervisor customization on page 2260 describes some hypervisor customization options for custom reset and system health monitoring

Conventions

This document uses the following conventions:

`Courier` is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

Italic is used to identify key words in device trees.

All source code examples are in Device Tree Syntax (DTS) form.

References

The following documents are referred to in this manual:

- Embedded Power Architecture Platform Requirements (ePAPR), version 1.1. power.org, 2011.
- Power ISA 2.06 Revison B, power.org
- EREF 2.0: A programer's Reference Manual for NXP Power Architecture Processors
- e500mc Core Reference Manual: Programming Model (E500MCRM)
- e5500 Core Reference Manual
- e6500 Core Reference Manual
- P4080 QorIQ Integrated Host Processor Reference Manual
- NXP Power Architecture Book E Virtual CPU Specification
- Debugging with gdb, the gnu Source-Level Debugger, Ninth Edition, for gdb version 6.7.50.20080119

11.1.2.2 Introduction

11.1.2.2.1 Overview

The NXP embedded hypervisor is a layer of software that enables the efficient and secure partitioning of a multicore system. A system's CPUs, memory, and I/O devices can be divided into groupings or *partitions*, as shown in the figure below. Each partition is capable of executing a guest operating system.

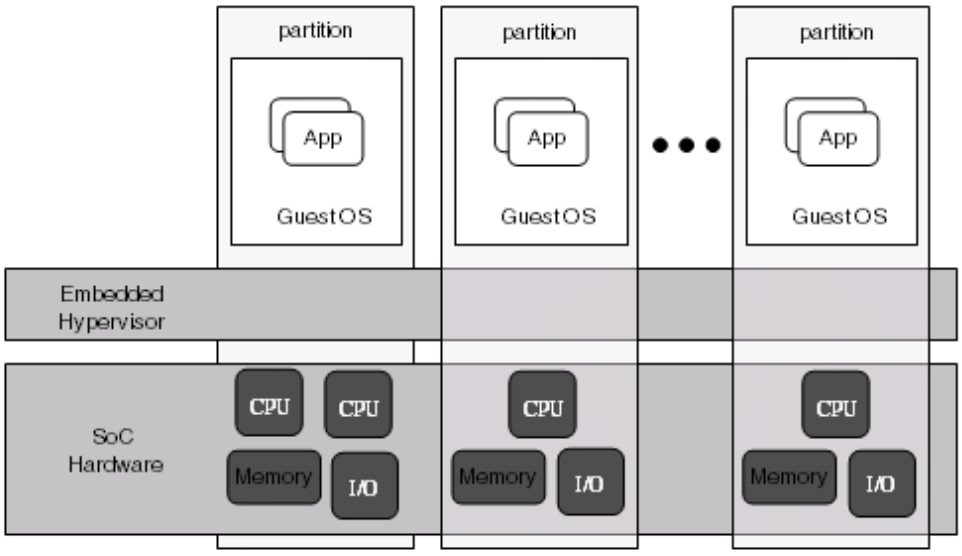


Figure 343. Partitioning with the Hypervisor in a Multicore Environment

Key features of the hypervisor architecture are summarized as follows:

- Partitioning. Support for partitioning of CPUs, memory, and I/O devices:
  - CPUs. Each partition is assigned one or more CPU cores in the system.
  - Memory. Each partition has a private memory region that is only accessible to the partition that is assigned the memory. In addition, shared memory regions can be created and shared among multiple partitions.
  - I/O devices. I/O devices may be assigned directly to a partition (Direct I/O), making the device a private resource of the partition, and providing optimal performance.
- Protection and isolation. The hypervisor provides complete isolation of partitions, so that one partition cannot access the private resources of another. The PAMU (an IOMMU) is owned and managed by the hypervisor to ensure device-to-memory accesses are constrained to allowed memory regions only.
- Sharing. Mechanisms are provided to selectively enable partitions to share certain hardware resources (such as memory).
- Virtualization. Support for mechanisms that enable the sharing of certain devices among partitions such as the system interrupt controller.
- Performance. The hypervisor uses the features of category Embedded.Hypervisor as defined by the Power ISA to provide security and isolation with very low overhead. Guest OSes take external interrupts directly without hypervisor involvement providing very low interrupt latency.
- Ease of migration. The hypervisor architecture uses a combination full emulation and para-virtualization to maintain high performance while requiring minimal guest OS changes when migrating code from an phyisical CPU to running as a guest under the hypervisor on the vcpu.

11.1.2.2.2 Software Architecture Overview

The hypervisor provides the capability to create partitions in which guest software can execute in isolation without the possibility of interference from other partitions in the system.

The hypervisor architecture consists of the following key components:

- Virtual CPU (vcpu)—Software executing in a partition under the hypervisor sees a CPU that is nearly identical to the physical CPU, except for some CPU state and features which are hypervisor privileged and not accessible.
- Hypercall services—Some services are provided via hypercalls, including: the interrupt controller, byte-channels, interpartition doorbells, IOMMUs, and partition management.
- Boot services—The hypervisor adopts the device tree and multicore boot architecture of the power.org ePAPR specification. This defines mechanisms for resource discovery within a partition and for starting multiple CPUs in a single partition.
- Debug services—Guest software that has its own debug stub can use the stub without change through the capabilities of the vcpu. In addition, the hypervisor provides a gdb stub in the hypervisor that provides an interface for a host debugger to use to debug guest operating systems.
- Direct I/O—High-speed peripherals are not virtualized and are assigned and managed directly by partitions. External interrupts are taken by partitions directly with no hypervisor involvement.

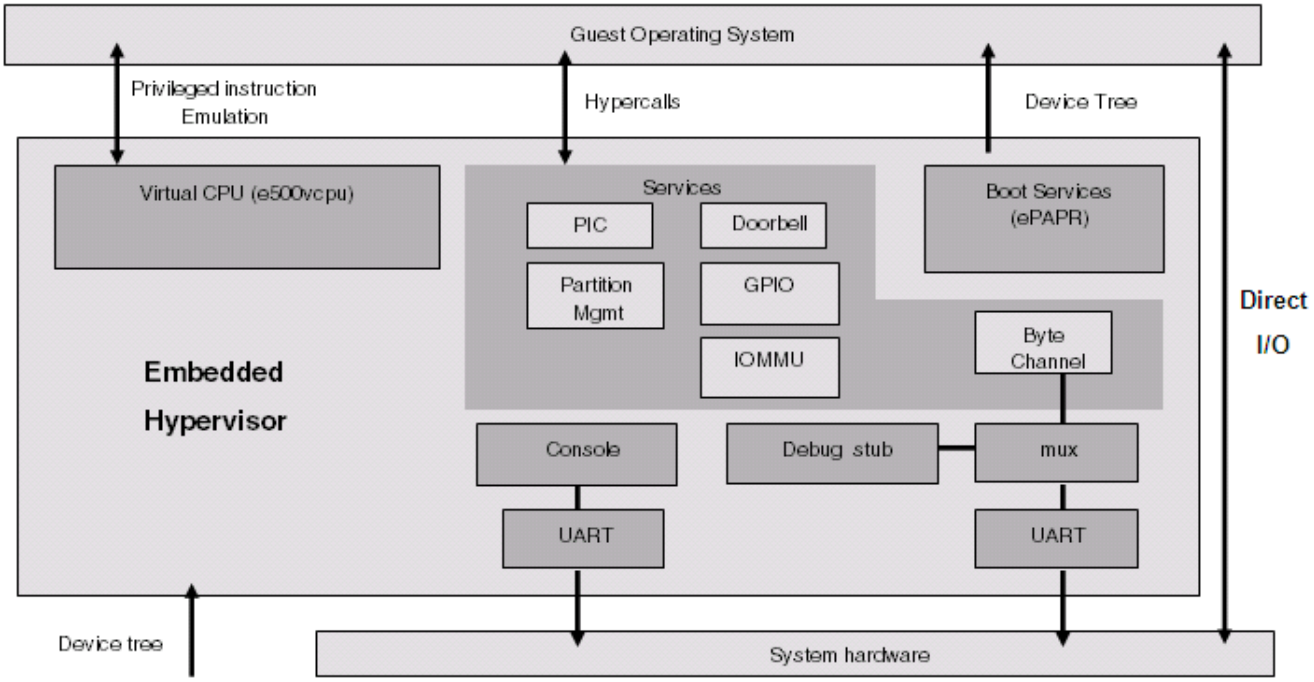


Figure 344. Hypervisor Block Diagram

11.1.2.3 vcpu (Virtual CPU)

11.1.2.3.1 vcpu Overview

Virtualization enables multiple operating systems to run on a system, each in their own isolated virtual machine. The Hypervisor creates and manages the virtual machines, one part of which is a virtual CPU (or vcpu). The vcpu emulates a physical CPU and the behavior of instructions, registers, and exceptions on the vcpu is nearly identical to the physical CPU being emulated.

There are a few minor architectural and programming model differences between the vcpu and the following physical CPUs:

- e500mc
- e5500
- e6500

For a description of the differences between vcpu and emulated CPU please review the *[6] NXP Power Architecture Book E Virtual CPU Specification*

This chapter describes the NXP Embedded Hypervisor specific considerations.

11.1.2.3.2 NXP Embedded Hypervisor Specific Considerations

11.1.2.3.2.1 vcpu configuration modes

A partition's configuration (see Partition Node Properties on page 2215) may specify configurable options for vcpus. This section describes several configurable modes available.

11.1.2.3.2.1.1 Guest Cache Lock Mode

The behavior of cache locking instructions (dcbtls, dcbtstls, dcblc, icbtls, icblc) is dependent on the setting of guest cache lock mode. When guest cache lock mode is enabled (the default), cache locking instructions are permitted to execute normally. If guest cache lock mode is disabled, executing cache-locking instructions is effectively a nop; the operation is ignored.

Guest cache lock mode is configured in the definition of a partition (see Partition Node Properties on page 2215) and guest software can determine the configuration of this mode from the guest device tree (see Hypervisor Node Properties on page 2153).

11.1.2.3.2.1.2 Guest Debug Mode

The availability of the debug resources (debug interrupt, IACx, DACx, DBCRx) of the vcpu is determined by the setting of guest debug mode. By default, guest debug mode is enabled.

When using a hypervisor-resident debug stub, guest debug mode can be disabled in a partition's configuration (see Partition Node Properties). Guest software can determine the configuration of this mode from the guest device tree (see Hypervisor Node Properties on page 2153) or by examining the DBCR0[EDM].

11.1.2.3.2.2 Registers

This section describes differences between the register model of the physical CPU and that of the vcpu.

11.1.2.3.2.2.1 Debug Registers

By default guest debug mode is enabled (see Guest Debug Mode on page 2140). When guest debug mode is disabled accesses to CPU debug registers are boundedly undefined.

In guest debug mode reads from all debug registers are supported.

11.1.2.3.2.2.2 Removed Registers

The registers listed in the table below are removed from the vcpu. Attempting accesses to one of these registers yields boundedly undefined results.

Table 392. Removed Registers

SPR Number	Name	Description
311	MSRP	MSR protect register
339	MAS5	MMU assist register 5
341	MAS8	MMU assist register 8
306	DBSRWR	Debug status register write register
307	EPCR	Embedded hypervisor control and status Only if 64-bit category is not supported.
338	LPIDR	logical partition id Register
432	IVOR38	Interrupt vector offset register 38. Guest processor doorbell
433	IVOR39	Interrupt vector offset register 39. Guest processor doorbell critical
434	IVOR40	Interrupt vector offset register 40. Embedded Hypervisor system call
435	IVOR41	Interrupt vector offset register 41. Embedded Hypervisor privilege
436	IVOR42	Interrupt vector offset register 42. LRAT error interrupt

Table continues on the next page...

Table 392. Removed Registers (continued)

SPR Number	Name	Description
447	GIVPR	Guest interrupt vector prefix register
382	GPIR	Guest processor identification register
440	GIVOR2	Guest interrupt vector offset register 2. Guest data storage interrupt.
441	GIVOR3	Guest interrupt vector offset register 3. Guest instruction storage interrupt.
442	GIVOR4	Guest interrupt vector offset register 4. Guest external input.
443	GIVOR8	Guest interrupt vector offset register 8. Guest system call interrupt.
444	GIVOR13	Guest interrupt vector offset register 13. Guest data TLB error.
445	GIVOR14	Guest interrupt vector offset register 14. Guest instruction TLB error.
383	GESR	Guest exception syndrome register
378	GSRR0	Guest save/restore register 0
379	GSRR1	Guest save/restore register 1
381	GDEAR	Guest DEAR
380	GEPR	Guest EPR
368-371	GSPRG0-3	Guest SPR general 0-3

If guest debug mode is disabled, the register below does not exist and attempted accesses yield boundedly undefined results.

Table 393. Registers Absent when Guest Debug Mode is Disabled

SPR Number	Name	Description
415	IVOR15	Interrupt vector offset register 15. Debug

11.1.2.3.2.3 Instruction model differences

The set of instructions and their behavior on the vcpu is identical to the physical CPU except as described in this section.

11.1.2.3.2.3.1 Instruction Model Overview

The vcpu supports four interrupt levels like the physical CPU.

- Noncritical
  - System call
  - Program
  - Alignment
  - External input
  - ISI

  - DSI
  - Decrementer
  - Fixed interval timer
  - Data TLB error
  - Instruction TLB error
  - Doorbell
  - Floating point unavailable
  - Performance monitor
- Critical
  - Watchdog
- Machine check
  - Machine check
- Debug
  - Debug

All interrupt definition and exception behavior in the vcpu is identical to the physical CPU except as described below:

- The following interrupts are hypervisor-only resources and do not exist in the vcpu:
  - Guest processor doorbell
    - Result of access to IVOR38 is boundedly undefined
  - Guest processor doorbell critical
    - Result of access to IVOR39 is boundedly undefined
  - Embedded Hypervisor system call
    - Result of access to IVOR40 is boundedly undefined
  - Embedded Hypervisor privilege
    - Result of access to IVOR41 is boundedly undefined
  - LRAT error
    - Result of access to IVOR42 is boundly undefined
- The existence of the debug interrupt depends on a configurable mode of the vcpu:
  - The vcpu's guest debug mode (see Guest Debug Mode on page 2140) enables and disables debug interrupts
  - MSR[DE] is read-only if guest debug mode (see Guest Debug Mode on page 2140) is disabled.
- A memory access to physical address location that is invalid or does not exist results in a machine check interrupt at the vcpu.

11.1.2.3.2.4 Memory management

11.1.2.3.2.4.1 Memory Management Overview

All memory management in the vcpu is identical to the physical CPU except for the following:

- Unlike the physical CPU, which uses a 48-bit virtual address (e500mc), 80-bit virtual address (e5500) or 86-bit virtual address(e6500) the vcpu address translation uses a 41-bit virtual address on e500mc, 73-bit virtual address on e5500 and 79-bit virtual address on e6500. The TLPID and TGS fields in the physical CPU virtual address do not exist in the vcpu.

- The MAS5 (MMU Assist 5) register defines TLB search values and is accessible only by the hypervisor. Attempted access by guest software are boundedly undefined.
- The MAS8 (MMU Assist 8) register defines state for TLB accesses and is accessible only by the hypervisor. Attempted access by guest software are boundedly undefined.
- The EGS and ELPID fields in EPLC and EPSC are not writeable.
- The NENTRY field in TLB1CFG may report a smaller number of TLB entries compared to the hardware
- The NENTRY field in TLB0CFG is zero—indicating that number of entries in TLB0 is implementation defined and not visibible to software.
- The width of the LPID as reported by MMUCFG is zero.
- Using the **tlbre** instruction to read TLB entries from TLB0 is boundedly undefined.
- Execute-only TLB mappings (not readable, not writeable) as not supported

Programming Notes

- The use of the **tlbivax** instruction is deprecated—it is recommended that **tlbilx** be used instead. This requires guest OSes to use intercore signaling to synchronize TLB invalidates, but avoid the need for using **tlbsync**.
- It is recommended that tlbilx be used for all invalidations of TLB0. Using **tlbwe** to clear a TLB entry's valid bit may have additional performance costs depending on the hypervisor software implementation.
- **tlbsync**—guest OS software is not required to take steps to synchronize **tlbsync** instructions across cores or partitions. The vcpu handles synchronization to ensure that only one **tlbsync** is in flight in the platform.
- The timing of the TLB management instructions may be different compared to the physical CPU.
- Note: the vcpu may internally cache more virtual-to-physical mappings than exist in the hardware TLB. Software should not make any assumptions about the number TLB mappings present in the vcpu. Invalidation of a TLB entry should be explicit.
- Note: If guest tlb management instructions are allowed tlbilx cannot be used to invalidate TLB1 entries. In this case tlbilx is executed in guest state, but all the TLB1 entries are invalidation protected (configured as such by the Hypervisor).

11.1.2.3.2.4.2 Guest TLB management

e6500 core allows guest TLB management instructions. In this case tlbwe and tlbilx are executed directly by the guest. Two partition configuration options allow for different configuration options:

- direct-guest-tlb-management - if present indicates that tlbwe and tlbilx can be executed in guest state
- direct-guest-tlb-miss - if present indicates that the TLB miss interrupts are taken directly by the guest

These two options can be enabled in various combinations:

Table 394. Guest TLB management options

direct-guest-tlb-management	direct-guest-tlb-miss	Comments
0	0	• provides the highest level of architectural correctenss • LRAT will still be used in this mode if the guest is using hardware page table walk • tlbilx can be used to invalidate TLB1 • no restriction on physical contiguity of guest memory

Table continues on the next page...

Table 394. Guest TLB management options (continued)

1	0	• tlbilx cannot be used to invalidate TLB1 • no restriction on physical contiguity of guest memory (e.g. TLB1 entries could be backed by smaller pages) • no HV mediation for TLB write unless an LRAT miss occur
1	1	• tlbilx cannot be used to invalidate TLB1 • restrictions on physical contiguity of guest memory • no HV mediation for TLB write unless an LRAT miss occur
0	1	• tlbilx can be used to invalidate TLB1 • restrictions on physical contiguity of guest memory • LRAT will still be used in this mode if the guest is using hardware page table walk

11.1.2.3.2.4.3 Invalid MMU Mappings

The hypervisor validates all physical address mapped by guest software. If guest software maps a physical address that does not belong to its partition the behavior is as follows:

- The **tlbwe** instruction completes normally and writes the mapping into the TLB
- The entry can be read with a **tlbre** instruction
- When guest software accesses the invalid mapping a machine check occurs

For page table walk originated bad mappings a machine check is reflected to the guest when the attempt to add in TLB a bad mapping is made by the hardware. In this case the entry is not added in the TLB.

11.1.2.3.2.4.4 Memory Management Address Translation

The vcpu MMU translates effective addresses generated by loads, stores, and instruction fetches into real addresses (used for memory bus accesses) using an interim virtual address as indicated by the following table.

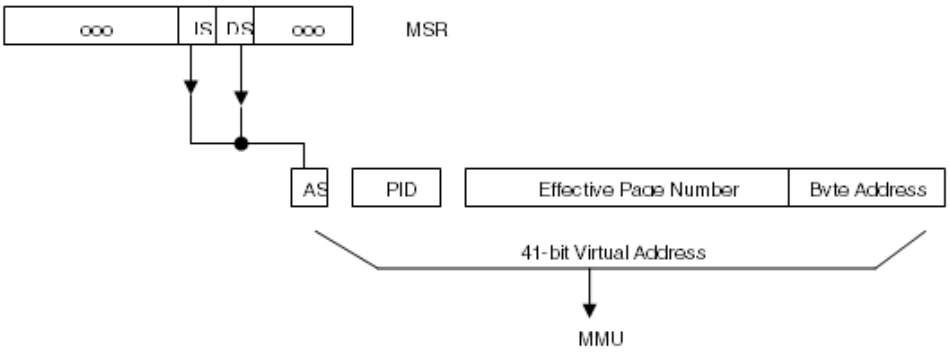
Table 395. Address Translation

	Effective address	Real address	Virtual address
e500mc	32	36	41
e5500	64	36	73
e6500	64	40	79

Address translation in the vcpu architecture starts with an effective address that is prepended with an address space (AS) value and a process ID (PID) to construct a virtual address (VA). The virtual address is then translated into a real address based on the translation information found in the on-chip TLB. As shown in the figure below, the AS bit for the access is selected from the value of MSR[IS] or MSR[DS], for instruction or data accesses, respectively.

The GS (guest state) and LPID (logical partition id) components of an physical CPU virtual address are not supported.

Figure 345. Address Translation



11.1.2.3.2.5 Timers

11.1.2.3.2.5.1 Timebase

The processor timebase (TBU/TBL) is automatically synchronized by the hypervisor across all physical CPUs managed by the hypervisor.

Unlike the physical CPU, in the vcpu TBU/TBL are not writeable. An attempt to write TBU/TBL is boundedly undefined.

11.1.2.3.2.5.2 Decrementer

The decrementer register and exception are identical to the physical CPU —a decrementer interrupt will occur when a decrementer exception exists and MSR[EE] = 1.

11.1.2.3.2.5.3 Fixed Interval Timer

The fixed interval timer configuration and behavior is identical to the physical CPU —a fixed interval timer interrupt will occur when a fixed interval timer exception exists and MSR[EE] = 1.

11.1.2.3.2.5.4 Performance Monitor

The performance monitor registers (PMRs) and performance monitor interrupt are identical to the physical CPU with the following caveat:

It is boundedly undefined if the guest alters MSR[PMM] when a perf interrupt is pending, including while the interrupt is being serviced, before the condition has been cleared.

11.1.2.3.2.5.5 Watchdog Timer

The vcpu provides a standard e500 watchdog timer:

- TCR[WPEXT] is concatenated with TCR[WP] to select a bit that triggers the watchdog timer
- TCR[WIE] enables watchdog timer interrupts
- The TSR[ENW] and TSR[WIS] bits behave normally—on first watchdog expiration a watchdog interrupt is generated (gated by MSR[CE] and TCR[WIE]). On the second time out TCR[WRC] governs the behavior.
- TSR[WRS] reflects TCR[WRC] on a watchdog generated reset.

The behavior of TCR[WRC] is different on the virtual CPU than on the physical CPU .

vcpu WRC definition:

- 00 No watchdog reset will occur
- 01 Partition reset (may be overridden by *watchdog-timeout*)
- 10 Partition reset (may be overridden by *watchdog-timeout*)

- 11 Partition reset (may be overridden by *watchdog-timeout*)

NOTE

one virtual CPU's watchdog expiration results in the entire partition being reset, not just the vcpu that timed out.

When a partition reset occurs, the virtual CPU that caused the watchdog reset will have its TSR[WRS] set to reflect TCR[WRC].

The action taken on the second watchdog timeout is governed by the value of WRC and the property *watchdog-timeout* on the partition node defining a partition (see Partition Node Properties on page 2215). A WRC nonzero value causes the current partition to reset unless the *watchdog-timeout* property is specified. See Partition Node Properties on page 2215 for allowable policies for *watchdog-timeout*.

11.1.2.3.2.6 Debug

All processor core debug features, including debug exception and interrupts are available unless guest debug mode is disabled (see Guest Debug Mode on page 2140).

MSR[DE] is read-only if guest debug mode is disabled, otherwise MSR[DE] is writeable and enables debug interrupts.

The vcpu supports a subset of the Power ISA embedded debug exceptions:

- Trap
- Instruction address compare
- Data address compare
- Instruction complete
- Branch taken

The following e500mc debug events are not supported:

- Return from interrupt
- Interrupt taken
- Unconditional debug event

The debug interrupt and supported debug function in a manner identical to the physical CPU.

11.1.2.3.2.7 Hardware Threads

On multi-threaded cores, the Hypervisor exposes the vcpu to guests as a single-threaded core. The thread management instructions (mftmr, mttmr), SPRs and TMRs are emulated by the Hypervisor: reads behave as if the core is single-threaded while writes are unsupported and trigger a warning in the Hypervisor console. Tables below list the supported SPRs and TMRs and the results of reads and writes.

Table 396. Supported SPRs

SPR Number	SPR	Read	Write
437	TENSR	1	Warning
438	TENS	1	Warning
439	TENC	1	Warning
446	TIR	0	Warning

Table 397. Supported TMRs

TMR Number	TMR	Read	Write
16	TMCFG0	0101h	Warning
288	IMSR0	Warning	Warning
289	IMSR1	Warning	Warning
320	INIA0	Warning	Warning
321	INIA1	Warning	Warning

11.1.2.4 Partitions and partition management

11.1.2.4.1 Partitions and Partition Management Introduction

A partition is a logical grouping of physical and virtual resources in which an operating system can securely run.

- Physical resources in a partition include CPUs, memory, and I/O devices.
- Virtual resources include hypervisor provided services such as byte-channels, interpartition doorbells, and interrupt controller services.

A partition generally is in one of two primary states:

- Stopped—the state of a partition after it has been created, exited, or stopped. A *stopped* partition is initialized and resources allocated to it, but is not running. A *stopped* partition can be started.
- Running—the state after control has been transferred to the partition on the partition's boot CPU. A running partition can be stopped or restarted.

In addition, two transitory states (starting, stopping) exist when a partition is being created or transitioning between the two primary states. In the *starting* and *stopping* states, no partition management actions are permitted.

The figure below shows the valid states of a partition and the actions that result in a state transition.

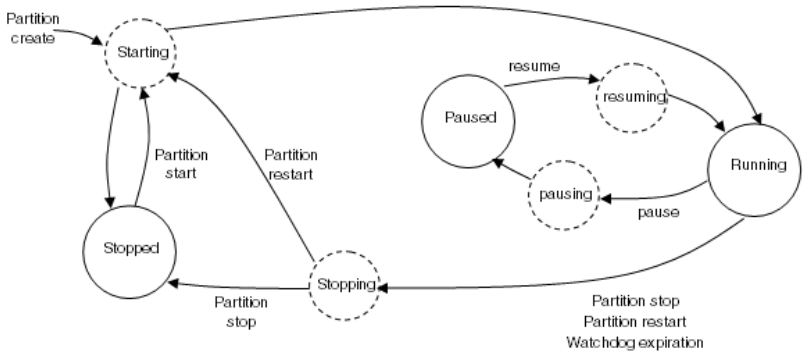


Figure 346. Partition State Diagram

A *paused* state also exists for debugging purposes. A paused partition executes no instructions and can take no interrupts. Pause and resume of partitions can be done by using the hypervisor's command shell.

11.1.2.4.2 Hardware and Guest Device Trees

The hypervisor is passed an ePAPR-compliant hardware device tree (from boot firmware, such as u-boot) when it is booted that describes all the hardware in the system.

A separate binary blob of configuration information is passed by address to the hypervisor which contains all hypervisor configuration parameters and partition definition information. This information is referred to as the "hypervisor configuration tree". Based on the hypervisor configuration tree the hypervisor will create partitions. The hypervisor will dynamically create guest device trees for each partition.Guest device trees describe the CPUs, memory, I/O devices, and other virtual resources that below to the partition.

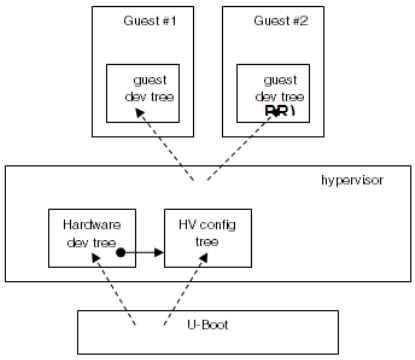


Figure 347. Device Tree Overview

Hypervisor configuration and the definition of the hypervisor configuration tree is described in Configuration on page 2209.

Guest device trees are described in detail in Partitions and partition management on page 2147 and in ePAPR virtualization on page 2158

11.1.2.4.3 Partition Physical Addresses (Guest Physical)

Guest software executing in a partition sees a physical address space created by the hypervisor that may not correspond directly to true physical addresses (real addresses in the Power Architecture) in the system hardware. These addresses are referred to as guest physical addresses (logical addresses in the Power Architecture). For example, each partition may have memory at guest physical address 0x0—where in each partition address 0x0 that corresponds to a unique true physical address.

This mapping of guest physical to true physical is specified in the definition of a partition and is maintained by the hypervisor and is completely transparent to guest software. Guest software becomes aware of the guest physical address ranges available to it via the guest device tree passed to the partition at initialization.

11.1.2.4.4 Hardware Resource Partitioning

Each hardware resource or I/O device is allocated in one of several ways:

- Direct access—The resource is directly accessible by guest software without hypervisor intervention. In the case of I/O devices, this means the device is assigned to a partition and becomes a private resource of the partition.
- Hypervisor-managed access—The hypervisor provides a service (via hcall or emulation) that enables partitions to access the resource

The table below identifies both CPU and I/O device resources on the SoC and the type of allocation.

Table 398. CPU and I/O Partitioning

Hardware Feature	Direct Access	Provided by hcall	Provided by Emulation	Hypervisor Only
CPU Resource				

Table continues on the next page...

Table 398. CPU and I/O Partitioning (continued)

Hardware Feature	Direct Access	Provided by hcall	Provided by Emulation	Hypervisor Only
MSR	X			
DEC,DECAR			X	
FIT			X	
Watchdog			X	
Timebase read	X			
Timebase write				X
Performance monitor	X9			
Debug registers (DBCR*n*, IAC*n*, DAC*n*)			X1	X2
Hardware implementation dependent registers				X
Intercore signaling (**msgsnd/msgclr**)			X	
L1 and L2 cache control			X	F
Cache locking	X8			X
PIR read	X			
PIR write				X
SPRG0—SPRG7, SPRG9	X			
SPRG8			X	
SRR0—SRR1	X			
CSRR0—CSRR1,MCSRR0-MCSRR1, DSRR0—DSRR1			X	
Exception registers (DEAR, EPR, ESR, XER)	X			
HID0			X10	X10
MMU	X11		X	
MMU assist registers	X			
TLB read			X	
TLB write	X11		X	

Table continues on the next page...

Table 398. CPU and I/O Partitioning (continued)

Hardware Feature	Direct Access	Provided by hcall	Provided by Emulation	Hypervisor Only
Interrupts/Exceptions				
External input interrupts	X			
TLB miss (data/instruction)			X	
DSI	X			
System calls	X			
all other interrupts/exceptions			X	
SoC Platform Resources				
MPIC		X		
DUART	X5	X6		
I2C	X		X3	
USB	X			
SD/MMC	X			
Local bus controller	X			
DMA controllers	X			
PCI-Express	X			
Serial Rapid I/O	X			
Datapath Peripherals Buffer Manager Portals Queue Manager Portals 10 Gigabit and 1 Gigabit Ethernet controllers SEC PME	X			
Global Utilities				X4
DDR SDRAM Controllers				X4

NOTE

1. In guest debug mode, guest software manages the CPU debug resources with the hypervisor providing the resources via emulation.
2. When guest debug mode is disabled, the hypervisor owns the CPU debug resources and they are not accessible by guest software
3. The I2C controllers can be configured to be owned by a single partition for direct access. Each controller can be assigned to a different partition with an emulated service provided by the hypervisor.
4. A recommended configuration. The system architect is free to assign any region of CCSR space to a partition through the partition's device tree.
5. In a typical configuration one DUART is allocated to the hypervisor for console and byte-channel services. Any remaining DUARTs are available for direct assignment to a partition.
6. Byte-channel services provide virtualized character I/O capabilities.
7. Some L1 and L2 cache control features are hypervisor only, others are provided by emulation.
8. The ability for guests to perform cache-locking operations is dependent on whether guest cache lock mode is enabled.
9. Guest software gets direct access to the performance monitor resources. However, due to some hardware quirks the hypervisor does trap/emulate accesses to the performance monitor registers while a perfmon interrupt is pending.
10. Certain HID0 bits are accessible to guests, and others are hypervisor only.
11. e6500 allows direct tlb management instructions. In this case tlbwe and tlbilx will be directly executed by the guest.

11.1.2.4.5 Guest device trees

11.1.2.4.5.1 Guest Device Trees Overview

Partitions discover the hardware and virtual resources available to them through a guest device tree loaded into their memory at partition initialization. The physical and logical structure of the device tree complies with the ePAPR specification (power.org's standard for *Embedded Power Architecture Platform Requirements, 1.0*).

All guest device trees will have the following nodes:

- A root node
- A cpus node with one or more child CPU nodes as defined in the ePAPR. The CPU nodes are standard for the underlying hardware and are described in the binding documentation for the physical CPU.
- A interrupt controller node if the partition has interrupt generating I/O devices.
- A memory node as defined in the ePAPR. Specifies the physical address and size of memory belonging to the partition.
- A hypervisor node

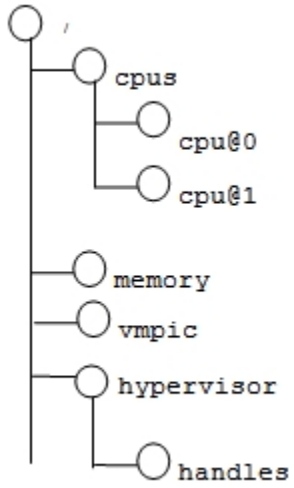


Figure 348. Figure 3-4. CPU, Memory, and Hypervisor Nodes Example

The guest device tree may have other nodes representing physical or virtual resources.

All physical addresses in the guest device tree are guest physical addresses (see Guest device trees on page 2151)

11.1.2.4.5.2 Root Node

The root node of a guest device tree has the properties listed in the table below.

Table 399. Root Node Properties

Property Name	Usage	Value Type	Definition
#address-cells	R	<u32>	Must be 2. Specifies the number of <u32> cells to represent the address in the *reg* property in *children* of root.
#size-cells	R	<u32>	Must be 2. Specifies the number of <u32> cells to represent the size in the *reg* property in *children* of root.
compatible	R	<stringlist>	Value consists of the compatible string list from the hardware device tree with the string "-hv" appended.
epapr-version	R	<string>	Must be epapr-version = "ePAPR-1.0"
label	R	<string>	Text label describing the partition. This value is propagated from the *label* property on the partition node that defines the partition.
Usage legend: R = Required, O = Optional, OR = Optional but recommended, SD = See definition			

11.1.2.4.5.3 CPU node

CPU nodes are as defined by the ePAPR.

11.1.2.4.5.4 Memory node

Memory nodes are as defined by the ePAPR.

11.1.2.4.5.5 Interrupts and the VMPIC Node

If a partition has interrupt generating I/O devices it will have an ePAPR virtual interrupt controller node.

11.1.2.4.5.6 Guest Device Trees Hypervisor Node

11.1.2.4.5.6.1 Hypervisor Node Properties

Each guest device tree has a hypervisor node located at the root of the tree. The following table defines properties that may be present on the hypervisor node.

Table 400. Hypervisor Node Properties

Property Name	Usage	Value Type	Definition
fsl,hv-version	R	<prop-encoded-array>	Specifies the version of the hypervisor. The value is a prop-encoded-array consisting of 4 cells defined as follows: <*major-version, minor-version, hcall-version, hcall-last-comp-version*> Where: • *major-version* is a <u32> value specifying the major version number of the hypervisor software (e.g. for version 1.02 major version is 1) • *minor-version* is a <u32> value specifying the minor version number of the hypervisor software • *hcall-version* is a <u32> identifying the version of the hcall interface. • *hcall-last-comp-version* is a <u32> identifying the lowest version number of the hcall interface which the hypervisor is backwards compatible with
fsl,hv-guest-cache-lock-disable	O	<none>	If present defines that guest cache lock mode is disabled.
fsl,hv-guest-debug-disable	O	<none>	If present defines that guest debug mode is disabled.
fsl,hv-pic-legacy	O	<none>	If present specifies that the partition uses the legacy interface between the CPU and interrupt controller (i.e., does **not** use the interrupt proxy functionality and Coreint interface).
fsl,hv-sys-reset-status	R	<string>	The *fsl,hv-sys-reset-status* property is set on the /hypervisor node of the guest device to convey to a partition the most recent reason the system hardware was reset. The values below define the hardware reset status—i.e. why the system hardware was reset. One of the hardware reset value will always be set. "**power-on-reset**"—The partition was reset due to a power on reset of the system hardware "**hard-reset**"—The partition was reset due to a hard reset of the system hardware

Table continues on the next page...

Table 400. Hypervisor Node Properties (continued)

Property Name	Usage	Value Type	Definition
fsl,hv-reason-stopped	R	<string>	The *fsl,hv-reason-stopped* property is set on the /hypervisor node of the guest device to convey to a partition the most recent reason the partition was stopped. Valid values are described below: **"stop"**—The partition was stopped due to a stop request (via hcall or shell). **"restart"**—The partition was stopped and restarted due to a restart request. **"watchdog"**—The partition was stopped and restarted due to a watchdog expiration. This property will not be present until a partition has been stopped/restarted for the first time.
fsl,hv-stopped-by	R	<string>	The value defines one string which identifies what component was responsible for the reason stopped set in the fsl,hv-reason-stopped property. Valid values are described below: **"self"**—The partition was stopped/restarted by the partition itself. **"manager"**—The partition was stopped/restarted by a partition manager. **"shell"**—The partition was stopped/restarted by the hypervisor command shell. This property will not be present until a partition has been stopped/restarted for the first time.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.4.5.6.2 Handles

A handles node exists under the /hypervisor node to represent virtual resources made available to the guest via a handle. For example:

- Byte-channels. See Byte-channel Services on page 2167 for a description of how byte-channels are represented in guest device trees.
- Interpartition doorbells. See Inter-partition doorbells on page 2169 for a description of how interpartition doorbells are represented in guest device trees.
- Partition nodes. Partition can be configured to perform partition management tasks on other partitions. A partition node provides a handle to managed partitions. See Partition Management Guest Device Tree Representation on page 2176 for a description of how managed partitions are represented in guest device trees.

11.1.2.4.6 Partition Creation

Partitions are defined using the hypervisor configuration tree—this is a tree data structure that describes the CPUs, memory, hardware I/O devices, and virtual resources that define the partition. See Partition definition on page 2215 for details on how partitions are defined.

Partitions are created during hypervisor initialization.

11.1.2.4.7 Partition Start

Partition **start** moves a created partition from the stopped state into the running state, which is the normal execution state of a partition.

This occurs in one of two ways:

- Auto-started. Partition start can occur automatically at hypervisor initialization for partitions defined in the hypervisor configuration tree. When the hypervisor starts the partition it automatically loads all OS images and supplementary images specified in the partition definition in the hypervisor configuration tree (see the guest-image, linux-rootfs, and load-image-table property in the partition definition in the hypervisor configuration tree. If the guest-image property is defined the partition is auto-started unless the no-auto-start property is used.
- Manager partition. A manager partition using the FH_PARTITION_START hcall. It is assumed that prior to starting the partition that all OS and supplementary images are loaded by the partition manager unless the load parameter is specified.

The hypervisor shell can also be used to start partitions, which may be useful during development and debugging.

11.1.2.4.8 Partition Restart

A partition in the running state can be restarted. A restart operation involves stopping the partition, reloading OS images, and starting (partition moves into the running state).

A restart action can come from several sources:

- A partition can ask for an explicit restart of itself via the FH_PARTITION_RESTART hcall
- Watchdog expiration can be configured to cause a partition restart
- A manager partition can restart a partition via the FH_PARTITION_RESTART hcall
- A debug stub may restart a partition.

This occurs in one of two ways:

- For auto-started partitions (see Partition Start on page 2155) the hypervisor performs the full restart sequence—stopping the partition, reloading all OS images and supplementary images, and starting the partition. In this case the partition never moves into the stopped state, but transitions directly from the stopping state to the starting state.
- For partitions that do not have a guest-image property and are managed, a restart causes the partition to be moved into the stopped state and an interrupt is sent to the manager indicating that a restart is requested.

Note: The no-auto-start property affects the initial boot only and not partition restarts.

The hypervisor shell can also be used to restart partitions, which may be useful during development and debugging.

11.1.2.4.9 Partition Stop

Partition stop moves a created partition from the running state into the stopped state.

This occurs in one of two ways:

- A partition may stop itself by using the FH_PARTITION_STOP hcall.
- A manager partition can stop a managed partition using the FH_PARTITION_STOP hcall. Note: this does not cause a clean shutdown of the target partition, but immediately halts the partition.
- A watchdog expiration with a policy configured to stop the partition

A stopped partition will have all interrupts source disabled at the interrupt controller and DMA will be disabled for all devices at the IOMMU unless the no-dma-disable or defer-dma-disable properties are set (see Partition Node Properties on page 2215 and Device Configuration Nodes on page 2222 for additional details).

The hypervisor shell can also be used to stop partitions, which may be useful during development and debugging.

11.1.2.4.10 Partition state at initialization

This section summarizes the initial state of a partition.

11.1.2.4.10.1 Image loading

For auto-started partitions, the hypervisor provides flexible mechanisms for loading images from a memory-mapped location (e.g. flash or a dedicated region of RAM) into a partition prior to partition start.

The guest-image property on a partition node specifies the source and destination addresses of the image that contains the entry point to which control is transferred to when the partition is started. See Partition Node Properties on page 2215, for details on supported image formats and how the entry point is calculated.

The linux-rootfs property on a partition node and can be used to specify the location of a Linux root filesystem to be loaded. The hypervisor will set the appropriate Linux boot arguments on the /chosen bootargs property. See Partition Node Properties on page 2215, for additional details and supported image formats.

A load-image-table property is also supported that allows additional supplementary images to be loaded. See Partition Node Properties on page 2215, for additional details and supported image formats.

11.1.2.4.10.2 Initial Mapped Areas

The ePAPR defines a concept of an *initial mapped area* or IMA. A partition's IMA is the region of memory that contains the entry point for the guest. Both boot CPUs and secondary CPUs begin client program execution in an IMA. The terms Boot IMA and Secondary IMA (SIMA) are used to distinguish the IMAs for boot CPUs and secondary CPUs where necessary.

11.1.2.4.10.2.1 Boot IMA

By default the hypervisor will configure a boot IMA at guest physical address 0x0. As per the ePAPR, the MMU will be configured with the effective address of the boot IMA to be 0x0. This configuration requires that at least one guest physical memory area (GPMA) be defined at guest physical address 0x0 (see Guest Physical Memory Areas on page 2221).

The DTB window (see *dtb-window* property, in Partition Node Properties on page 2215) must be located in the boot IMA.

The size of the boot IMA is passed to the client in r7 as per the ePAPR.

The hypervisor provides additional flexible mechanisms for defining boot IMAs that are not ePAPR complaint. The *init-map-paddr, init-map-size*, and *init-map-vaddr* properties allow customization of the guest physical address, effective address, and size of the boot IMA.

11.1.2.4.10.2.2 Secondary IMA

The guest physical address of secondary IMAs is determined by the entry_address field specified in the spin table used to release secondary vcpus. The effective address will be 0x0. The secondary IMA will always be 4KiB in size.

11.1.2.4.10.3 Initial State of Virtual CPUs

In a partition with multiple vcpus, vcpu #0 is the boot CPU and all other vcpus in the partition are considered secondary. See the ePAPR for additional details on boot and secondary CPUs.

11.1.2.4.10.3.1 CPU Initial State

Within a partition, the initial state of the boot and secondary vcpus complies with the entry conditions specified in the ePAPR. The vcpu state is summarized below:

Table 401. vcpu Initial State

Register	Value
MSR	PR=0, supervisor state EE=0, interrupts disabled ME=0, machine check interrupt disabled CE=0, critical interrupts disabled GS=1, guest state
R3	For boot CPUs will be effective address of the guest device tree image. For secondary CPUs will be value of R3 field in the ePAPR spin table.
R4	0
R5	0
R6	For boot CPUs: ePAPR magic value—0x45504150 For secondary CPUs: zero.
R7	Size of the boot or secondary IMA (see section Initial Mapped Areas on page 2156)
R8	0
R9	0
R10—R31	undefined
F0—F31	undefined (floating point registers)
TCR	WRC=0, no watchdog timer reset will occur WIE=0, watchdog timer interrupt disabled FIE=0, fixed interval timer disabled DIE=0, decrementer interrupt disabled
PIR	Contains the partition-relative CPU index for the current CPU.
other registers	Undefined
TLB	Has a mapping for the IMA as per the ePAPR. The initial mapping will be in TLB1. See Initial Mapped Areas on page 2156.

11.1.2.4.10.3.2 Secondary CPUs

At partition start, all secondary vcpus are in a quiescent state and spinning in the spin-table as per the ePAPR. They can be released from the spin by writing the location specified in the cpu-release-addr property as defined in the ePAPR.

The vcpu does not support a writeable PIR register and thus the pir field in the spin table is not supported.

11.1.2.4.10.4 State of Hypervisor Provided Services

All virtual interrupt sources have interrupts masked in the VMPIC.

11.1.2.4.10.5 State of Platform Hardware

The initial state of I/O devices is undefined, except that all hardware interrupt sources have interrupts masked in the VMPIC.

All DMA-capable devices have DMA disabled unless the no-dma-disable property is specified on a partition node or device node in the hypervisor configuration tree (see Partition Node Properties on page 2215 and Device Configuration Nodes on page 2222 for additional details).

11.1.2.4.11 Privileged Partitions

A partition can be designated as 'privileged' which grants it the ability to issue a platform reset hypercall. See Partition Node Properties on page 2215, for information on defining privileged partitions. See FH_SYSTEM_RESET on page 2186 for information on the FH_SYSTEM_RESET hcall.

11.1.2.5 ePAPR virtualization

11.1.2.5.1 ePAPR Virtualization Overview

The power.org Embedded Power Architecture Platform Requirements (ePAPR) defines virtualization standards including:

- Hypercall ABI
- Hypercall services
- Virtualization extensions to device tree, so that guest operating systems are aware of the resources and services offered by a hypervisor

11.1.2.5.2 Hypercall Application Binary Interface (ABI)

An explicit invocation of the hypervisor by a guest is called a hypercall or hcall. This is performed by executing an instruction that causes an exception, in much the same way as a Unix system call is performed. One argument is a token that designates the actual function to perform. The remaining arguments and their interpretation are specific to the function of the hcall.

The hcall function depends on a modified system call (LEVEL=1) instruction which traps directly to hypervisor mode in the processor.

Summary

- The hypercall number shall be contained in r11.
- Input parameters shall be contained in r3 through r10, inclusive.
- Hypercalls shall return a success code and place this value in r3.
- Further output parameters shall be contained in r4 through r11, inclusive.
- If more data must be transferred in either direction in a single hypercall, that data must be placed into memory, and that must be specified by the hypercall API (the ABI does not define this behavior).

Table 402. Register Volatility

Register	Description
r0	Volatile
r1–r2	Nonvolatile

Table continues on the next page...

Table 402. Register Volatility (continued)

Register	Description
r3	Volatile parameter and return value for status
r4–r10	Volatile input and output value
r11	Volatile hypercall Token and output value
r12	Volatile
r13–r31	Nonvolatile
LR	Nonvolatile
CTR	Volatile
XER	Volatile
CR2–CR4	Nonvolatile
Remaining CR fields	Volatile
Other Registers	Nonvolatile

Contents of registers that are considered "nonvolatile" shall be preserved across hypercalls.

11.1.2.5.3 ePAPR Hypercall Token Definition

The ePAPR hcall ABI specifies that an hcall token identifying the hcall be placed in r11. An hcall token is 32-bits and is defined as follows:

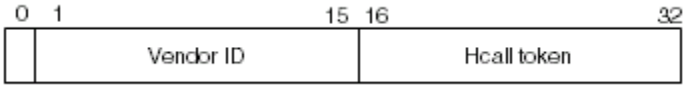


Figure 349. hcall token

Bit 0 (msb) is reserved and must be zero.

The vendor ID encoding is a 15-bit value defined as specified below:

Table 403. Vendor ID encoding

Vendor ID	Vendor Name
0	reserved for private, local use
1	ePAPR (hcall is defined by the ePAPR)
2	NXP Semiconductor

Table continues on the next page...

Table 403. Vendor ID encoding (continued)

Vendor ID	Vendor Name
3	International Business Machines
4	Green Hills Software
5	Enea
6	Wind River Systems
7	Applied Micro Circuits
42	KVM (Kernel Virtual Machine)

The ePAPR hcall tokens values are encoded as specified below.

Table 404. ePAPR Hypercall Token Definition

Hypercall Token Symbolic Name	Value
EV_BYTE_CHANNEL_SEND	1
EV_BYTE_CHANNEL_RECEIVE	2
EV_BYTE_CHANNEL_POLL	3
EV_INT_SET_CONFIG	4
EV_INT_GET_CONFIG	5
EV_INT_SET_MASK	6
EV_INT_GET_MASK	7
EV_INT_EOI	10
EV_DOORBELL_SEND	14

11.1.2.5.4 Hypercall Return Codes

Hypercalls return a status value in r3 encoded as follows:

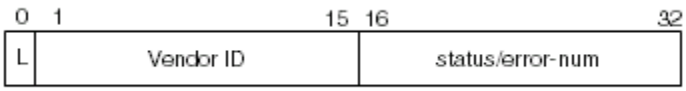


Figure 350. Hypercalls return

Vendor specific errors are encoded by setting the vendor ID (see ePAPR Hypercall Token Definition on page 2159).

Local/private errors that are not vendor specific can be encoded by setting the L bit.

The following tables defines the return codes that may be returned from hypercalls. A return code of zero indicates that the hypercall succeeded.

Table 405. ePAPR hcall return codes

Symbolic Name	Value	Meaning
	0	Success-- the operation completed successfully
EV_EPERM	1	Operation not permitted
EV_ENOENT	2	Entry Not Found
EV_EIO	3	I/O error occurred
EV_EAGAIN	4	The operation had insufficient resources to complete and should be retried
EV_ENOMEM	5	There was insufficient memory to complete the operation
EV_EFAULT	6	Bad guest address
EV_ENODEV	7	No such device
EV_EINVAL	8	An argument supplied to the hcall was out of range or invalid
EV_INTERNAL	9	An internal error occurred
EV_CONFIG	10	A configuration error was detected
EV_INVALID_STATE	11	The object is in an invalid state
EV_UNIMPLEMENTED	12	Unimplemented hypercall
EV_BUFFER_OVERFLOW	13	Caller-supplied buffer too small

11.1.2.5.5 ePAPR Hypervisor Node

Guest operating system can determine the virtualization resources available to them by looking at the **/hypervisor** node that will be present in the device tree passed to the guest operating system.

The name of the hypervisor node shall be "hypervisor" and it must be located at the root of the guest device tree.

The hypervisor node support the following properties:

Table 406. ePAPR hypervisor node

Property Name	Usage	Value Type	Definition
compatible	R	<string>	Must contain "epapr,hypervisor-<version#>" Where version# indicates the version of the ePAPR virtualization extensions that the hypervisor is compatible with.

Table continues on the next page...

Table 406. ePAPR hypervisor node (continued)

Property Name	Usage	Value Type	Definition
hcall-instructions	R	<prop-encoded-array>	Consists of up to 4 cells that specify a sequence of Power ISA instructions used in making a hypercall. Each cell contains a Power ISA opcode. Example: hcall-instructions = <0x44000022>; // sc level=1
guest-id	R	<u32>	A hypervisor provided guest identification number that is guaranteed to be unique across all partitions.
guest-name	R	<string>	A human readable string that describes the guest
has-idle	SD	<none>	If present, the hypervisor supports the EV_IDLE hcall
has-msgsnd-hcall	SD	<none>	If present, the hypervisor supports the EV_MSGSND hcall
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.5.6 ePAPR virtual interrupt controller services

In a virtualized implementation of the Power Architecture, interrupt controller services may be provided by a virtual interrupt controller accessed via a hypercall interface. The ePAPR "virtual interrupt controller" provides interrupt controller services for external interrupts.

External interrupts received by a partition can come from two sources:

- Hardware interrupts - hardware interrupts come from external interrupt lines or on-chip I/O devices
- Virtual interrupts - virtual interrupts are generated by the hypervisor as part of some hypervisor service or hypervisor-created virtual device.

Both types of interrupts are processed using the same programming model and same set of hcalls.

Each interrupt source in a partition has a partition-wide unique interrupt source number. Interrupt number 0 is reserved, and indicates a spurious interrupt (see the EV_INT_IACK hcall).

Table 407. Interrupt controller hcalls

Hypercall	Description
EV_INT_SET_CONFIG	Configures an interrupt
EV_INT_GET_CONFIG	Returns the configuration of an interrupt
EV_INT_SET_MASK	Sets the mask for an interrupt
EV_INT_GET_MASK	Returns the mask for an interrupt
EV_INT_EOI	Signals the end of processing for an interrupt

11.1.2.5.6.1 Virtual Interrupt Controller Device Tree Representation

11.1.2.5.6.1.1 Interrupt Controller Node

The ePAPR virtual interrupt controller is represented as a node in guest device trees with the properties as described in the table below:

Table 408. ePAPR virtual interrupt controller

Property Name	Usage	Value Type	Definition
compatible	R	<string>	Must contain "epapr,hv-pic".
hv-handle	O	<u32>	Specifies a handle to the interrupt controller. This handle may be used in hcalls that require an interrupt controller handle.
#interrupt-cells	R	<u32>	Must be 2
#address-cells	R	<u32>	Must be 0
interrupt-controller	R	<none>	Property must be present to indicate that this node is an interrupt controller
priority-count	R	<u32>	Defines the number of priorities supported by the virtual interrupt controller. The lowest priority (least-favored) is 0.
has-external-proxy	SD	<none>	If present, indicates that the hardware platform and virtual interrupt controller support the external proxy feature of the Power architecture.
no-priority	SD	<none>	If present indicates that the virtual interrupt controller does not implement the interrupt priority mechanism.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.5.6.1.2 Interrupt Specifiers

The *interrupts* property of interrupt children of the "epapr,hv-pic" node consists of two cells encoded as follows:

<interrupt-src-number flags>

Where:

- *interrupt-src-number* specifies the interrupt source number that may be used in the hcalls to configure and manage the interrupt source
- *flags* describes the level-sense encoding of the interrupt, encoded as:

Table 409. epapr,hv-pic interrupt specifier flags encoding

Bits	Description
31	Polarity 0 Polarity is active-low or negative edge triggered. 1 Polarity is active-high or positive edge-triggered.

Table continues on the next page...

Table 409. epapr,hv-pic interrupt specifier flags encoding (continued)

Bits	Description
30	Sense 0 The interrupt is edge sensitive 1 The interrupt is level sensitive

11.1.2.5.6.2 ePAPR interrupt controller hypercalls

11.1.2.5.6.2.1 EV_INT_SET_CONFIG

Hypercall: EV_INT_SET_CONFIG

Description: Configures the priority, destination CPU, and level/sense for an interrupt source

Arguments:

Table 410. EV_INT_SET_CONFIG Arguments

r11	hcall-token	EV_INT_SET_CONFIG
r3	interrupt	Interrupt source number (from the interrupt specifier)
r4	config	Specifies the configuration of the interrupt source. The config value is encoded as described in Interrupt Specifiers on page 2163.
r5	priority	Priority. Specifies the interrupt priority. The allowable range of priorities (lowest to highest) is described by the "priority-count" property on the "epapr,hv-pic" node. A priority value of 0 inhibits signaling of this interrupt.
r6	destination	Destination CPU. A number that identifies which CPU/thread receives the interrupt. This value matches the device tree "reg" property corresponding to the CPU/thread.

Return values:

Table 411. EV_INT_GET_CONFIG return values

r3	Status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : a parameter was out of range or invalid

11.1.2.5.6.2.2 EV_INT_GET_CONFIG

Hypercall: EV_INT_GET_CONFIG

Description: Returns the configuration of the specified interrupt

Arguments:

Table 412. EV_INT_GET_CONFIG Arguments

r11	hcall-token	EV_INT_GET_CONFIG

Table continues on the next page...

Table 412. EV_INT_GET_CONFIG Arguments (continued)

r3	Interrupt	Interrupt source number (from the interrupt specifier)

Return values:

Table 413. EV_INT_GET_CONFIG return values

r3	Status	Status of the hypercall 0: the operation completed successfully EV_EINVAL : the interrupt number was invalid
r4	config	Interrupt configuration. See Interrupt Specifiers on page 2163 for a description of the bit encoding of this value.
r5	priority	Priority. Specifies the interrupt priority. The allowable range of priorities (lowest to highest) is described by the "priority-count" property on the "epapr,hv-pic" node. A priority value of 0 inhibits signaling of this interrupt.
r6	destination	Destination CPU. A number that identifies which CPU/thread receives the interrupt. This value matches the device tree "reg" property corresponding to the CPU/thread.

11.1.2.5.6.2.3 EV_INT_SET_MASK

Hypercall: EV_INT_SET_MASK

Description: Sets the mask for the specified interrupt source

Arguments:

Table 414. EV_INT_SET_MASK Arguments

r11	hcall-token	EV_INT_SET_MASK
r3	interrupt	Interrupt source number (from the interrupt specifier)
r4	mask	Specifies whether the interrupt source is masked Zero: This interrupt source is enabled nonzero: This interrupt source is disabled

Return values:

Table 415. EV_INT_SET_MASK return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : the interrupt number was invalid

11.1.2.5.6.2.4 EV_INT_GET_MASK

Hypercall: EV_INT_GET_MASK

Description: Returns the mask for the specified interrupt source

Arguments:

Table 416. EV_INT_GET_MASK Arguments

r11	hcall-token	EV_INT_GET_MASK
r3	interrupt	Interrupt source number (from the interrupt specifier)

Return values:

Table 417. EV_INT_GET_MASK return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : the interrupt number was invalid
r4	mask	Specifies whether the interrupt source is masked Zero: This interrupt source is enabled nonzero: This interrupt source is disabled

11.1.2.5.6.2.5 EV_INT_EOI

Hypercall: EV_INT_EOI

Description: Signals the end of processing for the specified interrupt, which must be the highest priority interrupt currently in service.

An "in service" interrupt is one that has been acknowledged—either explicitly via EV_INT_IACK or by hardware that supports Power Architecture Category: External Proxy.

Arguments:

Table 418. EV_INT_EOI Arguments

r11	hcall-token	FH_VMPIC_ EOI
r3	interrupt	Interrupt source number (from the interrupt specifier)

Return values:

Table 419. EV_INT_EOI return values

r3	status	Status of the hypercall 0 the operation completed successfully EV_INTERNAL-an error occurred

NOTE

It is the responsibility of guest software to ensure that the interrupt source specified in this hcall is the highest priority interrupt in service.

11.1.2.5.7 Byte-channel Services

11.1.2.5.7.1 Byte-channel Services Overview

A byte-channel is a hypercall-based, interrupt-driven character-based I/O channel (similar to a UART). Hypercalls are available to send, receive, and poll a channel.

Table 420. hcalls for Byte-Channel Services

Hypercall	Description
EV_BYTE_CHANNEL_SEND	Sends data to a byte-channel.
EV_BYTE_CHANNEL_RECEIVE	Receives bytes of data from a byte-channel.
EV_BYTE_CHANNEL_POLL	Returns the status of the byte-channel send and receive buffers

11.1.2.5.7.2 Byte-channel Services Guest Device Tree Representation

Byte-channels are specified in the guest device tree as follows:

Table 421. Guest Device Tree

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "epapr,hv-byte-channel"
hv-handle	R	<u32>	Specifies the handle to the byte-channel. This value must be used by guest software for all byte-channel related hypercalls.
interrupts	R	<prop-encoded-array	Must consist of one or two interrupt specifiers encoded as follows: <rx-interrupt-specifier [tx-interrupt-specifier]> If only one interrupt specifier is present, then the implementation does not support TX interrupts. The virtual device shall generate a receive interrupt (RX) after an increase in the number of bytes available in the byte-channel's receive buffer. If implemented, the virtual device shall generate a transmit interrupt (TX) after an increase in the space available in the byte-channel's transmit buffer. Both TX and RX interrupts shall be edge-triggered.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.5.7.3 Byte-channel Hypercalls

The byte-channel hypercalls are nonblocking and synchronous. All hypercalls perform the requested task and return. There are no asynchronous side effects. If the request could not be completed (e.g. no space, no data) the return code of the hypercall indicates the reason the call did not complete.

11.1.2.5.7.3.1 EV_BYTE_CHANNEL_SEND

Hypercall: EV_BYTE_CHANNEL_SEND

Description: Description: Sends data to a byte-channel. The maximum number of bytes that can be sent is 16.

Arguments:

Table 422. EV_BYTE_CHANNEL_SEND Arguments

r11	hcall-token	EV_BYTE_CHANNEL_SEND
r3	handle	Byte-channel handle
r4	count	has count of bytes available in r5, r6, r7, and r8
r5,r6,r7,r8	byte string	The byte string starts in register r5 and proceeds toward the low order byte in register r8 for the number of characters specified in r4. The contents of all other byte locations of registers r5-r8 are undefined. Each register holds at most 4 bytes, which means on 64-bit CPUs only the low order 4 bytes of the registers are defined.

Return values:

Table 423. EV_BYTE_CHANNEL_SEND return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL: Invalid parameter EV_EAGAIN: The byte-channel buffer did not have sufficient space and no characters were sent. The operation should be retried at a later time.
r4	count	Count of characters sent.

11.1.2.5.7.3.2 EV_BYTE_CHANNEL_RECEIVE

Hypercall: EV_BYTE_CHANNEL_RECEIVE

Description: Receives bytes of data from a byte-channel. The maximum number of bytes received is 16.

Arguments:

Table 424. EV_BYTE_CHANNEL_RECEIVE Arguments

r11	hcall-token	EV_BYTE_CHANNEL_RECEIVE
r3	handle	Byte-channel handle
r4	max receive byte count	Specifies the maximum number of characters to receive. This value can be no larger than 16.

Return values:

Table 425. EV_BYTE_CHANNEL_RECEIVE return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : a parameter was invalid
r4	count	Count of characters available in r5, r6, r7 and r8 A count of zero indicates no characters are available.
r5,r6,r7,r8	character string	The byte string starts in register r5 and proceeds toward the low order byte in register r8 for the number of characters specified in r4. The contents of all other byte locations of registers r5-r8 are undefined. Each register holds at most 4 bytes, which means on 64-bit CPUs only the low order 4 bytes of the registers are defined.

11.1.2.5.7.3.3 EV_BYTE_CHANNEL_POLL

Hypercall: EV_BYTE_CHANNEL_POLL

Description: returns the status of the byte-channel send and receive buffers

Arguments:

Table 426. EV_BYTE_CHANNEL_POLL Arguments

r11	hcall-token	EV_BYTE_CHANNEL_POLL
r3	handle	Byte-channel handle

Return values:

Table 427. EV_BYTE_CHANNEL_POLL return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : handle was invalid
r4	rx count	Count of bytes available in the byte-channel's receive buffer.
r5	tx count	Count of space available in the byte-channel's transmit buffer.

11.1.2.5.8 Inter-partition doorbells

11.1.2.5.8.1 Inter-partition Doorbells Overview

A doorbell is an inter-partition signaling mechanism. A doorbell allows one partition to cause an interrupt in one (or possibly more) target partitions. The doorbell results in an external interrupt in the destination partition (i.e. the interrupt is gated off of MSR[EE] and the exception handler specified in IVOR4 execute).

Note, doorbell interrupts may be coalesced—if multiple senders issue a doorbell to the same receive endpoint, the receiver may see only one interrupt. If a sender issues multiple doorbells to a receiver that has interrupts disabled the receiver may only see one interrupt.

A partition is aware of the doorbells available to it through its guest device tree which contains a node for each doorbell endpoint.

11.1.2.5.8.2 Doorbell Send Endpoints

Doorbell send endpoints are represented in a guest device tree with the following properties:

Table 428. Doorbell Send Endpoints

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "epapr,hv-doorbell-send-handle"
hv-handle	R	<u32>	Specifies the handle to the doorbell. This value must be used by guest software for doorbell related hypercalls.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.5.8.3 Doorbell Receive Endpoints

Doorbell receive endpoints are represented in a guest device tree with the following properties:

Table 429. Doorbell Receive Endpoints

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "epapr,hv-doorbell-receive-handle"
interrupts	R	<prop-encoded-array	Must consist of one interrupt specifier.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.5.8.4 Doorbell Hypercall

Hypercall: EV_DOORBELL_SEND

Description: Sends a doorbell signal to a partition. This causes an external interrupt in the destination partition.

Arguments:

Table 430. EV_DOORBELL_SEND Arguments

r11	hcall-token	EV_DOORBELL_SEND
r3	handle	Specifies the doorbell handle of the partition to signal. This handle comes from the device tree.

Return values:

Table 431. EV_DOORBELL_SEND return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : handle was invalid EV_CONFIG: there was a configuration error with the doorbell

11.1.2.6 NXP hypervisor services

In additional to the ePAPR hypercall services, the hypervisor provides additional services described in this chapter.

11.1.2.6.1 NXP Hypervisor Services Overview

All NXP hypercalls follow the ePAPR hypercall ABI (see Hypercall Application Binary Interface (ABI) on page 2158).

11.1.2.6.2 NXP Hypercall Token Definition

The NXP hcall tokens values are encoded.

Table 432. NXP Hypercall Token Definition

Hypercall Token Symbolic Name	Value
FH_ERR_GET_INFO	1
FH_PARTITION_GET_DTPROP	2
FH_PARTITION_SET_DTPROP	3
FH_PARTITION_RESTART	4
FH_PARTITION_GET_STATUS	5
FH_PARTITION_START	6
FH_PARTITION_STOP	7
FH_PARTITION_MEMCPY	8
FH_DMA_ENABLE	9
FH_DMA_DISABLE	10
FH_SEND_NMI	11
FH_VMPIC_GET_MSIR	12
FH_SYSTEM_RESET	13
FH_GET_CORE_STATE	14
FH_ENTER_NAP	15
FH_EXIT_NAP	16
FH_CLAIM_DEVICE	17

11.1.2.6.3 Interrupt Controller Services

The ePAPR interrupt controller services (see ePAPR Hypervisor Node on page 2161) are used for managing interrupt sources. This section describes additional services and extensions to the ePAPR PIC services.

11.1.2.6.3.1 Interrupt Processing

Interrupts should be initialized and processed by guest software as follows:

- Guest software configures the vector number, priority, polarity, sense, and destination CPU of an interrupt source with the EV_INT_SET_CONFIG hypercall
- Interrupt sources can be masked and unmasked at the virtual interrupt controller with the EV_INT_SET_MASK hypercall
- To process a pending external interrupt:
  - The guest's interrupt service routine reads the processor core's EPR (external proxy register) to get the interrupt number of the pending interrupt. The EPR must be read before enabling interrupts (MSR[EE]).
  - When interrupt processing is complete the guest issues a EV_INT_EOI- this signals the end of processing for the highest-priority interrupt currently in service.

NOTE

Hardware interrupt sources cannot be configured while an interrupt is active.

11.1.2.6.3.2 Message Signaled Interrupt Processing

When a PCI Express message signaled interrupt occurs, an additional step is necessary in addition to those described in Interrupt Processing on page 2172 in processing the interrupt. Each MSI interrupt may be caused by one of 32 sources and this is indicated in the corresponding MPIC MSIR. This register can be read using the FH_VMPIC_GET_MSIR hcall.

11.1.2.6.3.3 Interrupt Priority

NOTE

The hypervisor does not support an ePAPR-compliant priority mechanism. The interrupt controller node specifies *no-priority* and does not specify a priority count.

All virtual interrupts (e.g. byte-channels, inter-partition doorbells, partition management interrupts) have the same priority and are sent to partitions in the order that they are received.

Hardware interrupt sources can be individually configured with priorities from 0 (least favored) to 15 (most favored) with the EV_INT_SET_CONFIG hypercall.

Given several concurrent incoming interrupts, the highest priority interrupt is asserted if the following apply:

The interrupt is not masked.

The priority of the interrupt is higher than any other interrupts in service.

11.1.2.6.3.4 Interrupt Controller Node in the Guest Device Tree

The table below defines additions to the ePAPR interrupt controller node representation (see Byte-channel Services Overview on page 2167).

VMPIC is represented in guest device trees with a device node name "vmpic" and properties as described below.

Table 433. VMPIC Representation

Property Name	Usage	Value Type	Definition
compatible	SD	<string>	If direct EOI is enabled for this partition (see section FH_VMPIC_GET_MSIR on page 2173) the compatible will include the value "fsl,hv-mpic-per-cpu".

Table continues on the next page...

Table 433. VMPIC Representation (continued)

Property Name	Usage	Value Type	Definition
reg	O	<prop-encoded-array>	A standard property. This property will only be present if direct EOI is enabled for this partition (see section FH_VMPIC_GET_MSIR on page 2173). If present contains the guest physical address and size of the per-CPU MPIC registers.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.6.3.5 ePAPR Interrupt Specifiers

Interrupt specifiers follow the ePAPR with an additional bit in the flags field specifying whether direct EOI may be used.

Table 434. Interrupt specifier flags encoding

Bits	Description
31	Polarity 0 Polarity is active-low or negative edge triggered. 1 Polarity is active-high or positive edge-triggered.
30	Sense 0 The interrupt is edge sensitive 1 The interrupt is level sensitive
29	MPIC direct 0 Direct EOI may not be used in the processing of this interrupt. 1 Direct EOI may be used in the processing of this interrupt

11.1.2.6.3.6 MSI Representation in the Guest Device Tree

An MSI node is identical to that of the hardware device tree except for the following:

- The *compatible* property string is "fsl,hv-mpic-msi"

The *reg property* is not present. This is because the MSI block is not accessible directly via loads and stores.

11.1.2.6.3.7 FH_VMPIC_GET_MSIR

Hypercall: FH_VMPIC_ GET_MSIR

Description: Description: Gets the MPIC MSIRx (Shared Message Signaled Interrupt Register) associated with the specified interrupt.

Arguments:

Table 435. FH_VMPIC_ GET_MSIR Arguments

r11	hcall-token	FH_VMPIC_GET_MSIR

Table continues on the next page...

Table 435. FH_VMPIC_ GET_MSIR Arguments (continued)

r3	interrupt	Interrupt number (from guest device tree) of the interrupt source for which to get the MSI interrupt status.

Return values:

Table 436. FH_VMPIC_ GET_MSIR return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : the interrupt handle was not for an MSI interrupt
r4	msir value	Contains the MSIRxx value for the specified interrupt

C API:

Table 437. FH_VMPIC_ GET_MSIR C API

unsigned int fh_vmpic_get_msir (*unsigned int* interrupt, unsigned int *msir_val)

11.1.2.6.3.8 Non Maskable Interrupts

An hcall mechanism is available that enables one CPU to send a non-maskable interrupt to other CPUs within the same partition. See the FH_SEND_NMI hcall (FH_SEND_NMI on page 2185).

The interrupt cannot be masked via the MSR and appears on the target CPUs as a machine check with MCSR[NMI] set.

11.1.2.6.3.9 Direct EOI

The hypervisor provides a mechanism by which in some circumstances an end-of-interrupt can be performed with no hypercall. When the *mpic-direct-eoi* property is specified in on the node defining a partition, the guest OS will be granted direct access to the per-CPU registers in the MPIC registers (see the *P4080 QorIQ Integrated Host Processor Reference Manual*), which enables the guest to perform direct EOIs.

Enabling direct EOI for a partition has some important considerations and restrictions.

Restrictions

- Direct EOI is only supported for hardware interrupts sources directly managed by the hardware MPIC in the SOC
- EOI for all other interrupt source, such as virtual interrupts (e.g. doorbells, byte-channels), must be done with the FH_VMPIC_EOI hcall. MPIC managed interrupts can be distinguished in the interrupt specifier for the interrupt (see Interrupt Controller Node in the Guest Device Tree on page 2172).

Considerations

- The partition and guest OS to which direct EOI is granted **must be trusted and well behaved**. The MPIC registers which are being granted contain PIC registers beside the EOI register which should not be accessed by the guest.
- Direct EOI will not be compatible with any future implementations of the hypervisor that support multiple operating systems per physical CPU.

To enable direct EOI, specify the mpic-direct-eoi property in the hypervisor configuration tree on the partition node specifying the partition (see Partition Node Properties on page 2215).

With direct EOI enabled, the vmpic node in the guest device tree will have a mpic-reg property and a compatible string that includes "fsl,hv-mpic-per-cpu" (see Interrupt Controller Node in the Guest Device Tree on page 2172).

11.1.2.6.4 Inter-partition doorbell services

In addition to the standard ePAPR inter-partition doorbells, the NXP hypervisor supports several fast doorbells with the same interface.

11.1.2.6.4.1 Fast Doorbells

The hypervisor supports up to four 'fast' doorbells. These are doorbells that use a different hardware mechanism which provides lower latency compared to regular doorbells. In addition, the direct EOI mechanism (see Direct EOI on page 2174) is supported for fast doorbells.

See Global Doorbell Definition on page 2213, for information on defining fast doorbells.

The send and receive endpoints are identical to regular doorbell.

NOTE

Due the underlying hardware mechanism used to implement fast doorbells, the ability to mask/unmask fast doorbells interrupts with the interrupt controllers services is limited. Masking the receive interrupt for any receive endpoint will mask all receive endpoints for that doorbell. It is recommended that doorbells not be masked at the interrupt controller.

11.1.2.6.5 Partition management

11.1.2.6.5.1 Partition Management Overview

The hypervisor architecture supports the concept of a manager partition. A manager partition has the capability to start, stop, and restart other partitions using hcalls. For partitions in the stopped state a manager partition can copy data to and from the managed partition, thus enabling the manager to load OS images, etc into the managed partition.

11.1.2.6.5.2 Partition Handles

Partitions are managed via handles. A manager partition will have a partition management node in its device tree for each partition it manages. The reg property in the partition management node defines the handle used in partition management hcalls.

Partitions can perform partition related hcalls on themselves using the special interrupt handle value of -1.

11.1.2.6.5.3 Partition Management Interrupts

A manager partition may get doorbell interrupts related to managed partitions. Three types of interrupts may be generated:

- Notification on managed partition state changes
  - Transition from *starting* to *running*
  - Transition from *stopping* to *stopped*
  - Transition from *pausing* to *paused*
  - Transition from *resuming* to *running*
- Notification on managed partition watchdog expiration
- Notification on managed partition restart request

The partition management node has 3 doorbell node children that define these interrupt sources.

In addition, a special doorbell is defined that enables a manager partition to signal a managed partition to do a clean shutdown. The manager has a send doorbell endpoint and the managed partition has a receive doorbell endpoint.

See Partition Management Guest Device Tree Representation on page 2176for details regarding these send and receive doorbell interrupts for partition management.

11.1.2.6.5.4 Partition Status

A manager partition can read the status of a managed partition using the FH_PARTITION_GET_STATUS hcall.

11.1.2.6.5.5 Partition Management Guest Device Tree Representation

Partitions are managed via handles. Each managed partition has a partition management node under the *handles* node in the manager partition's guest device tree. The node specifies a handle to the partition used by management related hypercalls. In addition, each partition management node contains 4 child nodes that represent 3 receive doorbell endpoints for the partition management interrupts that the manager may receive and 1send doorbell endpoint to request a clean shutdown by the managed partition.

The three receive doorbells are as follows:

- Managed partition state change—signaled when the managed partition moves between the *stopped* and *running* states
- Managed partition watchdog expiration—signaled when the managed partition has a watchdog expiration event
- Managed partition restart request—signaled when the managed partition has made a reboot request to the hypervisor, and the hypervisor does not have a client program image to load

The partition management node for the managed partition will have the properties in the table below:

Table 438. Guest Device Tree Partition Management Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must contain "fsl,hv-partition-handle"
reg	R	<u32>	Specifies the handle to the partition
label	R	<string>	Identical to the label property on the partition node in the hypervisor configuration tree describing the managed partition.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

A managed partition state change doorbell node is a child of the partition management node and has the following properties:

Table 439. Managed Partition State Change Doorbell Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must contain 2 strings—"fsl,hv-state-change-doorbell," "fsl,hv-doorbell-receive-handle"
interrupts	R	<prop-encoded-array>	Consists of one interrupt specifier.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

A managed partition watchdog expiration doorbell node is a child of the partition management node and has the following properties:

Table 440. Managed Partition Watchdog Expiration Doorbell Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must contain 2 strings—"fsl,hv-watchdog-expiration-doorbell," "fsl,hv-doorbell-receive-handle"
interrupts	R	<prop-encoded-array	Consists of one interrupt specifier.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

managed partition restart request doorbell node is a child of the partition management node and has the following properties:

Table 441. Managed Partition Restart Request Doorbell Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must contain 2 strings—"fsl,hv-reset-request-doorbell," "fsl,hv-doorbell-receive-handle"
interrupts	R	<prop-encoded-array	Consists of one interrupt specifier.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

managed partition shutdown request doorbell node is a child of the partition management node and has the following properties:

Table 442. Managed Partition Shutdown Request Doorbell Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must contain 2 strings—"fsl,hv-shutdown-request-doorbell," "fsl,hv-doorbell-send-handle"
reg	R	<u32>	Specifies the handle to the doorbell.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.6.5.6 Partition management hypercalls

The hypervisor API specifies a set of hcalls to be used by a manager partition to manage partitions.

Table 443. Partition management hcalls

Hypercall	Description
FH_PARTITION_START	Boots and starts execution of the specified partition.
FH_PARTITION_STOP	Stops a partition.

Table continues on the next page...

Table 443. Partition management hcalls (continued)

Hypercall	Description
FH_PARTITION_RESTART	Requests that the current partition be reinitialized and restarted.
FH_PARTITION_GET_ STATUS	Gets the status of a partition.
FH_PARTITION_MEMCPY	Copies data to/from another partition's memory.
FH_PARTITION_GET_DTPROP	Gets a guest device tree property from the specified partition
FH_PARTITION_SET_DTPROP	Sets a guest device tree property in the specified partition
FH_PARTITION_STOP_DMA	Disables DMA for devices in the specified partition

11.1.2.6.5.6.1 FH_PARTITION_START

Hypercall: FH_PARTITION_START

Description: Boots and starts the execution of the specified partition. Partitions with a status of "running" may not be started.

This hypercall starts the partition's boot CPU core only. Partitions with multiple cores release their secondary cores as they wish using the secondary core spin table as defined by the ePAPR.

Arguments:

Table 444. FH_PARTITION_START Arguments

r11	hcall-token	FH_PARTITION_START
r3	partition- handle	specifies the handle of the partition to start
r4	entry-point	specifies the offset into the partition's initial mapped area (IMA) where the operating system entry point is located
r5	load	If non-zero, specifies that any hypervisor loaded images should be loaded prior to starting the partition.

Return values:

Table 445. FH_PARTITION_START return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : the handle was invalid EV_INVALID_STATE : the targeted partition was not in the "stopped" state

C API:

Table 446. FH_PARTITION_START C API

unsigned int **fh_partition_start** (*int* **partition**, *uint32_t* **entry_point**, *int* **load**)

11.1.2.6.5.6.2 FH_PARTITION_STOP

Hypercall: FH_PARTITION_STOP

Description: Halts the execution of a running partition.

Arguments:

Table 447. FH_PARTITION_STOP Arguments

r11	hcall-token	FH_PARTITION_STOP
r3	partition- handle	Specifies the handle of the partition to stop. A value of -1 indicates the *current* partition-this results in the partition exiting and moving into the stopped state.

Return values:

Table 448. FH_PARTITION_STOP return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : the handle was invalid EV_INVALID_STATE : the targeted partition was not in the "running" state

C API:

Table 449. FH_PARTITION_STOP C API

unsigned int **fh_partition_stop** (*int* **partition**)

11.1.2.6.5.6.3 FH_PARTITION_RESTART

Hypercall: FH_PARTITION_RESTART

Description: Requests that a partition be stopped and restarted.

Arguments:

Table 450. FH_PARTITION_RESTART Arguments

r11	hcall-token	FH_PARTITION_RESTART
r3	partition-handle	Specifies the handle of the partition to restart. A value of -1 indicates the *current* partition.

Return values:

FH_PARTITION_RESTART return values

r3	status	Status of the hypercall 0 : success-the request to restart was successful EV_EINVAL : the handle was invalid EV_INVALID_STATE : the targeted partition was not in the "running" state

C API:

Table 451. FH_PARTITION_RESTART C API

unsigned int **fh_partition_restart** (*int* **partition**)

NOTE

For a restart of the current partition this hcall never returns.

11.1.2.6.5.7 FH_PARTITION_GET_STATUS

Hypercall: FH_PARTITION_GET_STATUS

Description: Gets the status of a partition.

Arguments:

Table 452. FH_PARTITION_GET_STATUS Arguments

r11	hcall-token	FH_PARTITION_GET_STATUS
r3	partition handle	specifies the id of the partition for which the status should be read

Return values:

Table 453. FH_PARTITION_GET_STATUS return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL: the supplied handle was not valid
r4	partition status	Status of the partition 0 : stopped 1 : running 2 : starting 3 : stopping 4 : pausing 5 : paused 6 : resuming

C API:

Table 454. FH_PARTITION_GET_STATUS C API

unsigned int fh_partition_get_status (*int* partition, *unsigned int* * status)

11.1.2.6.5.7.1 FH_PARTITION_MEMCPY

Hypercall: FH_PARTITION_MEMCPY

Description: Copies data between the current ("*local*") partition and another ("*remote*") partition's memory. The physical addresses of the source and destination memory and the amount of data to copy are represented in a scatter gather list. Copying in either direction is supported—local-to-remote, or remote-to-local.

The scatter gather list consists of a variable-length array of struct fh_sg_list values. This structure must be guest physically contiguous. aligned on 32-byte boundary, so that no single structure can span two pages.

```
struct fh_sg_list {
        uint64_t source;      // guest physical address to copy from
        uint64_t destination; // guest physical address to copy to
        uint64_t size;        // number of bytes to copy
        uint64_t reserved;    // reserved, must be zero
}
```

The physical addresses specified must be valid guest physical addresses within the respective partition—the destination physical address must be valid in the destination partition (and may not be valid in the source partition).

Arguments:

Table 455. FH_PARTITION_MEMCPY Arguments

r11	hcall-token	FH_PARTITION_MEMCPY
r3	source-partition-handle	Specifies the handle of the source partition. A value of -1 indicates the *local* partition, otherwise the value is the handle of the *remote* partition.
r4	destination-partition-handle	Specifies the handle of the destination partition. A value of -1 indicates the *local* partition, otherwise the value is the handle of the *remote* partition.
r5	sg-list-high	Specifies the high-order 32-bits of the physical address of a scatter gather list that describes the entire data block being copied.
r6	sg-list-low	Specifies the low-order 32-bits of the physical address of a scatter gather list that describes the entire image being loaded. See definition of *sg-list-high* for a definition of the scatter gather list.
r7	count	A count of the number of items in the scatter gather list

Return values:

Table 456. FH_PARTITION_MEMCPY return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : a parameter was invalid EV_EFAULT - bad guest address

C API:

Table 457. FH_PARTITION_MEMCPY C API

unsigned int **fh_partition_memcpy** (*int* **source**, *int* **target**, *phys_addr_t* **sg_list** *, unsigned int* **count** *)*

11.1.2.6.5.7.2 FH_PARTITION_SET_DTPROP

Hypercall: FH_PARTITION_SET_DTPROP

Description: Sets the value of a property for the specified node in the guest device tree. This call is not permitted on partitions in the "starting" state. The address arguments are guest physical addresses and must point to data that is within a physically contiguous region of memory.

If the specified property does not exist, it will be created. If the property does exist, it will be replaced.

Arguments:

Table 458. FH_PARTITION_SET_DTPROP Arguments

r11	hcall-token	FH_PARTITION_SET_DTPROP
r3	partition-handle	specifies the handle of the partition
r4	dtpath-addr-hi	specifies the high order 32-bits of the guest physical address of the null-terminated string containing the full path to the node to update
r5	dtpath-addr-lo	specifies the low order 32-bits of the guest physical address of the null-terminated string containing the full path to the node to update
r6	propname-addr-hi	specifies the high order 32-bits of the guest physical address of the null-terminated string containing the name of the property to update
r7	propname-addr-lo	specifies the low order 32-bits of the guest physical address of the null-terminated string containing the name of the property to update
r8	propvalue-addr-hi	specifies the high order 32-bits of the guest physical address of the value to set in the property
r9	propvalue-addr-lo	specifies the low order 32-bits of the guest physical address of the value to set in the property
r10	propvalue-length	length (in bytes) of the property value

Return values:

Table 459. FH_PARTITION_SET_DTPROP return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : a parameter was invalid EV_ENOMEM - no memory to expand device tree EV_EFAULT - bad guest address EV_ENOENT - path or property not found EV_INVALID_STATE - the target partition was not in the 'stopped' state

C API:

Table 460. FH_PARTITION_SET_DTPROP C API

```
unsigned int fh_partition_dtprop_set(int handle,
uint64_t dtpath_addr,
uint64_t propname_addr,
uint64_t propvalue_addr,
uint32_t propvalue_len);
```

11.1.2.6.5.7.3 FH_PARTITION_GET_DTPROP

Hypercall: FH_PARTITION_GET_DTPROP

Description: Gets the value of a property for the specified node in the guest device tree. This call is not permitted on partitions in the "starting" state. The address arguments are guest physical addresses and must point to data that is within a physically contiguous region.

Arguments:

Table 461. FH_PARTITION_GET_DTPROP Arguments

r11	hcall-token	FH_PARTITION_GET_DTPROP
r3	partition-handle	specifies the handle of the partition
r4	dtpath-addr-hi	specifies the high order 32-bits of the guest physical address of the null-terminated string containing the full path to the node
r5	dtpath-addr-lo	specifies the low order 32-bits of the guest physical address of the null-terminated string containing the full path to the node
r6	propname-addr-hi	specifies the high order 32-bits of the guest physical address of the null-terminated string containing the name of the property to get
r7	propname-addr-lo	specifies the low order 32-bits of the guest physical address of the null-terminated string containing the name of the property to get

Table continues on the next page...

Table 461. FH_PARTITION_GET_DTPROP Arguments (continued)

r8	propvalue-addr-hi	specifies the high order 32-bits of the guest physical address of the location to store the property value.
r9	propvalue-addr-lo	specifies the low order 32-bits of the guest physical address of the location to store the property value
r10	propvalue-length	maximum length (in bytes) of the property value

Return values:

Table 462. FH_PARTITION_GET_DTPROP return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : a parameter was invalid EV_ENOMEM - no memory to expand device tree EV_EFAULT - bad guest address EV_INVALID_STATE - the target partition was not in the 'stopped' state
r4	propvalue-length	actual length (in bytes) of the property value returned

C API:

Table 463. FH_PARTITION_GET_DTPROP C API

```
unsigned int fh_partition_dtprop_get(int handle,
uint64_t dtpath_addr,
uint64_t propname_addr,
uint64_t propvalue_addr,
uint32_t *propvalue_len);
```

11.1.2.6.5.7.4 FH_PARTITION_STOP_DMA

Hypercall: FH_PARTITION_STOP_DMA

Description: Disables DMA for all devices in the managed partition.

Some partition configurations may specify that events such as a watchdog timeout should stop a partition. Normally when a partition moves into the stopped state DMA is stopped. However, there are use cases where it may be required to have DMA continue while a partition is stopped and subsequently restarted. The *defer-dma-disable* property (see Partition Node Properties on page 2215) specifies this behavior. If the disabling of DMA is deferred, it can explicitly stopped with the FH_PARTITION_STOP_DMA hcall. FH_PARTITION_STOP_DMA gives control of when DMA is stopped to the manager partition.

Deferring the disabling of DMA enables the manager partition to check the state and error status of the target partition prior to shutting off DMA, as the disabling of DMA in the target partition may generate additional errors from I/O devices belonging to the partition that may still be active.

The target partition must be in the *stopped* state.

Arguments:

Table 464. FH_PARTITION_STOP_DMA Arguments

r11	hcall-token	FH_PARTITION_STOP_DMA
r3	partition-handle	specifies the handle of the target partition

Return values:

Table 465. FH_PARTITION_STOP_DMA return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_INVALID_STATE : the target partition was not stopped EV_EINVAL: handle was invalid

C API:

Table 466. FH_PARTITION_STOP_DMA C API

unsigned int **fh_partition_stop_dma** (*unsigned int* **partition-handle**)

11.1.2.6.6 General services

The hypervisor API specifies a set of hcalls to provide general services. The hcalls are summarized below.

Table 467. General services hypercalls

Hypercall	Description
FH_SEND_NMI	Sends a non-maskable interrupt to the selected CPUs (within the current partition)
FH_SYSTEM_RESET	Requests a hardware reset of the system

11.1.2.6.6.1 FH_SEND_NMI

Hypercall: FH_SEND_NMI

Description: Sends a non-maskable interrupt to the selected CPUs within the current partition..

The interrupt cannot be masked via the MSR and appears on the target CPUs as a machine check with MCSR[NMI] set.

Arguments:

Table 468. FH_SEND_NMI Arguments

r11	hcall-token	FH_SEND_NMI
r3	vcpu-mask	mask specifying which virtual CPUs should receive the NMI. The LSB (least significant bit) specifies vcpu 0.

Return values:

Table 469. FH_SEND_NMI return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL : the vcpu mask was not valid

C API:

Table 470. FH_SEND_NMI C API

unsigned int **fh_send_nmi** *(uint32_t* **vcpu_mask** *)*

11.1.2.6.6.2 FH_SYSTEM_RESET

Hypercall: FH_SYSTEM_RESET

Requests a hardware reset of the system. If successful the call does not return. This hcall is only supported in privileged partitions (see Privileged Partitions on page 2158).

Description:

Arguments:

Table 471. FH_SYSTEM_RESET Arguments

r11	hcall-token	FH_SYSTEM_RESET

Return values:

Table 472. FH_SYSTEM_RESET return values

r3	status	Status of the hypercall EV_EPERM—operation not permitted (partition was not privileged). EV_ENODEV—don't know how to reset on this hardware EV_EIO—reset failed

C API:

Table 473. FH_SYSTEM_RESET C API

unsigned int **fh_system_reset** *(void)*

11.1.2.6.7 IOMMU services

The NXP QorIQ platform contains an IOMMU (also known as a Peripheral Access Management Unit or PAMU) that provides a mechanism to control and authorize device-to-memory accesses. A PAMU provides authorization and translation services for I/O traffic entering the Host/Coherency Domain.

The hypervisor manages the PAMUs and provides services to partitions to configure access windows enabling DMA-capable devices to read and write system memory.

In the hypervisor configuration tree DMA *windows* can be defined which specify a set of address windows in terms of guest physical addresses (see DMA windows on page 2211). These *DMA window* objects define the allowable address ranges

that devices can access, and are then referenced by device nodes representing DMA-capable devices (see Device nodes on page 2222). Based on the hypervisor configuration tree, the hypervisor sets up the tables used by the PAMU hardware.

At partition initialization, DMA is enabled by default. Guest software can explicitly disable and enables DMA via the FH_DMA_ENABLE and FH_DMA_DISABLE hcall.

11.1.2.6.7.1 IOMMU Services Guest Device Tree Representation

DMA capable devices have one or more of the properties described below in their guest device trees.

Table 474. Guest Device Tree DMA Properties

Property Name	Usage	Value Type	Definition
fsl,hv-dma-handle	SD	<u32>	Value is an array of liodn handle used in making DMA enable/disable hcalls
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.6.7.2 IOMMU Hypercalls

The hypervisor API specifies two hcalls to enable and disable.

Table 475. IOMMU Hypercalls

Hypercall	Description
FH_DMA_ENABLE	Enables DMA for an I/O device.
FH_DMA_DISABLE	Disables DMA for an I/O device.

11.1.2.6.7.3 IOMMU Hypercall Interface

11.1.2.6.7.3.1 FH_DMA_ENABLE

Hypercall: FH_DMA_ENABLE

Description: Enables DMA for the specified device. Note: if the specified device already had DMA enabled the return value from this hcall will still indicate success.

Arguments:

Table 476. FH_DMA_ENABLE Arguments

r11	hcall-token	FH_DMA_ENABLE
r3	handle	Specifies the handle of the I/O device for which to enable DMA. Guest software determines this value from the "*fsl,hv-dma-handle*" property in the device node in the guest device tree.

Return values:

Table 477. FH_DMA_ENABLE return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_INTERNAL : an error occurred EV_EINVAL: the supplied LIODN was not valid

C API:

Table 478. FH_DMA_ENABLE C API

unsigned int **fh_dma_enable** (*unsigned int* **handle**)

11.1.2.6.7.3.2 FH_DMA_DISABLE

Hypercall: FH_DMA_DISABLE

Description: Disables DMA for the specified device. Note: if the specified device already had DMA enabled the return value from this hcall will still indicate success.

Arguments:

Table 479. FH_DMA_DISABLE Arguments

r11	hcall-token	FH_DMA_DISABLE
r3	handle	specifies the handle of the I/O device for which to disable DMA. Guest software determines this value from the "*fsl,hv-dma-handle*" property in the device node in the guest device tree.

Return values:

Table 480. FH_DMA_DISABLE return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_NTERNAL : an error occurred EV_EINVAL: the supplied LIODN was not valid

C API:

Table 481. FH_DMA_DISABLE C API

unsigned int **fh_dma_disable** (*unsigned int* **liodn**)

11.1.2.6.8 Error Management

11.1.2.6.8.1 Error Management Overview

The hypervisor provides an error management architecture to report and log hardware errors.

Several types of hardware errors may occur in a system:

1. **Guest-owned device errors** are errors caused by a partition-owned device that may assert an error condition. This type of error is specific to a particular device and should be handled by the device driver managing the device. The hypervisor routes error interrupts of this type to the partition owning the device and they appear as PIC-based interrupts to the partition. Error interrupts are represented in the partition's guest device tree as per the device tree binding for the device and are handled like any other device interrupt.
2. **Partition errors** are system errors that must be addressed by the partition responsible for causing the error. These errors are placed in a *guest event queue* for handling by the guest OS, and the partition is notified by a machine check interrupt. An example of this type of error is an IOMMU access violation where an incorrectly programmed device attempts to make an illegal DMA access. Partition errors also includes CPU machine check conditions (e.g. cache parity error) that occur while a guest operating system is running.
3. **System Errors** are errors that may be system-wide in scope and may be non-recoverable. These errors each have a configurable policy (see Error Policy Configuration on page 2233), with one option being to log the error in a *global event queue* for handling by a partition designated to be an "error manager".

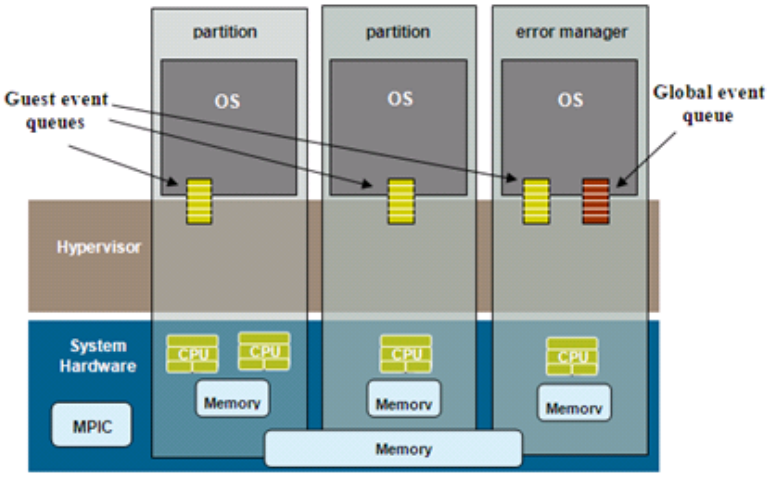


Figure 351. Global Event Queue

The error manager is designated by specifying the *error-manager* property on the partition node of the designated partition (see Partition Node Properties on page 2215).

The policy for each type of error can be configured in error configuration nodes in the hypervisor device tree. See Error Policy Configuration on page 2233.

11.1.2.6.8.2 Event notification interrupts

Guests are notified of errors in the guest event queue via a machine check (MCSR[MCP], IVOR1).

The error manager is notified of errors in the global event queue via a critical interrupt (IVOR0).

To handle both types of interrupts, operating systems invoke the FH_ERR_GET_INFO hcall to get the error event information.

11.1.2.6.8.3 Error Management Guest Device Tree Representation

The guest event queue is represented by a child node of the /**hypervisor** node in the guest device tree. The guest event queue node is represented with the following properties:

Table 482. Guest Event Queue Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Will be the value "fsl,hv-guest-error-queue". Defines a node used for accessing the guest event queue.
reg	R	<u32>	Specifies a handle that can be used in the fh_err_get_info hcall.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

The global event queue is represented in the guest device tree of an error manager partition with the following properties:

Table 483. Global Event Queue Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Will be the value "fsl,hv-error-manager". Specifies that this partition is an error manager.
reg	R	<u32>	Specifies a handle that can be used in the fh_claim_device or fh_err_get_info hcalls.
fsl,hv-claimable	SD	<string>	Will be present if 2 partitions are sharing error management responsibilities in an active/standby configuration. Valid values are: "active" - The partition is the active error manager "standby" - The partition is the standby error manager
fsl,hv-error-manager-cpu	R	<u32>	The virtual CPU number that will get critical interrupts indicating data in the global event queue.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.6.8.4 Error Domains, Error Types, Error Policies

Partition errors and system errors are classified into different error domains. Each domain has a unique type of error structure that is returned. The *error domain strings* and *error type strings* identify the domain and specific error.

The string identifiers are used to configure error policies and identify the error when an error is returned.

Table 484. Error Domains

Error domain String	Description
"ccf"	CoreNet coherency fabric errors
"misc"	Miscellaneous errors
"cpc"	CoreNet platform cache errors

Table continues on the next page...

Table 484. Error Domains (continued)

Error domain String	Description
"pamu"	Errors associated with the Peripheral Management Access Unit (an IOMMU)
"ddr"	DDR memory controller errors
"machine check"	Recoverable or guest machine check

Table 485. Error Policies

Policy	Policy Description
disable	The error condition is to be suppressed. Hardware will be configured so that the error is disabled.
notify	Place an error event in the global event queue, clear the error condition, and continue.
halt	Upon detection of the error condition, dump current state to the debug console, disable interrupts, and halt the hypervisor. If the hypervisor watchdog is enabled this will ultimately result in a watchdog timeout.
system-reset	Assert HRESET to reset the system.

11.1.2.6.8.5 FH_ERR_GET_INFO

Hypercall: FH_ERR_GET_INFO

Description: Removes an error event from the specified event queue. The caller provides the address of a physically contiguous buffer where the structure is returned with the details of the error.

Arguments:

Table 486. FH_ERR_GET_INFO Arguments

r11	hcall-token	FH_ERR_GET_INFO
r3	handle	Value from the reg property from the fsl,hv-guest-error-queue or fsl,hv-error-manager properties in the guest device tree.
r4	buf-size	maximum size (in bytes) of the error buffer
r5	errbuf-addr-hi	specifies the high order 32-bits of the guest physical address of the location to store the error structure.
r6	errbuf-addr-lo	specifies the low order 32-bits of the guest physical address of the location to store the error structure.

Table continues on the next page...

Table 486. FH_ERR_GET_INFO Arguments (continued)

r7	peek	Specifies whether to remove the entry or not. Valid values are: 0 - remove the first event from the queue 1 - read, but don't remove the first event from the queue All other values are reserved. Note: if an event is not removed from the queue the related interrupt will be reasserted when the ISR returns.

Return values:

Table 487. FH_ERR_GET_INFO return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_ENOENT : the queue was empty EV_EINVAL: invalid parameter EV_EFAULT - bad guest address
r4	buf-size	actual length (in bytes) of the error struct returned at the specified address

C API:

Table 488. FH_ERR_GET_INFO C API

unsigned int **fh_get_err_info**(uint32_t queue_select, uint32_t buf_size, uint64_t error_buf, uint32_t peek)

11.1.2.6.8.5.1 Error Structure

Error information returned by FH_GET_ERR_INFO is returned in an error structure. A portion of the structure is common to all error types and each error domain has unique fields that are represented in a sub-structure.

The error domain structures are specified in following sections.

Common error struct fields:

```
typedef struct {
    char domain[32];
    char error[128];
    char hdev_tree_path[256];
    char gdev_tree_path[256];
    union {
        // error domain structs go here
    };
} hv_error_t;
```

The common fields are defined as follows

Table 489. Error Structure common fields

Fields	Definition
domain	A character string that defines the error domain. See Error Domains, Error Types, Error Policies on page 2190 for a list of domains.
error	A character string that defines the error. See PAMU error domain on page 2193 for a list of error types.
hdev_tree_path	A string that specifies the path in the hardware device tree to the node that caused the error.
gdev_tree_path	A string that specifies the path in the guest device tree to the node that caused the error.

11.1.2.6.8.6 PAMU error domain

The PAMU error domain represents error conditions related to the Peripheral Management Access Unit.

Table 490. Error Types

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Operation error	pamu	"operation"	An operation error occurred in PAMU. An unsupported ATM or OTM code was read, or bad data was read due to an error in some other block (e.g. DDR or CPC). This error may or may not be fatal to a system depending on the device that PAMU was attempting to authorize when the error occurred.	disable, notify, halt, system-reset	notify
Single bit ECC error	pamu	"single-bit ecc"	A single-bit ECC error occurred accessing the PAMUs internal cache. The *single-bit-ecc-threshold* property may be specified on the error configuration node to set the policy of how many single bit ECC errors are to be tolerated before an error is asserted by the hardware. See Error Policy Configuration on page 2233 . The default threshold is 8.	disable, notify, halt, system-reset	disable

Table continues on the next page...

Table 490. Error Types (continued)

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Multiple-bit ECC error	pamu	"multi-bit ecc"	A multi-bit ECC error occurred accessing the PAMUs internal cache. This error is recoverable-the PAMU reverts to reading main memory if there is a cache failure.	disable, notify, halt, system-reset	disable
Access violation	pamu	"access violation"	A device made an unauthorized DMA access. Further DMA from the device will be disabled and an explicit FH_DMA_ENABLE hcall must be used to re-enable DMA.	disable, notify	notify

A structure contained within the general error structure (see) defines fields related to PAMU error conditions.

PAMU error struct fields:

```
struct pamu_error{
        uint32_t lpid;
        uint64_t access_violation_addr;
        uint32_t avs1;
        uint32_t avs2;
        uint32_t liodn_handle;
        uint32_t eccctl;
        uint32_t eccdis;
        uint32_t eccinten;
        uint32_t eccdet;
        uint32_t eccattr;
        uint64_t eccaddr;
        uint64_t eccdata;
        uint32_t poes1;
        uint64_t poeaddr;
} pamu_error_t;
```

The PAMU domain fields are defined as follows:

Table 491. PAMU error fields

Fields	Definition	Errors for Which the Field is Valid
lpid	The Partition ID (LPID) of the partition that caused the error.	access violation
access_violation_addr	The address being accessed that caused the access violation	access violation

Table continues on the next page...

Table 491. PAMU error fields (continued)

Fields	Definition	Errors for Which the Field is Valid
avs1	The value in the *Access Violation Status 1* Register. See the processor hardware reference manual for the encoding of this field.	access violation
avs2	The value in the *Access Violation Status 2* Register. See the processor hardware reference manual for the encoding of this field.	access violation
liodn_handle	LIODN handle to the device that caused the error. This refers to the handle for the DMA-owning guest. If no valid LIODN handle exists this value is 0xffffffff.	access violation
eccctl	See the PAMU EECTL register in the hardware reference manual for definition.	single-bit ecc, multi-bit ecc
eccdis	See the PAMU EEDIS register in the hardware reference manual for definition.	single-bit ecc, multi-bit ecc
eccinten	See the PAMU EEINTEN register in the hardware reference manual for definition.	single-bit ecc multi-bit ecc
eccdet	See the PAMU EEDET register in the hardware reference manual for definition.	single-bit ecc multi-bit ecc
eccattr	See the PAMU EEATTR register in the hardware reference manual for definition.	single-bit ecc multi-bit ecc
eccaddr	The address that encountered an ECC error while reading from the cache	single-bit ecc multi-bit ecc
eccdata	See the PAMU EEDHI and EEDLO register in the hardware reference manual for definition.	single-bit ecc, multi-bit ecc
poes1	See the PAMU POES1 register in the hardware reference manual for definition.	operation
poeaddr	See the PAMU POEAH and POEAL register in the hardware reference manual for definition.	operation

11.1.2.6.8.7 CCF Error Domain

The CCF error domain represents error conditions related to the CoreNet Coherency Fabric.

Table 492. CCF Error Types

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Multiple intervention response	ccf	"multiple intervention"	Multiple interventions occurred for a given transaction.	disable, notify, halt, system-reset	disable
Local access error	ccf	"local access"	A transaction missed in a local access window.	disable, notify, halt, system-reset	disable

A structure within the general error structure (see Error Structure on page 2192) defines fields related to CCF error conditions.

The fields are shown below:

```
struct ccf_error {
        uint32_t cedr;
        uint32_t ceer;
        uint32_t cecar;
        uint64_t cecaddr;
        uint32_t cmecar;
} ccf_error_t;
```

The CCF error fields are defined as follows:

Table 493. CCF error fields

Fields	**Definition**	**Errors for Which the Field is Valid**
cedr	See the CCF CEDR register in the hardware reference manual for definition.	multiple intervention, local access
ceer	See the CCF CEER register in the hardware reference manual for definition.	multiple intervention, local access
cecar	See the CCF CECAR register in the hardware reference manual for definition.	multiple intervention, local access
cecaddr	See the CCF CECADRH and CECADRL registers in the hardware reference manual for definition.	multiple intervention, local access
cmecar	See the CCF CECAR2 register in the hardware reference manual for definition.	multiple intervention, local access

11.1.2.6.8.8 CPC Error Domain

The CPC error domain represents error conditions related to the CoreNet Platform Cache.

Table 494. CPC Error Types

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Tag multi-way hit	cpc	"tag multi-way hit"	A tag multi-way hit occurred.	disable, notify, halt, system-reset	disable
Tag or status multiple-bit ECC	cpc	"tag status multi-bit ecc"	A multi-bit ECC error occurred in a tag or status field.	disable, notify, halt, system-reset	disable
Tag or status single-bit ECC	cpc	"tag status single-bit ecc"	A single-bit ECC error occurred in a tag or status field. The *single-bit-ecc-threshold* property may be specified on the error configuration node for this error to set the policy of how many single bit ECC errors are to be tolerated before an error is asserted by the hardware. See Error Policy Configuration on page 2233. The default threshold is 8.	disable, notify, halt, system-reset	disable
Data multiple-bit ECC error	cpc	"data multi-bit ecc"	A multi-bit ECC error occurred in data.	disable, notify, halt, system-reset	disable
Data single-bit ECC error	cpc	"data single-bit ecc"	A single-bit ECC error occurred in data. The *single-bit-ecc-threshold* property may be specified on the error configuration node for this error to set the policy of how many single bit ECC errors are to be tolerated before an error is asserted by the hardware. See Error Policy Configuration on page 2233. The default threshold is 8.	disable, notify, halt, system-reset	notify

Table continues on the next page...

Table 494. CPC Error Types (continued)

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Configuration error	cpc	"config"	There was a platform cache configuration error.	disable, notify, halt, system-reset	disable

A structure within the general error structure (see Error Structure on page 2192) defines fields related to PAMU error conditions.

CPC error struct fields:

```
struct cpc_error {

        uint32_t cpcerrdet;

        uint64_t cpcerraddr;

        uint32_t cpcerrattr;

        uint32_t cpcerrctl;

        uint32_t cpcerrinten;

        uint32_t cpcerrdis;

        uint32_t cpccaptecc;

} cpc_error_t;
```

The common fields are defined as follows:

Table 495. CPC error fields

Fields	Definition	Errors for Which the Field is Valid
cpcerrdet	See the CPC CPCERRDET register in the hardware reference manual for definition.	all CPC error types
cpcerraddr	See the CPC CPCERRADDR and CPCERREADDR registers in the hardware reference manual for definition.	all CPC error types
cpcerrattr	See the CPC CPCERRATTR register in the hardware reference manual for definition.	all CPC error types

Table continues on the next page...

Table 495. CPC error fields (continued)

Fields	Definition	Errors for Which the Field is Valid
cpcerrctl	See the CPC CPCERRCTL register in the hardware reference manual for definition.	all CPC error types
cpcerrinten	See the CPC CPCERRINTEN register in the hardware reference manual for definition.	all CPC error types
cpcerrdis	See the CPC CPCERRDIS register in the hardware reference manual for definition.	all CPC error types
cpccaptecc	See the CCF CPCCAPTECC register in the hardware reference manual for definition.	all CPC error types

11.1.2.6.8.9 DDR Error Domain

The DDR error domain represents error conditions related to the DDR memory controllers.

Table 496. DDR Error Types

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Memory select error	ddr	"memory select"	A transaction request was issued to the DDR memory controller and the address does not lie within any of the programmed address ranges for an enabled chip select.	disable, notify, halt, system-reset	disable
Single bit ECC error	ddr	"single-bit ecc"	A single bit ECC error occurred and exceeded the configured threshold. The *single-bit-ecc-threshold* property may be specified on the error configuration node for this error to set the policy of how many single bit ECC errors are to be tolerated before an error is asserted by the hardware. See Error Policy Configuration on page 2233. The default threshold is 8.	disable, notify, halt, system-reset	disable
Multiple-bit ECC error	ddr	"multi-bit ecc"	A multi bit ECC error occurred at the memory controller.	disable, notify, halt, system-reset	disable

Table continues on the next page...

Table 496. DDR Error Types (continued)

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Corrupted data error	ddr	"corrupted data"	A data corruption error was detected.	disable, notify, halt, system-reset	disable
Automatic calibration error	ddr	"auto calibration"	This error is set if the memory controller detects an error during its training sequence.	disable, notify, halt, system-reset	disable
Address parity error	ddr	"address parity"	An address parity error was detected.	disable, notify, halt, system-reset	disable

A structure within the general error structure (see Error Structure on page 2192) defines fields related to DDR error conditions.

DDR error struct fields:

```
struct ddr_error {
        uint32_t ddrerrdet;
        uint32_t ddrerrdis;
        uint32_t ddrerrinten;
        uint32_t ddrerrattr;
        uint64_t ddrerraddr;
        uint32_t ddrsbeccmgmt;
        uint32_t ddrcaptecc;
} ddr_error_t;
```

The common fields are defined as follows:

Table 497. DDR error fields

Fields	Definition	Errors for Which the Field is Valid
ddrerrdet	See the DDR controller ERR_DETECT register in the hardware reference manual for definition.	all DDR error types
ddrerrdis	See the DDR controller ERR_DISABLE register in the hardware reference manual for definition.	all DDR error types
ddrerrinten	See the DDR controller ERR_INT_EN register in the hardware reference manual for definition.	all DDR error types
ddrerrattr	See the DDR controller CAPTURE_ATTRIBUTES register in the hardware reference manual for definition.	all DDR error types
ddrerraddr	See the DDR controller CAPTURE_ADDRESS and CAPTURE_EXT_ADDRESS registers in the hardware reference manual for definition.	all DDR error types
ddrsbeccmgmt	See the DDR controller ERR_SBE register in the hardware reference manual for definition.	all DDR error types
ddrcaptecc	See the DDR controller CAPTURE_ECC register in the hardware reference manual for definition.	all DDR error types

11.1.2.6.8.10 Miscellaneous Error Domain

The "misc" error domain represents miscellaneous error conditions that don't fit into the other domain types.

Table 498. Miscellaneous Error Types

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Internal RAM multi-bit ECC	misc	"internal ram multi-bit ecc"	An unspecified block in the SOC had a multi-bit ECC error. This error is catastrophic. The *system-reset* policy for this error utilizes a hardware mechanism that will reset the SOC directly with no hypervisor software intervention required.	disable, notify, halt, system-reset	disable

11.1.2.6.8.11 Machine Checks

Error conditions detected in a CPU may result in a machine check exception. See the CPU reference manual for a complete description of the hardware implementation of the interrupt model and machine check conditions in the CPU.

Guest Machine Checks

All machine check exceptions that occur while a CPU is executing in guest state are reflected to the guest partition to be handled by the guest's machine check handler, except for ICPERR and DCPERR which are recoverable and handled by the hypervisor and not reflected.

Guest machine checks are logged to the global event queue with the "machine check" error domain.

Hypervisor Machine Checks

Any machine check exception that occurs in the hypervisor (while the CPU is executing in hypervisor state, MSR[GS]=0) is non-recoverable, except for the following:

- ICPERR - Instruction cache tag or data array parity error. Recovery is automatically handled by the CPU hardware.
- DCPERR - Data cache data or tag parity error due to a load. If the L1 cache is in write-shadow mode the error is recoverable by the hypervisor. If the L1 cache is not in write-shadow mode the error is non-recoverable
- IF - Instruction fetch error report. IF is treated as non-fatal if accompanied by ICPERR.

Non-recoverable errors result in the hypervisor dumping register state to the console and halting all CPUs with interrupts disabled. If the hypervisor watchdog is enabled (see section Hypervisor Watchdog on page 2203) the system hardware will be reset.

Recoverable machine checks that occur in hypervisor state are logged to the global event queue with the "machine check" error domain.

Table 499. Machine checks

Error condition	Error Domain	Error Type String	Description	Allowable Policies	Default Policy
Machine check	machinecheck	"machine check"	A recoverable machine check condition was detected, or a machine check occurred in a guest partition.	notify	notify

A structure within the general error structure (see Error Structure on page 2192) defines fields related to DDR error conditions.

```
struct mcheck_error {

        uint32_t mcsr;

        uint64_t mcar;

        uint64_t mcsrr0;

        uint32_t mcsrr1;

} mcheck_error_t;
```

Table 500. Machine check error fields

Fields	Definition
mcsr	Value of the machine check syndrome register when the error occurred. See the CPU reference manual for specific field definitions.

Table continues on the next page...

Table 500. Machine check error fields (continued)

mcar	Value of the machine check address register when the error occurred. See the CPU reference manual for specific field definitions.
mcsrr0	Effective address of the instruction executing or about to execute when the error occurred. See the CPU reference manual for additional details.
mcsrr1	Value of the machine state register (MSR) when the error occurred.

11.1.2.6.8.12 Hypervisor Watchdog

The hypervisor implements a watchdog timer to detect severe system failures that require a system reset. If the watchdog is configured, the CPU's watchdog timer is enabled on all CPUs and the watchdog interrupt will fire periodically. The TSR[WIS] will be reset from within the watchdog interrupt handler. If the watchdog interrupt handler fails to run, the watchdog will expire and a system hardware reset will occur.

This approach enables recovery from unexpected exception conditions encountered in the hypervisor that result in a crash. Examples include:

- Memory corruption
- Severe hardware errors that prevent the normal exception handlers from running

See the *watchdog-enable* property defined in Hypervisor configuration node on page 2231.

The hypervisor watchdog timer handler also executes a customizable system health monitor routine that can periodically monitor system specific conditions to determine whether the system is in a healthy state. See System Health Monitor on page 2260 for details on how to customze the system health monitor.

11.1.2.6.9 Power management

11.1.2.6.9.1 Power Management Overview

The hypervisor provides mechanisms that allow partitions to access the power saving features in the processor hardware for controlling CPU power states.

A virtual CPU can be in one of several sleep states:

- RUN—the processor is running normally
- IDLE—instruction execution is suspended. CPUs request to enter this state with the FH_IDLE hcall and subsequent instructions initiate until an asynchronous interrupt occurs. The CPU is left in a standby mode in which cache snooping and time base interrupts are still enabled.
- NAP—all clocks to the functional units of the CPU are shut down, except the timebase. No cache snooping is performed in nap mode. CPUs enter this state with the FH_ENTER_NAP hcall. In NAP state decrementer, FIT, watchdog, and external interrupts are masked at the CPU by the hypervisor until the CPU is woken. Return to "run" state occurs when another CPU wakes up the napping CPU with the FH_EXIT_NAP hcall. NAP state is intended to be used for temporarily disabling **secondary** CPUs.

The FH_GET_CORE_STATE hcall allows a partition or a manager partition to read the sleep state of a CPU.

11.1.2.6.9.2 FH_GET_CORE_STATE

Hypercall: FH_GET_CORE_STATE

Description: Gets the sleep state for the virtual CPUs in a partition. Possible states are "run", "idle", and "nap".

Arguments:

Table 501. FH_GET_CORE_STATE Arguments

r11	hcall-token	FH_GET_CORE_STATE
r3	handle	Handle to the partition (-1 indicates current partition).
r4	vcpu	Number of the virtual cpu within the partition

Return values:

Table 502. FH_GET_CORE_STATE return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_ENOENT : the queue was empty EV_EINVAL: invalid parameter
r4	state	sleep state of the virtual CPU 0 : run 1 : idle 2 : nap

C API:

Table 503. FH_GET_CORE_STATE C API

unsigned int **fh_get_core_state**(unsigned int handle, unsigned int vcpu, unsigned int *state)

11.1.2.6.9.3 FH_ENTER_NAP

Hypercall: FH_ENTER_NAP

Description: A request put the specified virtual CPU to into NAP mode. Note, some implementations may only support the caller's virtual CPU as an argument and return EINVAL if other CPU numbers are specified.

Arguments:

Table 504. FH_ENTER_NAP Arguments

r11	hcall-token	FH_ENTER_NAP
r3	handle	Handle to the partition (-1 indicates current partition).
r4	vcpu	Number of the virtual cpu within the partition

Return values:

Table 505. FH_ENTER_NAP return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL: invalid parameter

C API:

Table 506. FH_ENTER_NAP C API

unsigned int **fh_enter_nap**(unsigned int handle, unsigned int vcpu)

11.1.2.6.9.4 FH_EXIT_NAP

Hypercall: FH_EXIT_NAP

Description: A request to remove the specified virtual CPU from NAP mode.

Arguments:

Table 507. FH_EXIT_NAP Arguments

r11	hcall-token	FH_EXIT_NAP
r3	handle	Handle to the partition (-1 indicates current partition).
r4	vcpu	Number of the virtual cpu within the partition

Return values:

Table 508. FH_EXIT_NAP return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL: invalid parameter

C API:

Table 509. FH_EXIT_NAP C API

unsigned int **fh_exit_nap**(unsigned int handle, unsigned int vcpu)

11.1.2.6.10 High availability services

11.1.2.6.10.1 High Availability Service Overview

The hypervisor provides mechanism to create configurations where partitions can be arranged in an active and standby relationship. These mechanisms include:

- Notifications on managed partition state changes and watchdog expiration (see Partition Management Interrupts on page 2175)

- Device sharing between active and standby partitions. After an active partition crash, mechanisms exist for the new active partition to claim active ownership of shared devices, include interrupt routing re-configuration and IOMMU updates
- Error manager sharing between active and standby
- System reset if all partitions move into the stopped state (see sysreset-on-partition-stop property in Hypervisor Configuration Overview on page 2209).

11.1.2.6.10.2 Device Claiming

Devices can be shared between partitions in a failover configuration, whereby one partition has active ownership of a device at a time. Interrupts are routed to the active owner. Errors such PAMU access violations are placed in the error event queue of the active owner of the device.

If the active partition is stopped, another partition configured with access to the device may claim it and become the new active owner. To use this feature, all partitions assigned the shared device must specify the *claimable* property on the device configuration node for the device (see Device nodes on page 2222). At least one partition must have a shared device initially set to be "active".

When the "new active" partition claims a device, the interrupt configuration (priority, destination cpu) for that device will be reset to the default state. It is the responsibility of the new active partition to modify the interrupt configuration if necessary.

Claiming a device may also involve some reconfiguration of the PAMU and require DMA to be enabled for the claimed device. See DMA and Shared Devices on page 2224 for details.

11.1.2.6.10.2.1 Guest Device Tree Properties

A device node in the guest device tree contains several hypervisor specific properties that enable device drivers to determine status of a shared device and to claim a shared device on a failover.

Table 510. Guest Device Tree Shared Device Properties

Property Name	Usage	Value Type	Definition
fsl,hv-device-handle	SD	<u32>	May be present in a device node and provides a handle to perform device operations. Can be used in an fh_claim_device hcall.
fsl,hv-claimable	SD	<string>	Will be present in nodes identified as claimable in the hypervisor config tree. There are two possible values: "active"—The device is active—device interrupts and DMA errors are routed to the partition. "standby"—The device is 'standby' to this partition and should not be accessed. Interrupts and DMA errors will not be directed to a standby partition.

Table continues on the next page...

Table 510. Guest Device Tree Shared Device Properties (continued)

Property Name	Usage	Value Type	Definition
status	SD	<string>	Specifies the initial status of a device and will be present in a device node that is standby. It may be present in any other node. If the device is specified as "active" in the hypervisor configuration tree, any existing "status" property value set by boot firmware will be left as is, including possibly no status at all. If the device is specified as "standby" in the hypervisor config tree, the "status" property value will set to "disabled" if the existing status (in the hardware device tree) is "okay" or absent. If there is a non-ePAPR-compliant status that doesn't begin with "ok" it will be left alone.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.6.10.2.2 FH_CLAIM_DEVICE

Hypercall: FH_CLAIM_DEVICE

Description: The fh_claim_device hcall may be used to claim active ownership of a shared device or active ownership of error management.

For devices, invoking this hcall will result in the following changes:

- Interrupts for the device will be routed to the partition claiming ownership
- The PAMU may be reconfigured with updated true physical addresses of DMA windows if necessary (see DMA and Shared Devices on page 2224)
- Device specific errors, such as PAMU access violations, will be sent to the guest event queue of the partition claiming ownership.

In order to claim ownership of an interrupt inside an interrupt-map, the node containing the map has to be claimed, not the device node that maps to it.

If the target device is not already owned/claimed, an error will be returned.

In order to claim a device the partition currently owning the device must be in a "stopped" state.

Arguments:

Table 511. FH_CLAIM_DEVIC Arguments

r11	hcall-token	FH_CLAIM_DEVICE
r3	handle	Value from the fsl,hv-device-handle property in the guest device tree.

Return values:

Table 512. FH_CLAIM_DEVIC return values

r3	status	Status of the hypercall 0 : the operation completed successfully EV_EINVAL: invalid parameter EV_ERR_INVALID_STATE : if partition sharing device is not stopped

C API:

Table 513. FH_CLAIM_DEVIC C API

unsigned int **fh_claim_device**(unsigned int handle)

11.1.2.6.11 Warm boot

11.1.2.6.11.1 Warm Boot Overview

Some systems may have mechanisms that allow the system to be warm booted—i.e. the system is restarted from the bootloader, but no hardware reset of the system or CPUs is required. In a warm boot memory and I/O devices may not need to be reinitialized. It may be required that the system be kept alive across a warm boot which means that certain regions of memory must maintain their contents and any I/O devices doing DMA must continue to operate normally and must not be stopped. During the warm boot the hypervisor image would be reloaded and the hypervisor would be started from its entry point.

A system can potentially use a warm boot to upgrade the hypervisor itself, where the hypervisor image loaded during the warm boot process is a new, upgraded image.

11.1.2.6.11.2 Warm Boot Details

A custom, system-specific mechanism must be created to perform the warm boot of a system. This may involve a custom hypercall or a customization of the fh_system_reset() hypercall (see Custom Reset on page 2260).

Warm boot support is enabled/disabled in the hypervisor at compile time with the CONFIG_WARM_REBOOT kconfig option.

When the warm boot feature is used the following should be noted:

- The activation of warm boot mode is done by a hypervisor command line parameter "warm-reboot", passed using the hardware device tree *bootargs* property (see Bootargs on page 2210). If this parameter is present, after verification of the integrity of the hypervisor- persistent data, the hypervisor initialization process bypasses any initialization that would affect any in-flight DMA.
- The Peripheral Access Management Unit (PAMU) tables are maintained in a distinct "persistent data" area defined within the hypervisor private memory. This region must be defined by the system architect in the hypervisor configuration tree with the *hv-persistent-data* property. This persistent region contains additional information necessary to complete the warm-reboot phase. The hypervisor owns its content, which includes a version number, and it must not be modified. Refer to Hypervisor Private Memory on page 2232 for information on the *hv-persistent-data* property.
- The *zero-pma* property (see PMA Hypervisor Configuration Tree Representation on page 2211) which directs the hypervisor to set the content of a guest physical memory area to zero is ignored on a warm-reboot phase in order to preserve the memory's contents.

The following additional assumptions should be noted:

- It is assumed that the configuration tree passed to the hypervisor on the warm boot is identical to the configuration tree passed at cold boot. This means that all device and memory partitioning is identical and that the memory requirements of an upgraded hypervisor must fit within the memory allocated to the hypervisor in that configuration tree.

- It is assumed that the configuration tree passed will reflect the active/standby state of the system at the time it was warm booted—i.e the "claimable" property on the device node specifies the active owner.
- Any guest device tree updates made with the FH_PARTITION_SET_DTPROP hcall will be lost across a warm reboot.

11.1.2.7 Configuration

The hypervisor software is configurable via a combination of compile time and static configuration parameters passed at boot time. This chapter describes the configuration parameters passed to the hypervisor at boot time.

11.1.2.7.1 Hypervisor configuration

11.1.2.7.1.1 Hypervisor Configuration Overview

The hypervisor is passed a hardware device tree when it is booted that describes all the hardware in the system.

A separate binary blob of configuration information is passed by address to the hypervisor using the bootargs parameter on the hardware device tree's */chosen* node. The hypervisor configuration tree contains all hypervisor configuration parameters and partition definition information. Based on the information in the hypervisor configuration tree the hypervisor will dynamically create guest device trees for each partition.

The hypervisor configuration tree is physically a device tree, but is in a custom form for hypervisor use only and is not ePAPR compliant.

The hypervisor configuration tree specifies all configurable parameters and all partitions, including:

- Physical memory areas (PMAs)—specifies how physical memory is partitioned
- DMA windows—a set of tables that each define a guest physical map of regions to which DMA devices are allowed to access
- Byte-channel mux—the declaration of any byte-channel muxes
- Doorbells—a table that declares all doorbells
- General hypervisor configuration parameters
- Partition declaration—each partition has a partition node that specifies properties of the partition. For example:
  - Set of physical CPUs in the partition
  - List of guest physical memory areas (each referencing a physical memory area)
  - Device nodes—a set of nodes which assign I/O devices and portals to a partition. For DMA-capable devices the node describes which DMA window table the device should use
  - Image load table
  - Guest address where the device tree should be placed
  - Doorbells assigned to the partition
  - Byte-channels—a table that defines all byte-channels in the partition and the uart or mux endpoints each is attached to
  - Guest aliases
  - Guest chosen node
  - Managed partition relationships

11.1.2.7.1.2 Hypervisor configuration- General

11.1.2.7.1.2.1 Hypervisor Configuration Tree

The hypervisor configuration tree is represented as a device tree. However, it does not describe hardware and is not ePAPR compliant. It is specifically designed for hypervisor configuration.

All addresses in the hypervisor configuration tree are 64-bits and consist of two cells.

The root node of the hypervisor configuration tree must contain a compatible property that identifies the device tree as a hypervisor:

Table 514. Root Mode Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "fsl-hv-config"
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.1.2.2 Bootargs

When the hypervisor is booted by boot firmware it must be passed the address of the hypervisor configuration tree in the *bootargs* property of the /chosen node in the hardware device tree. Multiple parameters in the command line are separated by spaces.

The bootargs property must include the following value:

```
config-addr=[physical address of device tree]
```

The physical address may be expressed in decimal, hexadecimal (0xNNN), or binary (0bNNN).

An optional bootargs parameter is:

```
warm-reboot
```

This indicates that the hypervisor is being started following a warm system boot in warm boot mode. See Warm boot on page 2208 for details.

11.1.2.7.1.3 Physical memory areas (PMA)

11.1.2.7.1.3.1 PMA Overview

A key aspect of partitioning a system is to partition the system's physical memory. To do this physical memory is divided into physical memory areas (or PMAs). PMAs are statically defined by the system architect and cannot be changed dynamically in a running system. The hypervisor itself is also assigned a PMA for its internal use.

A PMA is defined with the following properties:

- True physical address (must be size aligned)
- Size (must be a power of 2)
- CPC partitioning—specifies one or more ways of the Corenet platform cache (CPC) to be allocated and dedicated to this PMA

Each PMA corresponds to a CoreNet coherence domain (CSDID). Coherence is implied—if a PMA is only referenced by one partition, the PMA is private to that partition and only CPUs within that partition are snooped to maintain cache coherency. For shared PMAs, all partitions that reference the PMA will have their CPUs listed in the CSDID for the PMA.

NOTE

The P4080 supports up to 32 CSDIDs. 3-4 will be reserved for hypervisor use and the remainder are available for use in partitioning—leaving about 28 PMAs allowed.

The hypervisor supports a mechanism to initialize a PMA to zeros during a cold boot of a system—see the *zero-pma* property.

11.1.2.7.1.3.2 PMA Hypervisor Configuration Tree Representation

Table 515. PMA Configuration tree Representation

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "phys-mem-area"
addr	R	<u64>	The true physical (real) address of the PMA
size	R	<u64>	The size of the PMA-must be a power of 2
allocate-cpc-ways	O	<prop-encoded-array>	An array of <u32> values each specifying a way of the CPC to allocate. Note: by default the Corenet platform cache (CPC) allocates all ways to all PMAs. If the *allocate-cpc-ways* property is specified for a PMA, the selected ways will be removed from the default pool of ways (used by all other PMAs) and only access to the PMAs that specifically reference the ways (using the *allocate-cpc-ways* property) will allocate from them.
zero-pma	O	<none>	If present causes the PMA to be initialized to zeros during the cold boot of a system. Note: if the hypervisor is being booted in warm-boot mode (see Warm boot on page 2208) this option is ignored.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.1.4 DMA windows

11.1.2.7.1.4.1 DMA Windows Overview

The IOMMU (PAMU) requires careful configuration to specify the allowable regions of guest physical address space that can be targeted by DMA-capable devices.

PAMU supports defining power-of-2 sized memory DMA windows which can be optionally subdivided into a power-of-2 number of subwindows. Please consult the chip Reference Manual for the maximum number of supported sub-windows. Subwindows enable the definition of regions that may be the target of DMA, but are discontiguous in guest physical space or true physical space.

In the context of the hypervisor, a DMA window is conceptually a table of guest physical memory regions that are the valid targets of a DMA operation. Each row of the table consist of a guest physical address and size

If a single address window is needed the DMA window is simply a table with one row defining the region.

If multiple, discontiguous (guest physical **or** true physical) address regions are needed, DMA sub-windows are used to represent and configure the regions. In this case, the DMA window is conceptually a table with a power-of-2 number of rows, with each row defining one sub-window. There are several key parameters in defining sub-windows that need to be understood:

- **Total size**. This is the total size of the overall DMA region being defined, encompassing all sub-windows. This must be a power of 2.
- **Base address**. This is the guest physical address at which the overall DMA region starts. This must be size aligned with the total size.

- **Sub-window count**. The number of sub-windows must be power-of-2. Please consult the chip Reference Manual for the maximum number of supported sub-windows.
- **Maximum sub-window size**. The maximum size of each subwindow is computed by: total-size / subwindow-count.
- **Sub-window starting address**. Each sub-window starts at an address computed by:
- overall-starting-address + (subwindow index * maximum-subwindow-size)
- **Sub-window size**. Each sub-window can specify a size that may be smaller than the maximum subwindow size.

An DMA window describes a particular configuration of guest address space and may be applicable to one or more partitions.

DMA windows are referenced by DMA-capable I/O devices in their device nodes. Each DMA-capable device expresses one or more logical I/O device numbers (LIODN) which uniquely identify the device. Each LIODN used must have a DMA window configured for it.

11.1.2.7.1.4.2 DMA Windows-Hypervisor Configuration Tree Representation

DMA windows are defined with the properties described in the following table:

Table 516. DMA Windows-Hypervisor Configuration Tree Representation

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "dma-window"
guest-addr	R	<u64>	The guest physical address of the base DMA window. Must be size-aligned with the size property
size	R	<u64>	The size of the base DMA window—must be a power of 2 between 4KiB and 64GiB.
subwindow-count	SD	<u32>	Must be present if subwindows are used. Value specifies the number of subwindows. Must be a power-of-2 number. Please consult the chip Reference Manual for the maximum number of supported subwindows.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

The dma-window node may optionally have child nodes describing subwindows. Subwindows are defined as child nodes of the dma-window node. Subwindow nodes are defined with the following properties:

Table 517. DMA Sub-windows-Hypervisor configuration tree Representation

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "dma-subwindow"
guest-addr	R	<u64>	The guest physical address of the subwindow. This address must fall within the base DMA window (see 6-3) and be aligned to the base DMA window size divided by the subwindow-count (see size and subwindow-count property in 6-3)
size	R	<u64>	The size of the subwindow—must be a power of 2 between 4KiB and 64GiB.

Table continues on the next page...

Table 517. DMA Sub-windows-Hypervisor configuration tree Representation (continued)

pcie-msi-subwindow	SD	<none>	The property value is a boolean and its presence specifies that the subwindow is to be used for PCI-E MSIs. The address of the MPIC MSIIRx register in this subwindow will be conveyed to the guest in the guest device tree.
srio-ccsr-subwindow	SD	<none>	The property value is a boolean and its presence specifies that the subwindow is to be used for RapidIO accesses into CCSR space. The RapidIO maintenance transactions are translated into memory accesses to the CCSR space and they must be authorized by PAMU. The sub-window with this property must cover the CCSR address region for the RapidIO interface.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

It is not required that all subwindows be defined. Only the sub-windows that define valid address ranges must be defined. This allows the definition of noncontiguous regions of guest physical address space that are the valid targets of DMA operations.

11.1.2.7.1.5 Global Doorbell Definition

A set of global doorbell configuration nodes are used to declare doorbells.

Each global doorbell node in the hypervisor configuration tree has the properties described below. Up to four "fast doorbells" may be defined (see Fast Doorbells on page 2175). Fast doorbells have lower latency than regular doorbells. A fast doorbell is declared by adding the string "fast-doorbell" on the global doorbell node.

Table 518. Doorbell Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "doorbell" To specify this doorbell as a fast doorbell, must include the string "fast-doorbell".
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.1.6 Byte-channel Mux Configuration

A byte-channel multiplexer is defined at the root of the hypervisor configuration tree. The properties are specified in the table below: This node is referenced by byte-channels that use a mux as an endpoint.

Table 519. Byte-channel Multiplexer Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "byte-channel-mux"
endpoint	R	<phandle>	A phandle to the device node representing the uart to which the byte-channel mux is connected.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.1.7 Guest device tree node updates

11.1.2.7.1.7.1 Guest Device Tree Node Updates Overview

When the hypervisor creates a new partition, it uses the hardware device tree and the hypervisor configuration tree as input to instantiate the partition and to create a guest device tree for the partition. A "node update" mechanism is provided that allows a system architect to control the hypervisor's generation of nodes in the guest device tree.

Updates are defined in special node with the name *node-update or node-update-phandle* which are placed in the hypervisor configuration tree.

11.1.2.7.1.7.2 node-update

The node-update node allows guest device tree nodes and properties to be added, deleted, and updated. This node has the special name *node-update* and it is placed on nodes in the configuration device tree.

By default the node-update node is **merged** with the guest device tree node it is targeting. This means that the properties and nodes under the *node-update* node are copied exactly as is to the guest device tree. In addition, several additional directives are supported:

- deleting properties
- deleting a child node (or an entire sub-tree)
- deleting all child nodes of the specified node
- updating string properties, by prepending a string value

The following nodes in the hypervisor configuration tree support node-update nodes:

- Device nodes (see Device nodes on page 2222)
- Doorbell nodes (see Inter-partition Doorbell Configuration on page 2228)
- Byte-channel nodes (see Byte-Channel Configuration on page 2228)
- Partition nodes (see Partition Node Properties on page 2215)
- Guest physical memory area nodes (see Guest Physical Memory Areas on page 2221)
- Global error manager node (see Error Manager on page 2229)

The node-update node supports updates to descendants of the node which the *node-update* modifies. Updates to descendants can be done by replicating the node hierarchy under the node-update node using the names of the descendant nodes.

Except for the property names in the table below, all properties under *node-update* will be copied exactly as is to the guest device tree. If the property already exists in the node it will be replaced. Note: phandle references will not be resolved for properties with phandle values. See the *node-update-phandle* node (node-update-phandle on page 2215) for node updates to properties with phandle values.

In addition to copying properties, the following special properties allow some additional control:

Table 520. Guest Device Tree Node Special Properties

Property Name	Usage	Value Type	Definition
delete-prop	O	<stringlist>	The value identifies one or more names of properties that should be deleted from the node.
delete-node	O	<stringlist>	The value identifies one or more names of nodes that should be deleted. The node and all descendants will be deleted.

Table continues on the next page...

Table 520. Guest Device Tree Node Special Properties (continued)

delete-subnodes	O	<none>	A Boolean property. Applicable to device nodes (see Section). If present specifies that only the node itself and no descendants are to be copied to the guest device tree.
prepend-stringlist	O	<stringlist>	The value is a string list that specifies one or more property name/value pairs. "prop-name1," "prop-value1," "prop-name2," "prop-value2," Each name/value pair consists of two strings-the first is the name of the property, the second is a string to prepend to the property. If the property does not already exist it will be created.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

NOTE

phandles values in node-update nodes are not supported. See node-update-phandle on page 2215, for information on updating properties with phandle values.

11.1.2.7.1.7.3 node-update-phandle

The only supported mechanism for updating properties with phandle values is a special node with the name *node-update-phandle.*

Each property listed under *node-update-phandle* has a property name with a value that specifies the hypervisor configuration tree phandle of the referenced node. Arrays of phandles are supported, but it is not supported to mix phandles and non-phandle values in the same property.

If one property references another node by phandle the node referenced **must** be assigned to the partition.

11.1.2.7.2 Partition definition

11.1.2.7.2.1 Partition Node Properties

Each partition is defined by a partition node.

A partitioned is defined by a set of properties on the main partition node (see table below) plus several subnodes. The properties specify various partition parameters. The subnodes specify:

- The guest physical memory areas that exist and how they map to physical memory areas (PMAs)
- The I/O devices that are assigned to a partition (direct I/O).
- Virtual devices such as byte-channels, doorbells, and partition management nodes
- Aliases to add to the guest device tree's /alias node
- Debug stub nodes that define the relationship between byte-channels and CPUs in a partition
- Modifications to be made to properties on the guest device tree's /chosen node
- Modifications to be made to properties on the guest device tree's root node

A partition node has the following properties:

Table 521. Partition Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "partition."
label	O	<string>	Specifies a text string that describes the partition. This string may be used by partition management tools to display a readable label for a partition. The value of the *label* property is automatically propagated to the root node of the guest device tree.
cpus	SD	<prop-encoded-array>	Array defining which physical CPUs belong to the partition. This value is a <prop-encoded-array> encoded as an arbitrary number of pairs *(cpu-index, count)*. *cpu-index* is a <u32> that specifies a cpu index of a cpu in the hardware device tree (the index is identical to the value of the "reg" property in cpu nodes) *count* is a <u32> that specifies the number of CPUs being enumerated. Each CPU identified in the cpus field is enumerated starting with zero from left to right. Example: cpus = <4 2> This identifies two CPUs starting at index 4—i.e. CPUs # 4 and 5. These physical CPUs have logical indexes of 0 and 1 within the partition. Note: On cores with multiple threads per core (e.g. e6500 dual-threaded core) the hardware threads are individually assigned to the partitions. The hypervisor will expose the vcpu to guests as a single-threaded core.
dtb-window	R	<prop-encoded-array>	Specifies a memory window where the guest device tree is to be loaded in guest memory. The value is a <prop-encoded-array> encoded as: < dtb-addr-start dtb-size> Where: • *dtb-addr* is a <u64> that specifies the guest physical address specifying the start of a memory range where the guest device tree is to be loaded. • *dtb-size* is a <u64> value specifying the size of the memory window for the DTB.

Table continues on the next page...

Table 521. Partition Node Properties (continued)

Property Name	Usage	Value Type	Definition
guest-image	SD	<prop-encoded-array>	Specifies an image to be loaded by the hypervisor prior to partition start. This is the image to which the hypervisor transfers control when starting the partition. The value is a triplet defined as follows: <*image-src-addr image-dest-addr size*> Where: • *image-src-addr* is a <u64> that specifies the source true physical address of an image to be loaded into the partition. The image may be a binary image or may be in ePAPR ELF format. • *image-dest-addr* is a <u64> that specifies the guest physical address within the partition's main memory segment. This must be a word aligned address. For ELF images this value can be -1, which specifies that load address is to be determined from the ELF image itself (see below). • *size* is a <u64> and specifies the size of the source address window the image is located in. This value must be nonzero and could be different than the actual source image size. For binary images this field does specify the size of the image. The following images types are supported: • ELF (Executable and Linking Format) • If the *image-dest-addr* equals -1, for files with an ELF header e_type field value of ET_EXEC, each PT_LOAD segment will be loaded at the physical address given by the program headers p_paddr field. • the entry point into the image is calculated from the offset between the v_addr field and the e_entry field in the ELF header. • uImage (u-boot Image format)—the entry point into the image is calculated from the offset between the load address and entry point. • Binary. For binary images, offset 0x0 in the image is treated as the entry point.

Table continues on the next page...

Table 521. Partition Node Properties (continued)

Property Name	Usage	Value Type	Definition
linux-rootfs	SD	<prop-encoded-array>	Specifies the address and size of a Linux root filesystem image to be loaded by the hypervisor prior to partition start. The value is a triplet defined as follows: <*image-src-addr image-dest-addr size*> Where: • *image-src-addr* is a <u64> that specifies the source true physical address of an image to be loaded into the partition. • *image-dest-addr* is a <u64> that specifies the guest physical address within the partition's main memory segment. This must be a word aligned address. • *size* is a <u64> specifies the size of the source address window the image is located in. This value must be nonzero and could be different than the actual source image size. For binary images this field does specify the size of the image. The following images types are supported: 1) uImage (u-boot Image format), 2) binary. In the guest device tree for this partition the *linux,initrd-start* and *linux,initrd-end* properties will be set in the /chosen node to reflect the guest physical start and end addresses of the rootfs image.

Table continues on the next page...

Table 521. Partition Node Properties (continued)

Property Name	Usage	Value Type	Definition
load-image-table	O	<prop-encoded-array>	Specifies the addresses of one or more supplementary images to be loaded by the hypervisor prior to partition start. The value is a prop-encoded array and is a table with one or more rows. Each row consists of a triplet defined as follows: <*image-src-addr image-dest-addr size*> Where: • *image-src-addr* is a <u64> that specifies the source true physical address of an image to be loaded into the partition. • *image-gphys-addr* is a <u64> that specifies the guest physical address within the partition's main memory segment. This must be a word aligned address. For ELF images this value can be -1, which specifies that load address is to be determined from the ELF image itself (see below). • *size* is a <u64> specifies the size of the source address window the image is located in. This value must be nonzero and could be different than the actual source image size. For binary images this field does specify the size of the image. The following images types are supported: 1) ELF (Executable and Linking Format), 2) uImage (u-boot Image format), 3) binary. • For ELF files, if the *image-dest-addr* equals -1 and the ELF header e_type field value is ET_EXEC, each PT_LOAD segment will be loaded at the physical address given by the program header's p_paddr field.
watchdog-timeout	O	<string>	If present this property overrides the default behavior of the TCR[WRC] bits that control whether a partition is reset on a CPU watchdog expiration. Allowable values are: "manager-notify"—Specifies that all manager partitions managing the current partition receive a "watchdog expiration" interrupt. "partition-stop"—Stop the partition with the expired watchdog. "partition-reset"—Restarts the partition with the expired watchdog.
watchdog-autostart-period	O	<u32>	If present, for each CPU belonging to the partition, hypervisor will automatically start the guest watchdog at guest boot, with a period taken from the value of this new property. Early during its boot process, the guest should overwrite the watchdog settings with its own settings before the watchdog expires. If it fails to do so, the watchdog will timeout and cause the hypervisor to execute the watchdog timeout action defined by the "watchdog-timeout" property. The format of the value is identical to the watchdog period set through the TCR register: it specifies one of 64-bit locations of the timebase used to signal a watchdog on a transition from 0 to 1.

Table continues on the next page...

Table 521. Partition Node Properties (continued)

Property Name	Usage	Value Type	Definition
guest-cache-lock-disable	O	<none>	If present defines that guest cache lock mode is disabled.
guest-debug-disable	O	<none>	If present defines that guest debug mode is disabled.
no-auto-start	O	<none>	If present specifies that this partition should not be started. All images specified should be loaded, but the partition should not be started.
mpic-direct-eoi	O	<none>	A Boolean property. If present, specifies that this partition is granted direct end-of-interrupt support. See Direct EOI on page 2174 for more information.
init-map-paddr	O	<u64>	Specifies the guest physical address at which the partition's boot IMA is to be located. This address must be size aligned with the *init-map-size* property. If not specified, the guest physical address of the boot IMA is 0x0. See Initial Mapped Areas on page 2156.
init-map-size	O	<u64>	Specifies the size of the partition's boot IMA in bytes. The size must be a power of 2 and must be a minimum of 4KiB bytes and a maximum of 4GiB. See Initial Mapped Areas on page 2156.
init-map-vaddr	O	<u64>	Specifies the effective address at which the boot IMA should be mapped. This address must be size aligned with the *init-map-size* property. If not specified, the effective address of the boot IMA is 0x0. See Initial Mapped Areas on page 2156.
privileged	O	<none>	A Boolean property that specifies if the partition is privileged. See Privileged Partitions on page 2158 for details on the capabilities of privileged partitions.

Table continues on the next page...

Table 521. Partition Node Properties (continued)

Property Name	Usage	Value Type	Definition
no-dma-disable	O	<none>	Specifying the boolean *no-dma-disable* property has the following effects for all DMA-capable devices assigned to the partition • DMA will automatically be enabled for all devices in the partition on a cold boot—i.e. it is never necessary to explicitly enable DMA for the device when a partition is booted. • DMA for the partition's devices can continue after a partition has been stopped. This option would typically be used in a scenario where an active and standby partition share a device that accesses shared memory. If the active partition crashes and is stopped, by default DMA would be stopped. The no-dma-disable property changes this behavior so DMA can continue. • The FH_PARTITION_DISABLE_DMA hcall will have no effect on any device in the partition
defer-dma-disable	O	<none>	DMA starts disabled on cold boot, and is disabled on a partition reset, but on a partition stop it is not disabled until the FH_PARTITION_DISABLE_DMA hcall requests it.
direct-guest-tlb-management	O	<none>	Allows direct guest TLB management. In this case tlbwe and tlbilx will be executed directly by the guest without hypervisor intervention. This feature is available only on e6500 cores.
direct-guest-tlb-miss	O	<none>	TLB miss interrupts will go directly to the guest.

11.1.2.7.2.2 Guest Physical Memory Areas

Each partition defines a set of guest physical memory areas (GPMAs) that map to PMAs (physical memory areas, see Physical memory areas (PMA) on page 2210). Each GPMA references an associated PMA and specifies the guest physical address to which the PMA should be mapped.

GPMAs are defined as child nodes of a partition node. A GPMA node supports the following properties:

Table 522. GPMA Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "guest-phys-mem-area."
phys-mem	R	<phandle>	Value is a phandle to the PMA (physical memory area) to which this GPMA is to be associated.

Table continues on the next page...

Table 522. GPMA Node Properties (continued)

Property Name	Usage	Value Type	Definition
guest-addr	O	<u64>	The guest physical address at which this GPMA should be mapped. If this property is not present the guest physical address will be identity mapped to the true physical (real) address of the region specified in the PMA. The guest-addr value must be aligned to the size of the associated PMA.
dma-only	O	<none>	If present, this property specifies that the memory region is not accessible by CPUs and is accessible by DMA devices only.

Node updates for GPMAs are supported.

A GPMA appears in the guest device tree as a standard memory node as defined by the ePAPR.

11.1.2.7.2.3 Device nodes

11.1.2.7.2.3.1 Device Nodes Overview

A "device" node is a child node of a partition node and is used to assign physical I/O devices to partitions. There are 2 types of devices represented by device nodes:

- standard devices—a device that does no DMA (e.g. UART)
- DMA devices—a device that is capable of doing DMA (e.g. DMA controller). Every device that performs DMA expresses a logical I/O device number (LIODN) that the platform IOMMU uses to authorize the access. Certain devices express multiple LIODNs and have some special characteristics in the hypervisor configuration tree.

Each type of device has some unique characteristics. Standard devices are the most straightforward and simply have a property that specifies the full path or alias to the node in the hardware device tree that represents the device being assigned to the partition.

The devices nodes appear in the guest device tree under a */devices* node.

DMA devices are represented with the device name (alias or full device tree path) and a reference to a DMA window (see DMA windows on page 2211) that identifies the guest physical address regions to which the device is allowed to DMA.

Qman Portals. Qman portals are unique in that they have multiple LIODNs associated with them that may be individually configured. Each LIODN is represented as a separate child node under the configuration node for the qman portal device and each has its own set of properties including the target DMA window for the logical device. See Device Nodes fsl,qman-portal on page 2225 for complete details.

Node updates for device nodes are supported.

11.1.2.7.2.3.2 Device Configuration Nodes

Device configuration nodes are represented with the following properties:

Table 523. Device Nodes

Property Name	Usage	Value Type	Definition
device	R	<string>	The alias or full path to the device.

Table continues on the next page...

Table 523. Device Nodes (continued)

Property Name	Usage	Value Type	Definition
map-ranges	O	<none>	If present defines that the parent bus address range specified in the *ranges* property should be addressable by guest software in the partition.
vcpu	SD	<u32>	The index of the virtual CPU within the assigned partition to which the device is associated. Currently applicable to nodes with a compatible value of "fsl,qman-portal" and "fsl,bman-portal" only.
claimable	O	<string>	The "claimable" property specifies that this device is being shared with another partition. A device configured like this can be claimed by a partition using the fh_claim_device hcall. Claiming a device causes the device to be active for the claiming partition and interrupts and DMA to be routed to the claiming partition. Allowable values include: "active"—The device is active—device interrupts and DMA errors are routed to the partition. "standby"—The device is 'standby' to this partition and should not be accessed. Interrupts and DMA errors will not be directed to a standby partition. Note: The "claimable" property specified on a device configuration node will apply to all descendant nodes of the assigned device—i.e. in the guest device tree all descendants of the node will be claimable.

In addition, the following properties specify the DMA-related characteristics of a device:

Table 524. Device Nodes, DMA-related properties

Property Name	Usage	Value Type	Definition
dma-window	SD	<phandle>	A phandle to the DMA window node in the hypervisor configuration tree that this device is allowed to target. This is required for DMA capable devices. A dma window specifies a set of guest physical address ranges that are the valid targets of DMA operations. For shared devices using the "claimable" mechanism, the dma-window property must be set on the "active" device only.
operation-mapping	SD	<u32>	Specifies an index in the PAMU operation mapping table. (See Qman portal assignment on page 2226 for a definition of valid values for this property)

Table continues on the next page...

Table 524. Device Nodes, DMA-related properties (continued)

Property Name	Usage	Value Type	Definition
stash-dest	SD	<u32>	Specifies which level in the hierarchy a stashing operation is to be directed. Valid values are as follows: 1: L1 cache 2: L2 cache 3: L3/CPC cache
snoop-cpu-only	SD	<none>	A Boolean property. If present specifies that the snoop traffic associated with the transaction should be restricted to only the core complex (cpu/cluster) the vcpu that this portal is associated with is part of.
liodn-index	SD	<prop-encoded-array>	Applicable to devices that have more than one LIODN. Specifies the index in the liodn property in the hardware device tree node with which this node is associated.
no-dma-disable	O	<none>	Specifying the boolean *no-dma-disable* property has the following effects: • DMA will automatically be enabled for the device on a cold boot—i.e. it is never necessary to explicitly enable DMA for the device when a partition is booted. • DMA for this device can continue after a partition has been stopped. This option would typically be used in a scenario where an active and standby partition share a device that accesses shared memory. If the active partition crashes and is stopped, by default DMA would be stopped. The no-dma-disable property changes this behavior so DMA can continue. • The FH_PARTITION_DISABLE_DMA hcall will have no effect on this device
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.2.3.3 DMA and Shared Devices

Devices can be shared between partitions in a failover configuration, whereby one partition has active ownership of a device at a time (see, High availability services on page 2205).

In this configuration the partitions that share the device each specify a claimable property indicating which partition is the initial active owner and which is standby.

The properties in Device Configuration Nodes on page 2222 that specify the DMA configuration of the device must be placed only in the device node that is the initial "active" device—i.e. standby devices must not specify a DMA configuration. When a shared device is claimed, the only reconfiguration of the PAMU that is performed by the hypervisor is an update of the true physical address of any DMA window that changed for the new active partition.

11.1.2.7.2.3.4 Propagation to Descendants

Certain properties of device nodes are applicable to descendants of the device in the hardware device tree to which the device node refers.

- **dma-window**. The *dma-window property* applies to all descendants of the device in the hardware device tree.

This behavior can be overridden for descendants of the base device by specifying a separate device node for the child node. See the Embedded Hypervisor User's Manual for an example.

11.1.2.7.2.3.5 simple-bus

For device nodes where all ancestors contain a compatible value of "simple-bus", and where no ancestor has been assigned to the partition, the hypervisor copies the node from the hardware device tree node to the root of the guest tree. Any *reg* and *ranges* properties which may have contained an offset, are updated to reflect the full physical address of the devices registers as mapped by any *ranges* properties in the ancestry of the node.

11.1.2.7.2.3.6 Device Nodes fsl,qman-portal

See Message Signaled Interrupts on page 2225 for a description of how Qman portals are assigned to a partition.

11.1.2.7.2.3.7 fsl,dpaa

For device nodes with a parent that contains a compatible value of "fsl,dpaa", both the node and parent are copied to the guest device tree.

11.1.2.7.2.3.8 fsl,p4080-pcie, fsl,p3041-pcie, fsl,p5020-pcie

11.1.2.7.2.3.8.1 Message Signaled Interrupts

PCI Express message signaled interrupts require special consideration.

1. Assignment of PCI and MSI nodes

   In the hardware device tree each PCI controller node references an associated MSI node which represents the MSI interrupt register bank in the MPIC. The fsl,msi property on the PCI controller node in the hardware device tree contains a phandle pointing to the MSI node. When assigning a PCI controller to a partition the corresponding MSI node must also be assigned.

   Each MSI bank supports 8 physical MPIC interrupts. Each of the 8 interrupts support 32 MSIs. When an MSI occurs, the guest software must read the interrupt handle via IACK or EPR and then invokes the **fh_vmpic_get_msir** hypercall to read the MSIRx (Shared message signaled interrupt register) which specifies which of the 32 interrupts occurred.

2. MSI Addresses

   PCI Express endpoints trigger message signaled interrupts by writing to the MPIC MSIIRx *(Shared message signaled interrupt index register)*. This 'MSI address' is programmed into the endpoints by the host PCI controller. There are two considerations related to the MSI address:

   PAMU configuration

   The endpoint access to write the MSI address is a normal memory write and thus must be authorized by PAMU. This requires that a special MSI subwindow be configured in the DMA setup for the PCI controller (see the *pcie-msi-subwindow* property) which specifies the guest physical address of the page where the MSIIRx register is located.

   The MSI address goes through a translation from PCI address space to a system address via the ATMUs in the PCI controller. Care must be taken to configure the inbound ATMU so the resulting system address hits in the MSI subwindow.

   msi-address-64

   The *msi-address-64* property will be set on the PCI controller node in the guest device tree and will contain the address in PCI address space to which endpoints should write to trigger MSIs.

3. MSI nodes

   MSI nodes contain the array of interrupt specifiers for each of the 8 MSI interrupts. See Interrupt Controller Services on page 2171 for additional detail on processing MSIs.

11.1.2.7.2.4 Qman portal assignment

Qman portals are represented in the hardware device tree with a node containing a compatible value of "fsl,qman-portal." Qman portals express six different LIODNs:

- Fman0
- Fman1
- SEC
- PME
- Dequeue data stashing
- Dequeue DQRR stashing

Qman portals are typically associated with physical CPUs, and thus a normal configuration would assign one Qman portal to a partition per physical CPU assigned.

There are 2 components involved in assigning Qman portal devices:

- Definition of a portal-devices node, that is common to all Qman portals in the partition
- A device assignment node for each Qman portal assigned to the partition

11.1.2.7.2.4.1 portal-devices

The Fman0, Fman1, SEC, and PME portal devices will share a common configuration for all portals assigned to a partition. This configuration is specified under a special portal-devices device configuration node.

The portal-devices node must be a child of the partition node and have the name "*portal-devices*".

Under the *portal-devices* node Fman0, Fman1, SEC, and PME must each define a subnode with the device configuration node as described in Device Configuration Nodes on page 2222:

- Fman0 and Fman1 must have a device property specifying the device tree path (or alias) to the respective hardware fman node and a *dma-window* property
- SEC subnodes must have a device property specifying the device tree path (or alias) to the hardware SEC node and a *dma-window* property specified
- PME subnodes must have a device property specifying the device tree path (or alias) to the hardware pme node and a *dma-window* property specified

See the NXP Embedded Hypervisor User's Manual for examples.

11.1.2.7.2.4.2 fsl,qman-portal

Qman portals are assigned to a partition like other hardware I/O devices using device configuration nodes as described in Device Configuration Nodes on page 2222. However, in addition under the partition configuration node Qman portals use two child nodes to represent the two stashing LIODNs expressed by the portal:

- Dequeue data stashing
- Dequeue DQRR stashing

The primary portal device configuration node may have the following 2 properties

- *device*—the value specifies the full path to the qman portal
- *vcpu*—the value specifies the virtual or logical cpu index with which the portal is to be associated. This property is optional and should be specified if a binding between portal and cpu is desired. If the vcpu property is present the *cpu-handle* property in the hardware device tree node will be updated with a phandle to the physical cpu node.

The subnodes use the properties described in Device Configuration Nodes on page 2222, but have the following additional requirements:

- Dequeue DQRR stashing subnode

- must have a *dma-window* property specified
- must have an *liodn-index* property with a value of 0
- must have an *operation-mapping* property set with the index of the Qman stashing operation mapping entry
- must have a *stash-dest* property which specifies the cache the stash is targeting
- can optionally have a snoop-cpu-only property specified

- Dequeue data stashing subnode
  - must have a *dma-window* property specified
  - must have an *liodn-index* property with a value of 1
  - must have an *operation-mapping* property set with the index of the Qman stashing operation mapping entry
  - must have a *stash-dest* property which specifies the cache the stash is targeting

See the NXP Embedded Hypervisor User's Manual for examples.

11.1.2.7.2.5 Operation Mapping Table Index

The table below defines the operation mapping table indexes that may be used as values in the *operation-mapping* property on device configuration nodes (see Device Configuration Nodes on page 2222).

Table 525. Operation Mapping Table Index Definition

Index	Device	Ingress Operation	Egress Operation
0	Qman	READ	READ
		EREAD0 (Enhanced read type 0)	RSA (Read with stash allocate)
		WRITE	WRITE
		EWRITE0 (Enhanced write type 0)	WWSAO (Write with stash allocate only)
		DIRECT0 (Directive type 0)	LDEC (Load external cache)
		DIRECT1 (Directive type 1)	LDECPE (Load external cache with preferred exclusive)
1	Fman	READ	READI (Read with invalidate)
		WRITE	WRITE
2	Qman Private	READ	READ
		WRITE	WRITE
		EREAD0 (Enhanced read type 0)	RSA (Read with stash allocate)
		EWRITE0 (Enhanced write type 0)	WWSA (Write with stash allocate)
3	SEC	READ	READI (Read with invalidate)
		WRITE	WRITE

11.1.2.7.2.6 Byte-Channel Configuration

Byte-channel endpoints are defined as nodes under the partition node. Each byte-channel node may have the following properties:

Table 526. Byte-Channel Node Properties (Hypervisor Configuration)

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "byte-channel."
endpoint	R	<phandle>	A phandle to the endpoint to which the byte channel is connected. Valid values are as follows: • A device node pointing to a physical UART • a byte-channel mux • another byte-channel node
mux-channel	SD	<u32>	Specifies the mux channel number. Required if the endpoint property points to a byte-channel mux
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

Node updates for byte-channel nodes are supported.

11.1.2.7.2.7 Inter-partition Doorbell Configuration

Doorbell send and receive endpoints are declared as nodes under the partition node. These nodes reference the global doorbell node with which they are associated.

Supported properties are shown in the table below:

Table 527. Doorbell Configuration Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	For receive doorbell endpoints must include "receive-doorbell." For send doorbell endpoints, must include "send-doorbell"
global-doorbell	R	<phandle>	Value is a phandle to the global doorbell node to which this endpoint is associated.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

Node updates for doorbell nodes are supported.

11.1.2.7.2.8 Partition Management Nodes

If the partition being defined via the partition node manages other partitions, this relationship is expressed by declaring a node that references the managed partition.

It is supported to have partitions that "manage" each other. This can be used to create high availability scenarios where partitions are notified about the state of a partition that they manage/monitor.

Supported properties are shown in the table below:

Table 528. Partition Management Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "managed-partition."
partition	R	<phandle>	Value is a phandle to the partition node which is to be managed by this partition. This property results in a partition management handle node to be placed under the manager partitions /hypervisor/handles node in its guest device tree. In addition, this property results in a set of partition management doorbells to be placed under the manager partition's /hypervisor/handles node in its guest device tree.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.2.9 Error Manager

A partition is designated to be an "error manager" by defining an error manager node as a child of the partition node. See Error Management on page 2188, for details on error management.

Table 529. Error Manager Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<stringlist>	Must include "error-manager". Specifies that this partition is an error manager.
vcpu	R	<u32>	Specifies the vcpu number within the partition that is to receive interrupts for global errors
claimable	O	<string>	The "claimable" property specifies that error management is being shared with another partition in a failover configuration. Error management responsibilities can be claimed by a partition using the fh_claim_device hcall. Valid values are: "active" - The partition is the active error manager "standby" - The partition is the standby error manager
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

Node updates for the error manager node is supported.

11.1.2.7.2.10 Guest Aliases

By default all assigned devices that have an alias in the hardware device tree will have that alias copied to the guest device tree.

In addition, a guest aliases node can be specified under a partition node to add additional aliases to the guest device tree. The name of this node must be aliases. The value must be a phandle to a node in the hypervisor configuration tree. The phandle values are resolved to a string value that is the full path to the referenced object in the guest device tree.

11.1.2.7.2.11 Guest Root Node Properties

The partition node supports *a node-update* node which updates the root node of the guest device tree.

11.1.2.7.2.12 Debug Stub Node

The hypervisor supports debug agents or stubs (built into the hypervisor) which may be use to debug guest software running in a partition. A byte-channel provides a communications channel from the stub to a host debugger.

To instantiate a debug stub for a partition a *debug-stub* node must be present as a child of the partition node (see Partition Node Properties on page 2215) which defines the properties of the stub.

Note: all debug stub nodes within a partition must be of the same type—i.e. the *compatible* string must be the same.

Debug stubs require the CPU's debug resources (debug registers and debug interrupt) for normal operation. This means that guest debug mode must be turned off, granting the debug resources to the hypervisor. See the *guest-debug-disable* property for partition nodes (section Partition Node Properties on page 2215).

Table 530. Debug Stub Node Properties

Property Name	Usage	Value Type	Definition
compatible	R	<string>	For GDB must include "gdb-stub," "debug-stub" For TRK must include "trk-stub", "debug-stub"
endpoint	R	<phandle>	Specifies the phandle of the byte-channel endpoint to which this stub is to be attached. Valid values are as follows: • A device node pointing to a physical UART • a byte-channel mux • another byte-channel node
mux-channel	SD	<u32>	Specifies the mux channel number. Required if the endpoint property points to a byte-channel mux
debug-cpus	O	<prop-encoded-array>	Array defining which virtual CPUs are to be debugged by the debug stub. This value is a <prop-encoded-array> encoded as an arbitrary number of pairs *(cpu-index, count)*. *cpu-index* is a <u32> that specifies a cpu index from the guest device tree (the index is identical to the value of the "reg" property in cpu nodes) *count* is a <u32> that specifies the number of CPUs being enumerated. If this property is not present the hypervisor will assume that all CPUs in the partition are to be debugged by this stub.
gdb-wait-at-start	O	<none>	Applicable to gdb stub nodes only (compatible = "gdb-stub"). If present specifies that the gdb stub should halt the guest OS prior to execution of the first instruction of the guest (its entry point) and wait for the host debugger to attach.

NOTE

There is an experimental hypervisor configuration option that enables debugging using the program interrupt instead using the CPU debug resources. This option may be used if there is a requirement to use a debug stub while enabling guest debug mode for a partition. See CONFIG_DEBUG_STUB_PROGRAM_INTERRUPT

11.1.2.7.3 Structure of Guest Device Trees

Partitions are passed an ePAPR compliant guest device tree. A guest device tree describes all the resources available to the partition. This includes nodes describing hardware devices, as well as nodes describing virtual devices.

Root node properties. Guest device trees will have a root node with the following standard ePAPR properties copied intact from the hardware device tree

- *epapr-version*
- *#address-cells*
- *#size-cells*

In addition, the *compatible* property on the root node will have the string "simple-bus" added to the stringlist if it is not already present.

The root node can be updated with a *node-update* property.

I/O Devices. Devices assigned to a partition by a device node will be placed at the root of the guest device tree.

Hypervisor node. Each guest will have a hypervisor node at the root of the guest device tree. The hypervisor node contains properties and handles to be used by a guest.

11.1.2.7.4 Hypervisor configuration node

The hypervisor is also configured via the hypervisor configuration tree. A special node and subtree defines set of global configuration parameters and a number of device nodes assigning I/O devices to the hypervisor:

Configuration parameters specify the hypervisor's console, the hypervisor's private memory, and interrupt configuration. The parameters are described in the following table.

Table 531. Hypervisor configuration parameters

Property Name	Usage	Value Type	Definition
compatible	R	<string>	Must include "hv-config."
stdout	O	<phandle>	phandle to a serial device or byte-channel which should be used for the hypervisor's console. This property takes precedence over any *stdout* alias set in the hardware device tree.
sysreset-on-partition-stop	O	<none>	If present this property specifies that if all partitions move into the "stopped" state that the system should be reset.

Table continues on the next page...

Table 531. Hypervisor configuration parameters (continued)

Property Name	Usage	Value Type	Definition
watchdog-enable	O	<u32>	If present defines specifies that the hypervisor watchdog is to be enabled (on all CPUs). The value specifies the watchdog timer period. The value specifies one of 64-bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. 0 - selects TBU[32], the msb of the timebase 63 - selects TBL[63], the lsb of the timebase
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.4.1 Hypervisor Private Memory

The hypervisor requires a region of physical memory for its private use. A child node of the hypervisor configuration node defines this region (similar to a guest physical memory area):

Table 532. Hypervisor Subnode Properties

Property Name	Usage	Value Type	Definition
compatible	R	<string>	Must include "hv-memory."
phys-mem	R	<phandle>	Value is a phandle to the PMA (physical memory area) which is assigned to the hypervisor..
hv-persistent-data	O	<prop-encoded-array>	If present, it defines a memory region within the hypervisor's memory where PAMU-related data structures are located. The value specifies the offset and size within the memory area. The region include a 4KiB region at the beginning reserved for hypervisor use (for PAMU-related hypervisor housekeeping). The property value is a prop-encoded-array consisting of 4 cells defined as: <offset-hi offset-lo size-hi size-lo> offset-hi | offset-lo - a pair of <u32> specifying the offset in the memory area where the hypervisor persistent data begins size-hi | size-lo - a pair of <u32> cells specifying the size of the memory area The size of the memory region is SoC dependent. Contact NXP support for recommendation for the value of the size.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.4.2 Hypervisor Owned I/O Devices

A number of SOC devices must be assigned to the hypervisor. These are assigned using device assignment nodes which are children of the *hypervisor configuration node*. The device assignment nodes definition use a syntax identical to that of assigning devices to partitions (see Device nodes on page 2222).

For hypervisor owned serial ports the baud rate is configured as follows—if the serial port device node in the hardware device tree passed to the hypervisor has a *current-speed* property, that speed will be used. Otherwise, if the device assignment node for the serial device has a *baud* property (see table below) the value of that property is used. If neither property is present a baud rate of 115200 is used.

Table 533. *baud* property

Property Name	Usage	Value Type	Definition
baud	O	<u32>	Specifies the baud rate to be used for the hypervisor console when it is attached to a serial port (see Hypervisor Console Configuration on page 2234). This property is valid on a device assignment node for a serial device that is a child of the hypervisor configuration node. This property takes precedence over any current-speed property set in the hardware device tree.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

For all serial configurations the following serial port parameters are assumed:

- data bits: 8
- parity: none
- stop bits: 1

NOTE

For the QorIQ SoCs the following devices must be assigned to the hypervisor for proper operation:

- Interrupt controller (compatible "chrp,open-pic")
- Corenet local access windows (compatible "fsl,corenet-law")
- Corenet coherency fabric (compatible "fsl,corenet-cf")
- Memory controllers (e.g. compatible ""fsl,p4080-memory-controller")
- Platform cache controllers (e.g. compatible "fsl,p4080-l3-cache-controller")
- IOMMU (e.g. compatible "fsl,p4080-pamu")
- Global utilities (e.g. compatible "fsl,qoriq-device-config-1.0")

11.1.2.7.4.3 Error Policy Configuration

For hypervisor owned devices (see Hypervisor Owned I/O Devices on page 2232), an error handling policy may be configured, which would override the default policy for the error. Default policies are defined in , . An error configuration node is used to define error policies.

Table 534. Error Policy Properties

Property Name	Usage	Value Type	Definition
compatible	R	<string>	Must include "error-config."
Table continues on the next page...			

Table 534. Error Policy Properties (continued)

Property Name	Usage	Value Type	Definition
domain	R	<string>	Specifies the domain of the error. See Error Domains, Error Types, Error Policies on page 2190 for a list of error domain strings.
error	R	<string>	Specifies the error type the policy applies to. See Error Domains, Error Types, Error Policies on page 2190 for a list of error type strings.
policy	R	<string>	Specifies the policy for the error. Valid policies are: "disable" - error is ignored/masked "notify" - notification to global event queue "halt" - hypervisor halts "system-reset" - hardware system reset
single-bit-ecc-threshold	O	<u32>	A value from 0 to 256 that specifies how many single bit ECC errors are to be tolerated before an error is asserted by the hardware.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition			

11.1.2.7.4.4 Hypervisor Owned Byte-channels

A byte-channel can be defined to be the hypervisor's console. A node defining a hypervisor-owned byte-channel is a child of the hypervisor configuration node and the definition is identical to that of byte-channels under partition nodes (see Byte-Channel Configuration on page 2228).

11.1.2.7.4.5 Hypervisor Console Configuration

The hypervisor provides support for a console that can be used for debugging. It provides informational and error messages and an interactive shell that can be used to display hypervisor status and state.

The console can be attached to either a UART or byte-channel.

If the *stdout* alias is specified in the hardware device tree, the serial device referenced will be used as the console, unless overridden by the *stdout* property on the hypervisor configuration node specifies the console device (see Hypervisor configuration node on page 2231).

The console device must be A) assigned to the hypervisor (UART), or B) defined (byte-channel) in the hypervisor configuration node.

11.1.2.8 Debugging

11.1.2.8.1 Hypervisor Console and Shell

The hypervisor has a console which is may be attached to a uart or byte-channel. The console is used by the hypervisor to print error and warning messages. In addition, through a compile time configuration option, a debug command prompt can be enabled that supports some commands that provide greater visibility into the partition and hypervisor state.

The console to byte-channel connection definition is specified in the stdout property in the hypervisor configuration node (see Hypervisor configuration node on page 2231).

Table 535. Supported Shell Commands

Command	Description
buildconfig	Display build time configuration options
cdt	Display hypervisor configuration tree
error_policy	Display error policies
gdt print <partition#>	Displays the guest device tree. Partition number can be displayed by the **info** command.
gtlb <partition #> <vcpu#>	Displays the guest TLB for the specified partition and virtual CPU.
guestmem <partition-#> <address> [<length>]	Displays guest memory by guest physical address
help [command]	If the *command* argument is not specified, lists all the shell commands with a short help message.
hdt	Displays the hardware device tree.
info	Lists the partitions that exist and their current status. The partition numbers displayed with this command are used in other shell commands.
paact	Displays the PAACT table for the PAMU.
pause	Pauses a running partition. In this state the partition halts execution and all interrupts are disabled.
reset	Resets the system-asserts the reset request signal.
restart <partition #>	Restarts a running partition. This includes loading hypervisor-loaded images. NOTE: **restart** honors the optional no-auto-start property that can be present on partition nodes. If no-auto-start is present, images are loaded but the partition is not started.
resume	Resumes a paused partition.
start [load] <partition #>	Starts a stopped partition. If the load argument is specified any hypervisor loaded images specified in the hypervisor configuration tree are loaded as well. NOTE: **start load** <partition #> honors the optional no-auto-start property that can be present on partition nodes. The effect of **start load** with no-auto-start is that images are loaded but the partition is not started.
stop	Stops a running partition.
version	Displays the version of the hypervisor.

Table continues on the next page...

Table 535. Supported Shell Commands (continued)

Command	Description
loglevel [group] [level]	Changes the current log-level of the specified logging group. Use the command with no arguments to display the groups and their current log levels. Log level 0 disables logging and log level 15 is the most verbose. Setting higher levels may be dangerous (e.g enabling debug on a byte-channel mux (bcmux) when the hypervisor console is on the mux).

NOTE

The shell command syntax is preliminary and subject to change in future releases of the hypervisor.

11.1.2.8.2 GDB Debug Stub

11.1.2.8.2.1 GDB Debug Stub Overview

The hypervisor can act as a remote target for the GNU Debugger (GDB). This provides the capability to debug guest software running in a partition.

The hypervisor implements support for the *gdb Remote Serial Protocol* as described in version 6.7 of the gnu Debugger (GDB) reference manual. (See Appendix D of *Debugging with gdb, The gnu Source-Level Debugger*.

The hypervisor also supports the target description format as defined in *Target Descriptions*, Appendix F of the GDB 6.7 reference manual.

The gdb stub supports limited debugging capabilities for SMP guests. One stub per CPU must be defined and a separate GDB host debugger per stub must be used. Hardware breakpoints must be used for setting breakpoints for SMP guests. See Debug Stub Node on page 2230, for details on configuring stubs in the hypervisor configuration tree.

11.1.2.8.2.2 Supported GDB Commands

The hypervisor gdb stub supports the following commands designated as required by the GDB manual:

- 'g' (Read register)
- 'D' (detach)
- 'G' (write register)
- 'm' (read memory)
- 'M' (write memory)
- 'c' (continue)
- 's' (Single step)

The hypervisor gdb stub also support the following commands:

- '?' (indicates reason target halted)
- z0 and Z0 (set/remove memory breakpoints)
- z1 and Z1 (set/remove hardware breakpoints)

The stub supports two hardware breakpoints.

- z2 and Z2 (set/remove write watchpoint)
- z4 and Z4 (set/remove access watchpoint)

The general query packet 'qXfer:features:read' which provides access to the target description XML file

- p (read_register)
- P (write_register)
- qOffsets (get section offsets)
- qSupported (supported_features)
- D (detach)
- X (raw_write)
- qRcmd (monitor)

supports a "restart" command to restart a partition

- k (kill)

11.1.2.9 Byte-Channel Serial Multiplexer Protocol

11.1.2.9.1 Byte-Channel Serial Multiplexer Protocol Overview

The hypervisor implements a byte-channel multiplexer which can multiplex up to 32 byte-channels over a serial RS232 link.

The multiplexer implements an escape-sequence based channel switching protocol. An escape sequence sets the 'current channel' and all data that follows belongs to that channel until a new current channel is set which changes the channel.

The byte value 0x18 (CTRL-X character) specifies the start of an escape sequence. The byte following the 'escape sequence start' is an escape sequence **command**.

Up to 32 channels are supported.

11.1.2.9.2 Escape Sequence Commands

Table 536. Escape Sequence Commands

Escape Sequence and Command	Meaning
0x18 0x00	Reserved
0x18 0x01	TX reset -- tells the remote end to retransmit the tx channel switch the next time it transmits even if on the same channel as before
0x18 0x02-0x18 0x2F	Reserved
0x18 0x30	Set current channel to 0
0x18 0x31	Set current channel to 1
0x18 0x32	Set current channel to 2
0x18 0x33	Set current channel to 3
0x18 0x34-0x18 0x4F	Set current channel to 4-31
0x18 0x50-0x18 0x7F	Reserved
0x18 0x18	Send the character 0x18 through as data

11.1.2.10 Linux hypervisor management driver

11.1.2.10.1 Linux Hypervisor Management Driver Overview

A Linux character device driver allows user space access to certain hypervisor services for partition management. This section documents the ioctl API provided by this device driver.

All the IOCTLs are defined in the Linux header file: fsl_hypervisor.h

11.1.2.10.2 FSL_HV_IOCTL_PARTITION_RESTART

IOCTL code: FSL_HV_IOCTL_PARTITION_RESTART

Description: Restart a running partition

FSL_HV_IOCTL_PARTITION_RESTART takes a **fsl_hv_ioctl_restart** structure as an argument.

Table 537. FSL_HV_IOCTL_PARTITION_RESTART fields

Parameter	Description
__u32 partition	An input parameter. Partition ID to restart, or -1 for the calling partition
__u32 ret	An output parameter. Specifies the return value. Values are described below: 0 : success EINVAL : Invalid Partition ID FH_ERR_INVALID_STATE : Partition state is invalid (not running)

11.1.2.10.3 FSL_HV_IOCTL_PARTITION_GET_STATUS

IOCTL code: FSL_HV_IOCTL_PARTITION_GET_STATUS

Description: Query the status of a partition

FSL_HV_IOCTL_PARTITION_GET_STATUS takes a **fsl_hv_ioctl_status** structure as an argument.

Table 538. FSL_HV_IOCTL_PARTITION_GET_STATUS fields

Parameter	Description
__u32 partition	An input parameter. Partition ID of the partition to query, or -1 for the calling partition
__u32 ret	An output parameter. Specifies the return value. Values are described below : 0 : success EINVAL : Invalid Partition ID
__u32 status	An output parameter. Returns the status of the partition. Returned Values for "status" : 0 = Stopped 1 = Running 2 = Starting 3 = Stopping

11.1.2.10.4 FSL_HV_IOCTL_PARTITION_START

IOCTL code: FSL_HV_IOCTL_PARTITION_START

Description: Start a (stopped) partition.

FSL_HV_IOCTL_PARTITION_START takes a fsl_hv_ioctl_start structure as an argument.

Table 539. FSL_HV_IOCTL_PARTITION_START fields

Parameter	Description
__u32 partition	An input parameter. Partition ID of partition to start
__u32 entry_point	An input parameter. The offset within the guest IMA to start execution
__u32 load	An input parameter. Flag identifying whether to load images or not.
__u32 ret	An output parameter. Specifies the return value. Values are described below: 0 : Success EINVAL : Invalid Partition ID EV_INVALID_STATE : Partition to start is not in stopped state.

11.1.2.10.5 FSL_HV_IOCTL_PARTITION_STOP

IOCTL code: FSL_HV_IOCTL_PARTITION_STOP

Description: Stop a running partition.

FSL_HV_IOCTL_PARTITION_STOP takes a fsl_hv_ioctl_stop structure as an argument.

Table 540. FSL_HV_IOCTL_PARTITION_STOP fields

Parameter	Description
__u32 partition	An input parameter. Partition ID of partition to start, or -1 for the calling partition.
__u32 ret	An output parameter. Specifies the return value. Values are described below: 0 : Success EINVAL : Invalid Partition ID FH_ERR_INVALID_STATE : Partition is not running or it is paused

11.1.2.10.6 FSL_HV_IOCTL_MEMCPY

IOCTL code: FSL_HV_IOCTL_PARTITION_MEMCPY

Description: Copy memory between partitions. Data is copied from the "source" partition to the "target" partition. Memory copying between two remote partitions or within the same partition is not supported.

FSL_HV_IOCTL_PARTITION_MEMCPY takes a fsl_hv_ioctl_memcpy structure as an argument.

Table 541. FSL_HV_IOCTL_PARTITION_MEMCPY fields

Parameter	Description
__u32 source	An input parameter. Partition ID of source partition, or -1 for the calling partition
__u32 target	An input parameter. Partition ID of target partition, or -1 for the calling partition
__u64 local_vaddr	An input parameter. User-space virtual address of a buffer in the "source" partition
__u64 remote_paddr	An output parameter. Guest physical address of the a buffer in the "target" partition. This buffer must be guest physically contiguous.
__u64 count	An input parameter. Number of bytes to copy. Both the source and target buffers must be at-least "count" bytes long.
__u32 ret	An output parameter. Specifies the return value. Values are described below: 0 : Success EINVAL : Source or Target Partition ID is invalid. EFAULT : Unable to map source or target buffer

IOCTL Return code:

Table 542. FSL_HV_IOCTL_PARTITION_MEMCPY return codes

Return value	Description
EINVAL	Memory copying between two remote partitions or within the same partition is not supported.
ENOMEM	Internal memory allocation failures
EACCES	Unable to lock source buffer

11.1.2.10.7 FSL_HV_IOCTL_DOORBELL

IOCTL code: FSL_HV_IOCTL_DOORBELL

Description: Ring a doorbell.

FSL_HV_IOCTL_DOORBELL takes a **fsl_hv_ioctl_doorbell** structure as an argument.

Table 543. FSL_HV_IOCTL_DOORBELLfields

Parameter	Description
__u32 doorbell	An input parameter. Handle of the doorbell to ring doorbell
__u32 ret	An output parameter. Specifies the return value. Values are described below: 0 : Success EINVAL : Invalid handle

11.1.2.10.8 FSL_HV_IOCTL_GETPROP

IOCTL code: FSL_HV_IOCTL_GETPROP

Description: Get a device tree property from another partition

FSL_HV_IOCTL_GETPROP takes a fsl_hv_ioctl_prop structure as an argument.

Table 544. FSL_HV_IOCTL_GETPROP fields

Parameter	Description
__u32 handle	An input parameter. Handle of partition whose tree to access
__u64 path	An input parameter. Virtual address of path name of node to access
__u64 propname	An input parameter. Virtual address of name of property to access
__u64 propval	An output parameter. Virtual address of property data buffer. The property value is returned in this parameter.
__u32 proplen	An input parameter. Size of property data buffer
__u32 ret	An output parameter. Specifies the return value. Values are described below: 0 : Success EINVAL : Invalid Partition handle ENOMEM : Internal memory allocation failures ERR_INVALID : Partition in boot stage EFAULT : Unable to access path or property name ENOENT : Invalid path or Property not found in partition device tree EV_BUFFER_OVERFLOW : Caller supplied buffer (propval) is too small

IOCTL Return code:

Table 545. FSL_HV_IOCTL_GETPROP return codes

Return value	Description
EINVAL	Property length larger than max. supported property length (32K)
ENOMEM	Internal memory allocation failures
ENAMETOOLONG	Path name or Property name too long

11.1.2.10.9 FSL_HV_IOCTL_SETPROP

IOCTL code: FSL_HV_IOCTL_SETPROP

Description: Set a device tree property in another partition

FSL_HV_IOCTL_SETPROP takes a fsl_hv_ioctl_prop structure as an argument.

Table 546. FSL_HV_IOCTL_SETPROP fields

Parameter	Description
__u32 handle	An input parameter. Handle of partition whose tree to access
__u64 path	An input parameter. Virtual address of path name of node to access
__u64 propname	An input parameter. Virtual address of name of property to access
__u64 propval	An input parameter. Virtual address of property data buffer
__u32 proplen	An input parameter. Size of property data buffer
__u32 ret	An output parameter. Specifies the return value. Values are described below: 0 : Success EINVAL : Invalid Partition handle EFAULT : Unable to access path or property name ENOMEM : Invalid path or Property could not be set ERR_INVALID : Partition in boot stage

IOCTL Return code:

Table 547. FSL_HV_IOCTL_SETPROP return codes

Return value	Description
EINVAL	Property length larger than max. supported property length (32K)
ENOMEM	Internal memory allocation failures
ENAMETOOLONG	Path name or Property name too long

Table 548. IOCTL Code Encoding

IOCTL	Number Encoding
FSL_HV_IOCTL_PARTITION_RESTART	1
FSL_HV_IOCTL_PARTITION_GET_STATUS	2
FSL_HV_IOCTL_PARTITION_START	3
FSL_HV_IOCTL_PARTITION_STOP	4
FSL_HV_IOCTL_MEMCPY	5
FSL_HV_IOCTL_DOORBELL	6

Table continues on the next page...

Table 548. IOCTL Code Encoding (continued)

IOCTL	Number Encoding
FSL_HV_IOCTL_GETPROP	7
FSL_HV_IOCTL_SETPROP	8

11.1.2.11 Hypervisor Debug Stubs

11.1.2.11.1 Hypervisor debug stubs overview

This chapter specifies the interface for implementing a debug stub in the NXP Embedded Hypervisor.

In order to integrate into the hypervisor a debug stub must implement an initialization interface and register several callbacks.

The hypervisor provides API that enable a stub to do the following:

- Access guest memory by effective address
- Access guest memory by guest physical address
- Access guest registers for a virtual CPU (vcpu)—GPRs, FPRs, SPRs, PMRs, MSR, CR, PC

A NXP Embedded Hypervisor byte-channel must be configured in the guest device tree to provide and enable communications with a host debugger.

When a partition is configured to support debugging via a debug stub, the stub has full access to the physical CPU debug resources and guest is not permitted access to the debug resources.

11.1.2.11.2 Hypervisor-to-Stub Interfaces

11.1.2.11.2.1 Stub Per-CPU Data

The hypervisor maintains per-partition and per-virtual-CPU data structures that a debug stub may need to access. The data structures are declared in **include/percpu.h**.

- Per partition data:

```
typedef struct guest_t;
```

- Per virtual CPU data:

```
typedef struct gcpu;
```

A stub can get direct access to the virtual CPU data using the get_gcpu() function:

```
gcpu_t *gcpu = get_gcpu();
```

Within the gcpu_t struct a pointer field exists for use by a stub to maintain per virtual CPU private data.

```
void *dbgstub_cpu_data;
```

Within the gcpu_t struct a pointer exists to the debug stub's device tree configuration node:

```
dt_node_t *dbgstub_cfg;
```

Access to **per-partition** hypervisor data is accessible via a pointer in the `gcpu_t` structure.

```
guest_t *guest;
```

11.1.2.11.3 Hypervisor Registration of Debug Stubs

The hypervisor interfaces to a debug stub through a set of callbacks that a stub statically defines in a source file:

```
#include "stubops.h"

typedef struct {

        const char *compatible;

        void (*vcpu_init)(void);

        void (*vcpu_start)( trapframe_t *trapframe);

        void (*vcpu_stop)(void);

        int (*debug_interrupt)(trapframe_t *trapframe);

} stub_ops_t;} stub_ops_t;
```

The attr_debug_stub macro must be used when declaring the structure which will place the declaration in a special section by the linker so the hypervisor can find it.

Example:

```
#include "stubops.h"

static stub_ops_t attr_debug_stub stub_ops = {

        .compatible = "gdb-stub",

        .vcpu_init = gdb_stub_init,

        .vcpu_start = gdb_stub_start,

        .vcpu_stop = gdb_stub_stop,

        .debug_interrupt =  gdb_stub_process_trap,

};
```

The "compatible" field must be identical to the device tree compatible field in the node that declares the stub in the partition node of the hypervisor configuration device tree (see Debug Stub Node on page 2230 for additional details).

11.1.2.11.3.1 Callbacks

11.1.2.11.3.1.1 Call back context

All callbacks should only change hypervisor state through the APIs provided.

11.1.2.11.3.1.2 vcpu_init callback

Function prototype: void (*vcpu_init)(void);

The vcpu_init() function is called once for each virtual CPU during partition initialization..

The function performs tasks such as the following:

- initializing the stub byte-channel (see Byte-channel Registration on page 2257).
- configuration debug control registers (e.g. DBCR0[IDM])
- enable debug interrupts via MSR[DE]
- allocation of per-vcpu memory and setting the gcpu->dbgstub_cpu_data pointer

A pointer to the stub's device tree configuration node is available in gcpu->dbgstub_cfg.

11.1.2.11.3.1.3 vcpu_start callback

Function prototype: void (*vcpu_start)(trapframe_t *trapframe);

The vcpu_start() function is called every time a virtual CPU is started (including after partition reboots).

The function performs tasks such as the following:

- Register handlers for any byte-channels used.

Note: the registered byte-channel handlers will run on the physical CPU that manages the physical UART interrupt (e.g. CPU 0). The byte-channel handler should use the hypervisors internal event mechanism to signal events to other physical CPUs (see Hypervisor gevents on page 2256).

11.1.2.11.3.1.4 vcpu_stop callback

Function prototype: void (*vcpu_stop)(void);

The vcpu_stop() function is called for each virtual CPU in a partition when the CPU being stopped. The function should undo operations performed in vcpu_start()-e.g. deregister byte-channel callbacks.

11.1.2.11.3.1.5 debug_interrupt callback

Function prototype: int (*debug_interrupt)(trapframe_t *trapframe);

Stubs must implement a call back to be invoked for handling debug interrupts.

Return Value

The debug_interrupt() callback must return the value 0 if the callback handled the interrupt. If the reason for the interrupt could not be determined (i.e. the debug event was not targeted to this stub) the call be must return the value 1.

11.1.2.11.3.2 Build Time Configuration

Stubs need to be added to the hypervisor's build system via the Kconfig and a Makefile.

In the `Kconfig` file in the hypervisor source create a new config entry (see the existing GDB stub as an example).

In the file `Makefile.build` conditionally add the stub source file(s).

Example:

```
hv-src-$(CONFIG_GDB_STUB) += gdb-stub.c
```

11.1.2.12 Hypervisor APIs

This section describes the hypervisor APIs available to debug stubs and other custom code .

11.1.2.12.1 Trapframe

Many APIs require a trapframe_t pointer passed to them. A trapframe contains the current context of the guest. For debug stubs, the stub will receive a pointer to this structure when the stub's callback functions are invoked.

NOTE

The trapframe should be considered an opaque data structure and should not be accessed directly by stubs.

11.1.2.12.2 SPR access

For direct access to SPRs (e.g. debug registers) for internal use by the debug stub the mtspr() and mfspr() routines should be used.

Header file:

libos/io.h

Table 549. SPR access functions

Function	Description
void mtspr(int reg, register_t val)	writes a special purpose register
register_t mfspr(int reg)	reads a special purpose register

11.1.2.12.3 Guest Register access

Header file:

greg.h

Table 550. Guest Register access functions

Function	Description
int read_gspr(trapframe_t *regs, int spr, register_t *val)	Reads special purpose registers.
int write_gspr(trapframe_t *regs, int spr, register_t val)	Reads special purpose registers
int read_ggpr(trapframe_t *regs, int gpr, register_t *val)	Reads general purpose registers
int write_ggpr(trapframe_t *regs, int gpr, register_t val)	Reads general purpose registers
int read_gfpr(trapframe_t *regs, int gpr, register_t *val)	Reads floating point registers
int write_gfpr(trapframe_t *regs, int gpr, register_t val)	Reads floating point registers
void read_gmsr(trapframe_t *regs, register_t *val)	Reads the machine state register
void write_gmsr(trapframe_t *regs, register_t val, int as_guest)	Reads the machine state register.
void read_gcr(trapframe_t *regs, register_t *val)	Reads the condition register
void write_gcr(trapframe_t *regs, register_t val)	Reads the condition register
void read_gpc(trapframe_t *regs, register_t *val)	Reads the program counter
void write_gpc(trapframe_t *regs, register_t val)	Writes the program counter

Table continues on the next page...

Table 550. Guest Register access functions (continued)

Function	Description
int read_pmr(trapframe_t *regs, int pmr, register_t *val)	Reads a PMR (performance monitor register)
int write_pmr(trapframe_t *regs, int pmr, register_t val)	Writes a PMR (performance monitor register)

Table 551. Guest Register access function parameters

Parameter	Description
trapframe_t *regs	The trapframe of the current context.
int spr	The spr number
int gpr	The gpr number
register_t *val	A pointer to where the data being read should be placed-- a hypervisor effective address
register_t val	The data being written.
int as_guest	The **as_guest** parameter specifies whether the access is being done 'as a guest' or by the hypervisor for internal use. Value values are: 1 0 The value 1 should be used when a stub is making an MSR access on behalf of a host debugger which is debugging a **guest**. The value 0 should be used when the stub needs to write the MSR for its own private use, such as setting DE to enable debug interrupts.

Return values

The read_gspr, write_gspr, read_ggpr, write_ggpr, read_gfpr, and write_gfpr functions return an int.

Table 552. Guest Register access return values

Return Code	Description
0	The operation succeeded
non-zero	an invalid argument was passed (e.g. bad register number)

11.1.2.12.4 Memory access

11.1.2.12.4.1 By Guest Effective Address

APIs to access a guest's virtual memory are provided. These functions assume the current context of the guest—i.e. the LPID, PID, and AS (address space).

Header file:

guestmemio.h

Table 553. By Guest Effective Address functions

Function	Description
static inline void guestmem_set_data(trapframe_t *regs)	Must be called for accessing **data**. Sets the address space context for subsequent guestmem_in/out calls.
static inline void guestmem_set_insn(trapframe_t *regs)	Must be called for accessing **instructions**. Sets the address space context for subsequent guestmem_in/out calls.
int guestmem_in32(uint32_t *ptr, uint32_t *val)	Reads 32-bits of data
int guestmem_in8(uint8_t *ptr, uint8_t *val)	Reads 8-bits of data
int guestmem_out32(uint32_t *ptr, uint32_t val)	Writes 32-bits of data
int guestmem_out8(uint8_t *ptr, uint8_t val)	Writes 8-bits of data
int guestmem_icache_block_sync(char *ptr)	Should be called for modifications to instructions. For the cache block corresponding to the effective address of ptr, flushes the data cache and invalidate the icache.

Table 554. By Guest Effective Address function parameters

Parameter	Description
trapframe_t *regs	The trapframe of the current context.
uint32_t *ptr uint8_t *ptr	The guest effective address where the memory access is to be made
uint8_t *val uint32_t *val	A pointer to where the data being read should be placed—a hypervisor effective address
uint8_t val uint32_t val	The data being written

Return values

The guestmem_in32, guestmem_in8, guestmem_out32, and guestmem_out8, functions return an int with the following meaning:

Table 555. By Guest Effective Address return values

Return Code	Description
GUESTMEM_OK	The operation succeeded
GUESTMEM_TLBMISS	A TLB miss/error occurred
GUESTMEM_DSI	A data storage interrupt occurred

NOTE

For access to virtual addresses other than the current context a stub may use the external PID load/store facilities of the CPU.

11.1.2.12.4.2 By Guest Physical Addresses

Functions are available to access a guest memory by guest physical address. The API handles setting up a temporary TLB mapping for the access.

Header file:

paging.h

Table 556. By Guest Physical Addresses functions

Function	Description
size_t copy_to_gphys(pte_t *tbl, phys_addr_t dest, void *src, size_t len int cache_sync)	Write guest memory using a guest physical address. Returns the number of bytes successfully copied.
size_t copy_from_gphys(pte_t *tbl, void *dest, phys_addr_t src, size_t len)	Read guest memory using a guest physical address. Returns the number of bytes successfully copied.

Table 557. By Guest Physical Addresses function parameters

Parameter	Description
pte_t *tbl	Pointer to the guest physical page table. This pointer is available in the per-virtual CPU data structure e.g. get_gcpu()->guest->gphys
phys_addr_t dest	Guest physical address to copy to
phys_addr_t src	Guest physical address to copy from
phys_addr_t addr	Guest physical address to map
void *src	Hypervisor effective address to copy from

Table continues on the next page...

Table 557. By Guest Physical Addresses function parameters (continued)

Parameter	Description
void *dest	Hypervisor effective address to copy to
int cache_sync	If non zero causes the i-cache to be synchronized. Will flush the data cache and invalidate the icache for the range of addresses written. This flag should be used when writing instructions.

Function	Description
void *map_gphys(int tlbentry, pte_t *tbl, phys_addr_t addr, void *vpage, size_t *len, int maxtsize, register_t mas2flags, int write)	Temporarily map guest physical memory into a hypervisor virtual address. Returns the hypervisor effective address of the mapping. If the region being mapped is not accessible to the guest the return value is NULL. The mapping is temporary and does not need to be cleared.

Parameter	Description
int tlbentry	A temporary TLB index. Must be the value TEMPTLB1 or TEMPTLB2
pte_t *tbl	Pointer to the guest physical page table. This pointer is available in the per-virtual CPU data structure e.g. get_gcpu()->guest->gphys
phys_addr_t addr	Guest physical address to map
void *vpage	virtual base of window to hold mapping. Use the virtual address that the hypervisor has initialized in temp_mapping[].
size_t *len	number of bytes to map
int maxtsize	maxtsize tsize of the virtual window
register_t mas2flags	MMU assist register 2 flags which specify whether the mapping is memory or I/O. Value values are: TLB_MAS2_IO TLB_MAS2_MEM
int write	if non-zero, fail if write access is not allowed

Example:

```
vdest = map_gphys(TEMPTLB1, tbl, dest, temp_mapping[0],
                  &chunk, TLB_TSIZE_16M, 1);
```

11.1.2.12.5 Guest TLB Search

The guest TLBs can be searched using the following API which works in a manner very similar to the **tlbsx** instruction.

Header file:

`tlb.h`

Table 558. Guest TLB Search functions

Function	Description
`int guest_tlb_search(phys_addr_t ea,` `int as, int pid, tlb_entry_t *mas);`	Searches the guest TLB with the lookup controlled by the effective address in **ea**, the address space specified in **as**, and the process id specified in *pid*. The function returns the values of mas0, mas1, mas2, mas3, and mas7 as defined by the e500 architecture.

Table 559. Guest TLB Search function parameters

Parameter	Description
uintptr_t ea	The effective address to search for.
int as	The address space to search in-valid values are 0 and 1
int pid	The process ID to search for
mas_regs_t *mas	Pointer to a struct containing the MMU assist register values.

Data from this API is returned in the tlb_entry_t struct, the address of which is passed in the call. The values of the The mas_regs_t struct is defined as:

```
typedef struct mas_regs {
    register_t mas0;
    register_t mas1;
    register_t mas2;
    register_t mas3;
    register_t mas7;
} tlb_entry_t;
```

The values of the mas register fields is identical to that of the physical CPU. See the e500 specific core reference manual for details.

Return values

A return value of 0 indicates the function complete without error. A non-zero value indicates an error occurred

11.1.2.12.6 Reading Guest TLBs

The guest_tlb_read() API provides a mechanism by which a stub can iterate over and read the TLBs for a vcpu.

The TLBSEL field in mas0 selects which TLB is being read.

Header file:

tlb.h

Table 560. Reading Guest TLBs functions

Function	Description
int guest_tlb_read(tlb_entry_t *mas, uint32_t *flags)	Iterates over and reads entries from a guest TLB. A **mas** struct is passed as an argument and the TLB to be read is selected by setting the TLBSEL bit in the mas0 field. To start an iteration sequence the flags argument must have the TLB_READ_FIRST bit set.

Table 561. Reading Guest TLBs function parameters

Parameter	Description
mas_regs_t *mas	A pointer to a guest_mas_t structure which contains the fields read from the TLB entry. The encoding is identical to the MAS registers defined in the hardware architecture Refer to the Core Reference Manual for the specific encoding of each MAS register. The TLB to be read is selected by setting the TLBSEL bit in the mas0 field. Note: between iterations of invoking this API the values in the 'gmas' struct must not be changed.
uint32_t *flags	An in/out parameter. A bitmask of flags to the API as described below: **Flag**: TLB_READ_FIRST **Description**: Specifies that the first TLB entry shall be returned, regardless of the state of the *mas* struct. This flag is self clearing.

Table 562. Reading Guest TLBs return values

Return value	Description
0	success
ERR_NOTFOUND	There are no more TLB entries to be read.

A return value of 0 indicates the function complete without error. A non-zero value indicates an error occurred

11.1.2.12.7 CCSR space access

Stubs can get direct access to the CCSR region the stub must get a base address by mapping the region using the map() API. However, note, this allows the stub access to all the CCSR region, not just the portion allocated to the partition. To restrict accesses to only the regions accessible to the partition use the guest physical access methods (see By Guest Physical Addresses on page 2249).

11.1.2.12.7.1 CCSR Map

Header file:

paging.h

Table 563. map functions

Function	Description
void *map(phys_addr_t paddr, size_t len, int mas2flags, int mas3flags)	Given a physical address, create a permanent hypervisor mapping.

Table 564. map function parameters

Parameter	Description
phys_addr_t paddr	paddr base of physical region to map
size_t len	len length of region to map
int mas2flags	mas2flags WIMGE bits
int mas3flags	mas3flags permission bits

Return values

returns the virtual address of the mapping, or NULL if out of resources

11.1.2.12.7.2 get_ccsr_phys_addr

Header file:

hv.h

Table 565. get_ccsr_phys_addr functions

Function	Description
phys_addr_t get_ccsr_phys_addr(size_t *ccsr_size)	get the CCSR physical address from the device tree

Table 566. get_ccsr_phys_addr function parameters

Parameter	Description
phys_addr_t paddr	paddr base of physical region to map
size_t len	len length of region to map
int mas2flags	mas2flags WIMGE bits
int mas3flags	mas3flags permission bits
size_t *ccsr_size	if not NULL is the returned size of CCSR, in bytes

Return values

returns the physical address of CCSR space, or NULL if there was an error

11.1.2.12.7.3 Example

```
          size_t ccsr_size;
          phys_addr_t ccsr_pa = get_ccsr_phys_addr(&ccsr_size);
          uint32_t *ccsr_va = map(ccsr_pa, ccsr_size, TLB_MAS2_IO,
TLB_MAS3_KERN);
```

11.1.2.12.8 I/O space access functions

A stub may access memory mapped I/O space using IO functions defined in **libos/include/libos/io.h**

Table 567. I/O space access functions

Function	Description
static inline uint8_t in8(const uint8_t *ptr);	Reads 8-bits of data from memory mapped I/O space
static inline uint16_t in16(const uint16_t *ptr);	Reads 16-bits of data from memory mapped I/O space
static inline uint32_t in32(const uint32_t *ptr);	Reads32-bits of data from memory mapped I/O space
static inline out8(const uint8_t *ptr, uint8_t val);	Writes 8-bits of data to memory mapped I/O space
static inline out16(const uint16_t *ptr, uint16_t val);	Writes 16-bits of data to memory mapped I/O space
static inline out32(const uint32_t *ptr, uint32_t val);	Writes 32-bits of data to memory mapped I/O space

11.1.2.12.9 Virtual CPU sleep state

An API is provided for a debug stub to query the sleep state of a virtual CPU. A vcpu can be running, in an idle state, or in a nap state.

Header file:

hv.h

Table 568. Virtual CPU sleep state functions

Function	Description
int get_vcpu_state(guest_t *guest, unsigned int vcpu)	get the sleep state of the specified vcpu

Table 569. Virtual CPU sleep state function parameters

Parameter	Description
guest_t *guest	pointer to the guest structure
unsigned int vcpu	Virtual CPU number to query

Return values

Value returned is one of the following:

```
FH_VCPU_RUN

FH_VCPU_IDLE

FH_VCPU_NAP
```

11.1.2.12.10 Restart a Partition

The current partition (which the stub is debugging) can be restarted via the restart_guest() API.

Header file:

hv.h

Table 570. Restart a Partition functions

Function	Description
void restart_guest(guest_t *guest)	Restarts the current partition.

Any hypervisor loaded images are automatically loaded on a partition restart. To prevent this, set the guest_t no_auto_load field to 1.

Table 571. Restart a Partition function parameters

Parameter	Description
guest_t *guest	Pointer to the guest structure.

11.1.2.12.11 Memory Allocation

For private dynamic memory allocation, stubs have access to a standard set of C memory allocation routines such as malloc() and free().

Refer to the **libos/include/libos/malloc.h** include file for a full description of these interfaces.

11.1.2.12.12 Device trees

11.1.2.12.12.1 Device Trees Overview

Three conceptual device trees exist in the NXP Embedded Hypervisor architecture that may be of interest to a debug stub or a debug stub: 1) the hardware device tree, 2) the hypervisor configuration tree, 3) guest device trees.

The hardware device tree is an ePAPR-compliant provided by boot firmware (u-boot) and simply describes the hardware of the system. The hardware device tree is accessible by a global variable (declared in init.c):

```
dt_node_t *hw_devtree;
```

The hypervisor configuration blob describes the configuration of the hypervisor and all partitions. See the NXP Embedded Hypervisor specification for complete details. The configuration device tree is accessible by a global variable (declared in init.c):

```
dt_node_t *config_tree;
```

Each partition has its own guest device tree. The guest device tree is the view of hardware and virtual resources as seen by guest software. This is accessible via pointer in the guest_t structure.

```
struct dt_node *devtree;
```

All the device trees can be accessed and parsed using hypervisor routines (see include/devtree.h).

11.1.2.12.12.2 Stub Node in the Hypervisor Configuration Tree

To have a byte-channel communications mechanism, a byte-channel must be defined for each debug stub and the association between CPUs and byte-channel numbers must be specified in the partition node defining the partition in the hypervisor configuration tree (see Debug Stub Node on page 2230 for details).

11.1.2.12.13 Hypervisor gevents

The hypervisor provides a mechanism for event signaling that enables one component to signal another component (including across physical CPUs) referred to as a gevent. A gevent uses an event number and a callback to run when the gevent is signaled.

An gevent handler function must have the following prototype:

```
void function_name(trapframe_t *regs)
```

Gevents handlers are registered with the register_gevent() API. The gevent handler will always run with the trapframe_t pointing to the guest context. The interrupt context of the handler is that of external interrupts, meaning that external interrupts are disabled (MSR[EE]=0. When the handler returns it will return to the guest (unless other gevents or interrupts were pending).

A common example for using gevents for a debug stub is to handle received data on a byte-channel. When data is received on a debug stub's byte-channel, a stub-defined receive-data callback is usually invoked. This callback runs in the context of the critical interrupt on the CPU that received the UART interrupt. However the debug stub consuming data from the bytechannel may be running on a different physical CPU. The gevent mechanism can be used to invoke to invoke the main loop of the debug stub on a different CPU as if the exception came from the guest—i.e. the trapframe is a guest trapframe.

Signaling Events

To signal an event use the setgevent() routine.

Table 572. Signaling Events functions

Function	Description
int register_gevent(eventfp_t handler)	Register a gevent handler. Return value is the gevent number.
void setgevent(gcpu_t *gcpu, int event)	Send an event to a target CPU. Parameters are a pointer to the target gcpu_t struct and the event number.

Table 573. Signaling Events function parameters

Parameter	Description
gcpu_t *gcpu	Pointer to the gcpu structure.
int event	Gevent number

11.1.2.12.14 Byte-channels

11.1.2.12.14.1 Byte-channel Registration

A debug stub is responsible for initializing its byte-channel communications channel. This must be done once per virtual CPU and normally would be done in the vcpu_init() callback.

Header file:

```
byte_chan.h
```

Table 574. Byte-channel Registration functions

Function	Description
int init_byte_channel(dt_node_t *node)	Given a pointer to the stub's configuration node initializes the byte-channel. Return value 0 - the function completed normally 1 - an internal error occurred If the function is successful node->bch contains the handle of the byte-channel. If this field is NULL the init_byte_channel() API failed.

The byte_chan_handle_t structure set by init_byte_channel() contains a send and receive queue which can configured with callbacks.

To define a callback for data received, set the data_avail field in the queue structure. The optional consumer field can specify private data to be passed to the callback.

To define a callback when space is available in the queue, set the space_avail field in the queue structure. The optional producer field can specify private data to be passed to the callback.

Example

```
vcpu_init callback:
 init_byte_channel(stub->node);

                if (!stub->node->bch) {

                        printlog(LOGTYPE_DEBUG_STUB, LOGLEVEL_ERROR,

                                 "%s: gdb stub byte chan allocation failed\n",

                                 __func__);

                        return;

                }
vcpu_start callback:
/* register the callbacks */

                stub->node->bch->rx->consumer = gcpu;

                smp_lwsync();

                stub->node->bch->rx->data_avail = rx;
```

In the example below, a receive callback for a byte-channel sends an event indicating that data is received.

```
        /** Callback for data received

         */

        static void rx_callback(queue_t *q)

        {

                setgevent((gcpu_t*)q->consumer, GEV_GDB);

        }
```

11.1.2.12.14.2 Byte-channel Send/Receive of Data

Routines are available to send and receive data on a byte-channel.

Table 575. Byte-channel Send/Receive of Data functions

Function	Description

ssize_t byte_chan_send(byte_chan_handle_t *bc, const uint8_t *buf, size_t len)	Given a handle to a byte-channel, send a buffer of character data.
	Parameters
	`bc // handle to the byte-channel to use`
	`buf // the buffer to send`
	`len // the number of bytes to send`
	Return value
	The number of bytes sent

ssize_t byte_chan_receive(byte_chan_handle_t *bc, uint8_t *buf, size_t len)	Given a handle to a byte-channel, receive a buffer of character data.
	Parameters
	`bc // handle to the byte-channel to use`
	`buf // the buffer to fill with data`
	`len // the size of the buffer`
	Return value
	The number of bytes received

size_t queue_get_avail(const queue_t *q)	Returns the number of available byte of data in the queue
size_t queue_get_space(const queue_t *q)	Returns the amount of available space (in bytes) in the queue

Example:

To see how many bytes of data are available on the receive queue of a byte-channel

```
        count = queue_get_avail(bc->rx);
```

11.1.2.13 Hypervisor customization

11.1.2.13.1 Hypervisor Customization Overview

In addition to the creation of debug stubs, the APIs defined in can also be used to implement custom modifications to the hypervisor for several uses described in this section.

11.1.2.13.2 Custom Reset

The hypervisor implements a system reset function that can be accessed by privileged partitions via an hcall (see FH_SYSTEM_RESET on page 2186). By default the hypervisor implementation simply write the SOC's Global Utilities reset control register to initiate a hardware reset request.

The hypervisor's reset function can be overridden by a custom implementation of a function with the following prototype (defined in **include/guts.h**):

```
int system_reset(void);
```

The implementation should take any system-specific steps that are necessary to reset the system. If the reset occurs normally, the function will never return. If **system_reset** fails for any reason, the function must return a non-zero return value.

A new source file can be added to the hypervisor by adding a line to the **Makefile.build** hypervisor source file: :

Example:

```
hv-src-y += my_reset.c
```

Any custom code added to the hypervisor should only used documented interfaces (see Hypervisor APIs on page 2245).

11.1.2.13.3 System Health Monitor

The hypervisor implements a watchdog timer to detect severe system failures that require a system reset. The hypervisor watchdog interrupt executes a customizable system health monitor routine that can periodically monitor system specific conditions to determine whether the system is in a healthy state.

The hypervisor's default stub implementation of the system health check function can be overridden by implementing a function with the following prototype (defined in **include/timers.h**):

int system_health_check(void);

The implementation should take any system-specific steps that are necessary to monitor and determine the health of the system. If the system is determine to be healthy, a value of zero must be returned. If the system needs to be reset a non-zero value must be returned.

A new source file can be added to the hypervisor by adding a line to the Makefile.build hypervisor source file:

Example:

hv-src-y += my_sys_health_check.c

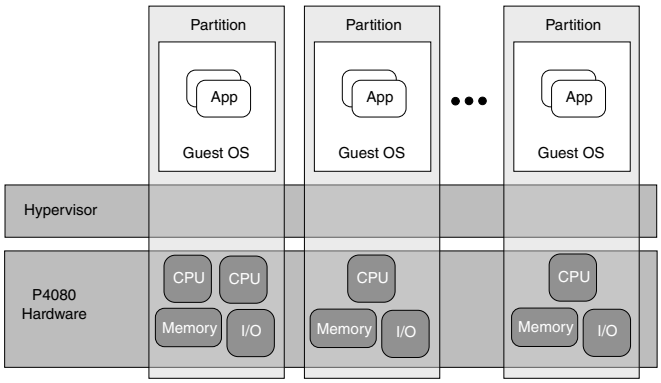
Any custom code added to the hypervisor should only used documented interfaces (see Hypervisor APIs on page 2245).

11.1.3 Embedded Hypervisor Software User Manual

11.1.3.1 Introduction

This document describes how to create partitioned systems using the NXP Embedded Hypervisor software. The architecture of the hypervisor is described in the *NXP Embedded Hypervisor Software Reference Manual* [2].

The NXP Embedded Hypervisor is a layer of software that enables the efficient and secure partitioning of a multi-core system. A system's CPUs, memory, and I/O devices can be divided into logical groupings or *partitions*. Each partition is capable of executing a guest operating system.



Key features of the hypervisor software architecture are summarized below—

- Partitioning. Support for partitioning of CPUs, memory, and I/O devices:
  - CPUs. Each partition is assigned one or more CPU cores.
  - Memory. Each partition has a private memory region that is only accessible to the partition that is assigned the memory. In addition, shared memory regions can be created and shared among multiple partitions.
  - I/O devices. SoC I/O devices may be assigned directly to a partition (direct I/O), making the device a private resource of the partition, and providing optimal performance.
- Protection and Isolation. The hypervisor provides complete isolation of partitions, so that one partition cannot access the private resources of another. The SoC PAMU (an IOMMU) is managed by the hypervisor to ensure device-to-memory accesses are constrained to allowed memory regions only.
- Sharing. Mechanisms are provided to selectively enable partitions to share certain hardware resources (such as memory).
- Virtualization. Support for mechanisms that enable the sharing of certain devices among partitions such as the system interrupt controller.
- Performance. The hypervisor software uses the features of the NXP Embedded Hypervisor APU (see reference []) to provide security and isolation with very low overhead. Guest operating systems take external interrupts directly without hypervisor involvement providing very low interrupt latency.
- Ease of migration. The hypervisor uses a combination full emulation and para-virtualization to maintain high performance and requiring minimal guest OS changes when migrating code from a physical CPU to the hypervisor.

Refer to the reference manual [2] for a detailed description of the hypervisor software architecture including a description of virtual CPUs, hypercall services, and device tree properties.

A high-level understanding of the **power.org** *ePAPR* [1] specification (and device trees) is a pre-requisite for understanding the content of this document. The hypervisor adopts the device tree and multi-core boot architecture of the ePAPR. In ePAPR terminology, the hypervisor is a *client program* of the boot firmware (e.g. u-boot). It also is a *boot program* in its role of booting partitions.

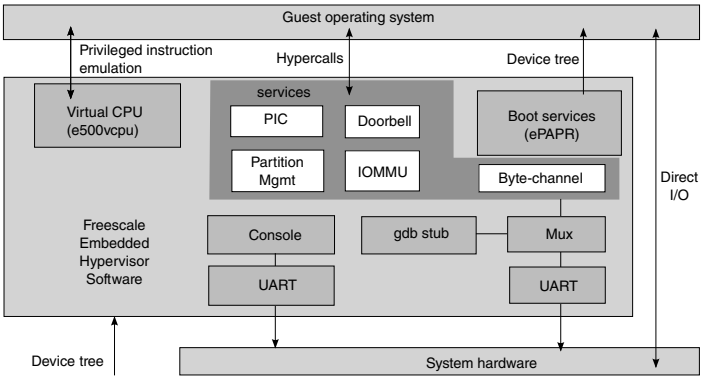
A device tree is data structure that represents system hardware. It consists of nodes arranged in a tree structure with each node having properties/value pairs. CPUs, memory, and I/O devices are represented by nodes. Refer to the ePAPR for a description of device trees.

11.1.3.1.1 Hypervisor Software Architecture Overview

The NXP Embedded Hypervisor provides the capability to create partitions in which guest software can in isolation without the possibility of interference from other partitions in the system.

The software architecture consists of several key components:

- Virtual CPU (vcpu)—Software executing in a partition sees a view of the CPU core that is nearly identical to the physical CPU, except for some CPU state which is hypervisor privileged and not accessible. The reference manual [2] describes the virtual CPU in detail
- Hypercall services—Some services are provided via hypercalls, including: the interrupt controller, byte-channels, inter-partition doorbells, IOMMUs, and partition management. Hypercalls are like system calls where guest software can make explicit requests for services to the hypervisor.
- Boot services—The hypervisor adopts the device tree and multicore boot architecture of the power.org ePAPR specification. This defines mechanisms for resource discovery within a partition and for starting multiple CPUs in a single partition.
- Debug services—Guest software with a debug stub can use the stub without change through the capabilities of the vcpu. In addition, the hypervisor provides a gdb stub in the hypervisor that provides an interface for a host debugger to use to debug guest operating systems.
- Direct I/O—high speed peripherals are not virtualized and are assigned and managed directly by partitions. External interrupts are taken by partitions directly with no hypervisor involvement.



11.1.3.1.2 References

[1] Power.org *Standard for Embedded Power Architecture Platform Requirements (ePAPR)*. power.org, 2011.

[2] *Embedded Hypervisor Software Reference Manual* v1.0, 2011

[3] *e500mc Core Reference Manual: Programming Model* (E500MCRM)

[4] *e5500 Core Reference Manual: Programming Model*

[5] *e6500 Core Reference manual*

[6] *EREF 2.0: Aprogrammer's Reference Manual for Power Architecture Processors*

11.1.3.1.3 Conventions used in the Examples

Many of the examples in this document use *device tree syntax* to express device trees. This syntax is defined in the ePAPR [1].

In device trees property values that are numbers (e.g. addresses) are expressed in terms of cells (each cell being a 32-bit value). For the examples in this document the assumption is that unless otherwise stated that device tree nodes are in a context where *#address-cells* and *#size-cells* both have a value of 2-i.e. addresses are expressed as 64-bit values. See the ePAPR for information about the *#address-cells* and *#size-cells* properties.

11.1.3.2 Partitioning a system with embedded hypervisor software

11.1.3.2.1 Hypervisor boot overview

11.1.3.2.1.1 Hypervisor Boot Sequence

The hypervisor must be booted with an ePAPR compliant boot program such as u-boot. The boot program initialization includes the following:

- Probing and initialization DDR memory
- Configuring the physical memory map of the system (sets up Local Access Windows (LAWs for memory and I/O regions)
- Enabling caches
- Configuring clocks
- Allocating and programming certain hardware identifiers (e.g. cache stash ids, logical I/O device numbers)
- Loading the hypervisor program image into memory (e.g. from flash)
- Creating a spin table as specified by the ePAPR to facilitate multi-cpu boot.
- Releasing all secondary CPUs from reset and set them spinning in the spin table
- Configuring the *hardware device tree* and loading it into memory with the *bootargs* property on the */chosen* node set to specify the address of the hypervisor configuration tree
- Transferring control to the hypervisor program image on the boot cpu

As part of its initialization the hypervisor releases all secondary CPUs from the spin table set up by the boot program. The hypervisor then initializes each partition and boots guest operating systems as per each partition's configuration (see Guest boot overview on page 2264).

11.1.3.2.1.2 Device Tree Overview

A device tree is a tree-based data structure defined by the ePAPR [1] which describes resources available to an operating system. When the control is passed to the operating system a pointer to the device tree is passed.

Device trees are developed in a text format referred to as "device tree syntax" or DTS. These text DTS files are compiled into a binary form referred to as "device tree blob" or DTB. An open source tool, DTC, is available to do this compilation.

11.1.3.2.1.3 Hardware Device Trees, Configuration Trees, and Guest Device Trees

The hypervisor will be passed an ePAPR-compliant "hardware device tree" (from u-boot) when it is booted that describes all the hardware in the system.

A separate tree of configuration information is loaded into memory by the boot program with its address passed to the hypervisor using the *bootargs* property of the hardware device tree's /chosen node. This configuration information is referred as the "hypervisor configuration tree", and it contains all hypervisor configuration parameters and partition definition information. Based on the configuration tree, the hypervisor will create partitions. The hypervisor dynamically creates ePAPR-compliant guest device trees for each partition. A guest device tree describes the resources available to the partition including —CPUs, memory, I/O devices, and other virtual resources.

See below—u-boot copies the hardware device tree and configuration tree into hypervisor memory. The hypervisor places a guest device tree in the memory of each guest.

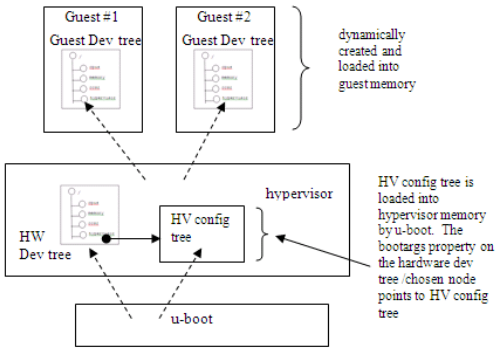


Figure 352.

Hypervisor configuration trees are described in detail in Partitioning a system on page 2265.

11.1.3.2.2 Guest boot overview

11.1.3.2.2.1 Guest Boot Sequence

The hypervisor uses the hardware device tree and the configuration tree as input to create and instantiate all partitions. The hypervisor does the following to boot a partition:

- Does internal housekeeping to instantiate and create the partition. This includes configuring the PAMUs for the devices owned by the partition.
- Creates a guest device tree and copies it into the partition's memory
- Loads all images into the partition's memory as specified by the *load-image-table, guest-image*, and *linux-rootfs* properties.
- Creates an ePAPR compliant spin table and sets the cpu node's *status, enable-method*, and *release-addr*
- Initializes the virtual CPU as per the requirements of the ePAPR
- Transfers control to the entry point of the guest operating system on the boot cpu for that partition

NOTE

Note: some partitions may not have their images loaded and be started by the hypervisor. A usage model is also supported where a single partition can be initially started which manages other partitions, including loading images and starting them.

11.1.3.2.2.2 Guest Device Trees

The device tree which is passed from the hypervisor to the guests is referred to as a *guest device tree*. This device tree describes 1) the hardware resources (cpu, memory, I/O devices) available to the partition, 2) virtual resources available to a partition (e.g. virtual mpic, byte-channels), and 3) partition configuration properties (e.g. partition name). See the reference manual [2] for details.

11.1.3.2.3 Guest Physical Addresses

Guest software executing in a partition sees a physical address space created by the hypervisor that may not directly correspond to physical addresses in the system hardware. System hardware addresses are referred to in this document as **true** *physical addresses*. The addresses in the guest's physical address space are referred to as **guest** *physical addresses*. For example, there may be multiple partitions, each with memory at guest physical address 0x0—where within each partition, address 0x0 corresponds to a unique *true* physical address (see the example in the figure below).

In the definition of a partition, physical memory regions can be assigned to the partition and the guest physical to true physical mapping can be specified (see *guest physical memory areas* in [2]). The hypervisor maintains this mapping in a manner that is completely transparent to guest software. Guest software is made aware of the *guest* physical address ranges available to it via the guest device tree passed to the partition at initialization.

Guest software has no knowledge of and has no means to determine true physical addresses. DMA transactions are programmed by guest software in terms of guest physical addresses.

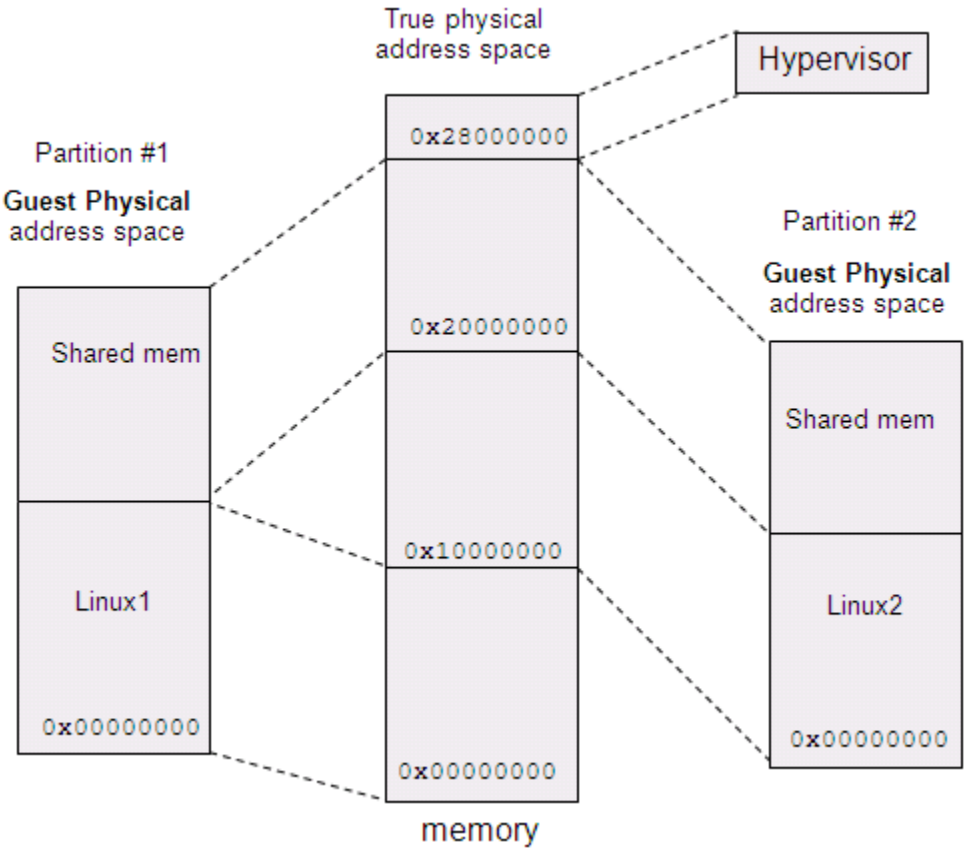


Figure 353. guest physical and true physical addresses

In the above example, partition #1 has 256 MB of private memory at guest physical 0x0, which also maps to true physical 0x0.

Partition #2 has 256 MB of private memory at guest physical 0x0 that corresponds to true physical 0x10000000.

The partitions share a 128 MB region of shared memory that is located at true physical 0x20000000. Both partitions see that shared memory at guest physical 0x10000000.

11.1.3.3 Partitioning a system

This section describes the key aspects of defining a partition in the hypervisor configuration tree.

The hypervisor configuration tree is physically a device tree, but is in a custom form for hypervisor use only and is not ePAPR compliant.

The hypervisor configuration tree defines all configurable parameters and all partitions:

- Physical memory areas (PMAs)—specify how physical memory is partitioned
- DMA windows—a definition of regions to which DMA devices are allowed to access
- General hypervisor configuration parameters, including:

  - The memory region reserved for private hypervisor use
  - The physical devices assigned to the hypervisor
  - Hypervisor console configuration
- Byte-channel mux—the declaration of any byte-channel multiplexers
- Doorbells—a declaration of all doorbells
- Partition definition—each partition has a partition node that specifies properties of the partition
  - Assignment of physical CPUs to a partition
  - List of guest physical memory areas (each referencing a physical memory area)
  - Device nodes—a set of nodes which assign I/O devices and portals to a partition. For DMA-capable devices the node describes which DMA window table the device should use
  - Image loading properties—the location of any images to be loaded by the hypervisor
  - Guest address where the device tree should be placed
  - Doorbells assigned to the partition
  - Byte-channels—a table that defines all byte-channels in the partition and the UART or mux endpoints each is attached to
  - Guest aliases
  - Modifications to the guest device tree such as the guest's /chosen node

11.1.3.3.1 Configuration Tree Structure

The hypervisor configuration tree is represented as a device tree as described by the ePAPR. However, it does not describe hardware and is not ePAPR compliant. It is specifically designed for hypervisor configuration.

All addresses in the configuration tree are 64-bits and thus consist of two cells.

The root node of the configuration tree must contain a compatible property that identifies the device tree as a hypervisor configuration tree:

Example:

```
/dts-v1/;
/{
compatible = "fsl,hv-config";

// contents go here

};
```

When the hypervisor is booted by boot firmware it must be passed the address of the configuration tree in the bootargs property of the */chosen* node in the **hardware** device tree.

The bootargs property must include the following value: config-addr=[physical address of blob]

Example:

```
chosen {
     bootargs = "config-addr=0x00400000";
};
```

11.1.3.3.2 Partitioning memory

11.1.3.3.2.1 Physical Memory Areas

A key aspect of partitioning a system is to partition the system's physical memory. To do this physical memory is divided into *physical memory areas* (or PMAs). PMAs are statically defined by the system architect and cannot be changed dynamically in a running system. The hypervisor itself is also assigned a PMA for its internal use.

PMAs are defined in terms of **true physical** addresses.

When a partition is defined, *guest physical memory areas* (or GPMAs) are created which define references to the specific PMAs that are to be assigned to the partition. The GPMA defines the **guest physical** address at which the memory is seen in the partition's guest physical address map.

A partition-private memory region is defined by assigning a PMA to only a single partition. Shared memory regions (between partitions) can easily be created by simply assigning the same PMA to multiple partitions.

PMAs have several constraints:

- Size must be a power of 2
- Must be size aligned

The L3/CPC (Corenet platform cache) by default is shared by all PMAs—i.e. memory transactions to **any** memory address allocate in the CPC. The CPC supports a feature by which ways of the cache can be removed from the default set of ways allocated from and can be partitioned for private or shared use by certain PMAs.

Example

Table 576. PMA example

PMA	True physical address	Size	CPC ways to allocate to (default is all)	Comment
pma0	0x00000000	128MB		hypervisor
pma1	0x08000000	128MB		p2
pma2	0x10000000	256MB		p1
pma3	0x20000000	64MB		p3
pma4	0x60000000	64MB	30,31	shared, heap

Example

Below is an example of "pma0" and "pma4" (from table above) being defined in the hypervisor configuration tree. The "pma4" node specifies that ways 30 and 31 of the platform cache are allocated for pma4 and not for other PMAs.

```
pma0 {
    compatible = "phys-mem-area";
    addr = <0x0 0x00000000>;
    size = <0x0 0x08000000>;  // 128MB
};

pma4 {
    compatible = "phys-mem-area";
    addr = <0x0 0x60000000>;
    size = <0x0 0x04000000>;  // 64MB
```

```
                    allocate-cpc-ways = <30 31>;
                };
```

NOTE

The hypervisor uses the coherence subdomain feature of the QorIQ SoC to implement PMAs. This allows cache coherency snoop traffic in the system to be restricted to only the physical CPUs that actually access the memory. There are a limited number of coherence subdomains supported by the SoC hardware and thus only up to 28 PMA are supported with the full coherence subdomain support.

More than 28 PMAs are supported, but memory accesses would be snooped by all CPUs.

11.1.3.3.2.2 Shared Memory

As described in the section above, sharing memory between partitions is accomplished by assigning the same PMA to multiple partitions.

11.1.3.3.2.3 DMA Windows

The SoC's IOMMU (PAMU or *Peripheral Access Management Unit*) requires careful configuration to specify the allowable regions of memory that can be targeted by DMA-capable devices. Guest software specifies **guest physical** addresses when programming a device to do DMA. The PAMU (configured by the hypervisor) handles the translation of guest physical addresses to true physical. This translation of addresses is transparent to guest software.

PAMU supports defining power-of-two memory sized DMA windows which can be optionally be sub-divided into a power-of-two number of sub-windows. Please consult the chip specific Reference Manual for the maximum number of supported sub-windows. The sub-windows enable the definition of address regions that can be the target of DMA operations where the regions may be discontiguous in guest physical address space or true physical address space.

In the context of the hypervisor, a *DMA window* is conceptually a table of guest physical memory regions that are the valid targets of a DMA operation. Each row of the table consists of a guest physical address and size.

A DMA window describes a particular configuration of guest address space and may be applicable to one or more partitions.

Each DMA-capable I/O device must reference one DMA window.

Single Region DMA Window

If the DMA region being defined is a single contiguous region of memory a single DMA window can be defined. The DMA window is a table with one row defining the region.

Example 3-1, Single Contiguous Region

Example—window specifying one 256MB region

	Guest Physical	Size
0	0x00000000	256MB

The DMA window represented above would look like this in the hypervisor configuration tree:

```
        window0 {
            compatible = "dma-window";
            guest-addr = <0x0 0x00000000>;
            size = <0x0 0x10000000>; // 256 MB
        };
```

DMA Windows for Discontiguous Address Regions

If multiple, discontiguous address regions (whether with respect to the guest physical or true physical address spaces) are needed, DMA sub-windows are used to represent and configure the regions. In this case, the DMA window is conceptually a table with power-of-two rows, with each row defining one sub-window. There are several key parameters in defining sub-windows that need to be understood:

- **Total size**. This is the total size of the address space that may be the target of DMA-encompassing all address regions. This must be a power of 2. See the *size* property of dma-window node.
- **Base address**. This is the guest physical address at which the overall DMA region starts. This must be size aligned with the total size. See the *guest-address* property of dma-window node.
- **Sub-window count**. The number of sub-windows must be a power-of-two. Please consult the chip Reference Manual for the maximum number of supported sub-windows. Note, that this simply defines how many subwindows the total size is divided into. It does not specify the number of **valid** subwindows. See *subwindow-count* property of dma-window node.
- **Maximum sub-window size**. The maximum size of each sub-window is computed by: total-size / sub-window-count.
- **Sub-window starting address**. Each sub-window starts at an address computed by:

```
overall-starting-address + (sub-window index * maximum-sub-window-size)
```

- **Sub-window size**. Each sub-window can specify a size that may be smaller than the maximum sub-window size.

Example 3-1, Two address regions—Contiguous with respect to guest physical and discontiguous with respect to true physical

A partition has 128MB of memory at guest physical address 0x0, composed of two 64MB memory regions that are discontiguous with respect to true physical.

	Guest Physical	True Physical	Size
0	0x00000000	0x64000000	64MB
1	0x04000000	0x54000000	64MB

Below is a valid set of parameter for specifying this configuration.

Total size: 0x08000000 (128MB)

Base address: 0x00000000

Sub-window count: 2

Maximum sub-window size: 64MB

Sub-window starting address:

- For sub-window 0: 0x0
- For sub-window 1: 0x04000000

Sub-window size: 64MB for both sub-windows

The DMA window represented in would look like this in the hypervisor configuration tree:

```
        window1 {
                compatible = "dma-window";
                guest-addr = <0x0 0x00000000>;  // base address
                size = <0x00000000 0x08000000>; // 128MB
                subwindow-count = <2>;
```

```
                sub-window@0 {
                    compatible = "dma-subwindow";
                    guest-addr = <0x0 0x00000000>;
                    size = <0x0 0x04000000>; // 64 MB
                };
                sub-window@1 {
                    compatible = "dma-subwindow";
                    guest-addr = <0x0 0x04000000>;
                    size = <0x0 0x04000000>; // 64 MB
                };
        };
```

Example 3-2, Two address regions discontiguous with respect to both guest physical and true physical

A partition has 64MB of memory at guest physical address 0x0 and 64MB at 0x60000000. Both are the targets of DMA. The shaded rows in the table below show the discontiguous regions that are the valid destinations of DMA. The other sub-windows but are not defined since they are not valid DMA targets.

	Guest Physical	Size
0	0x00000000	64MB
1	0x10000000	-
2	0x20000000	-
3	0x30000000	-
4	0x40000000	-
5	0x50000000	-
6	0x60000000	64MB
7	0x70000000	-

Below is a valid set of parameter for specifying this configuration.

Total size: 0x80000000 (2 GB)

Base address: 0x00000000

Sub-window count: 8

Maximum sub-window size: 256MB

Sub-window starting address:

- For sub-window 0: 0x0
- For sub-window 6: 0x60000000

Sub-window size: 64MB for both sub-windows

The DMA window represented in would look like this in the hypervisor configuration tree:

```
        window1 {
                compatible = "dma-window";
                guest-addr = <0x0 0x00000000>;  // base address
                size = <0x00000000 0x80000000>; // 2GB
```

```
                subwindow-count = <8>;
                sub-window@0 {
                    compatible = "dma-subwindow";
                    guest-addr = <0x0 0x00000000>;
                    size = <0x0 0x04000000>; // 64 MB
                };
                sub-window@6 {
                    compatible = "dma-subwindow";
                    guest-addr = <0x0 0x60000000>;
                    size = <0x0 0x04000000>; // 64 MB
                };
        };
```

DMA Windows for Message Signaled Interrupts

PCI Express message signaled interrupts are generated by PCI devices writing to a interrupt controller register which causes the MSI. This register access must go through the PAMU like any other memory transaction, and thus a DMA subwindow must be defined for MSIs. Refer to the pcie-msi-subwindow property in the reference manual [2] for details.

DMA Windows for RapidIO

SRIO maintenance transactions are translated into regular memory accesses to the CCSR space and must be authorized by PAMU, and thus a DMA subwindow must be defined for such accesses. Refer to the *srio-ccsr-subwindow* property in the reference manual [2] for details.

Example 3-3 , DMA window for RapidIO

In the example below, a partition has 512MB of memory and support SRIO maintenance transactions, which requires a DMA window covering both memory and CCSR space. Memory is at guest physical 0x0 (0x70000000 true physical) and CCSR space is at 0xf_fe0000000. The first two sub-windows are used to map the guest regular memory and the last one is used to map the CCSR space.

NOTE

Note: A DMA window to cover the entire guest space is needed because the RapidIO uses the same LIODN for both maintenance transactions (which are accesses into the CCSR space) and general transactions (which are accesses in regular guest memory).

Sub-win index	Guest Physical	True Physical	Size
0	0x0_00000000	0x0_70000000	256M
1	0x0_10000000	0x0_80000000	256M
255	0xf_f0000000	0xf_f0000000	256M

Below is a valid set of parameter for specifying this configuration.

Total size: 0x1_000000000 (64GB)

Base address: 0x00000000

Sub-window count: 256

Maximum sub-window size: 256MB

Sub-window starting address:

- For sub-window 0: 0x0
- For sub-window 1: 0x0_100000000

- For sub-window 255: 0xf_f0000000

Sub-window size: 256MB for each sub-window

The DMA window represented in would look like this in the hypervisor configuration tree:

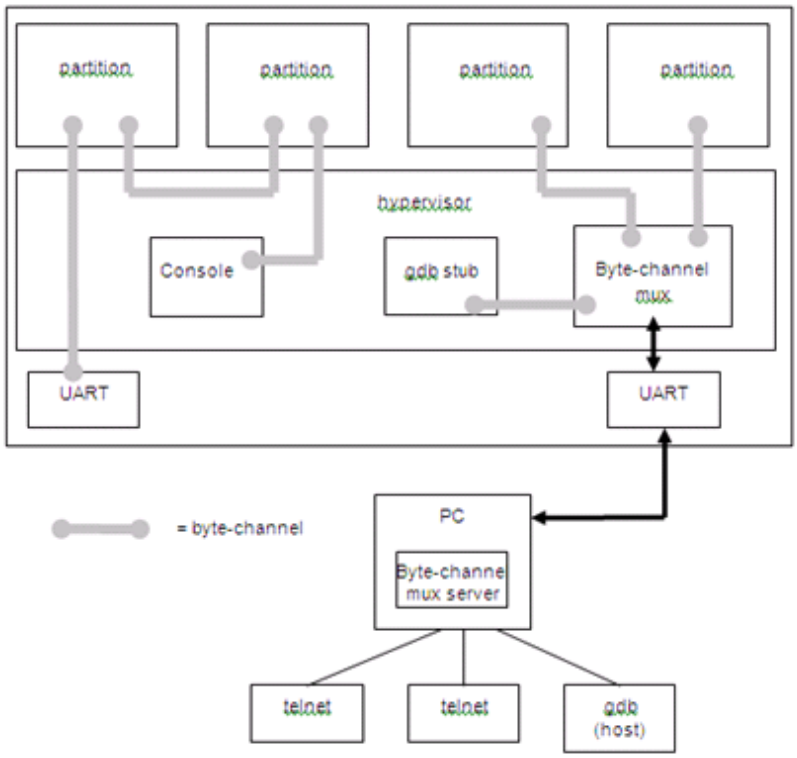
```
dw_srio: dw_srio {
        // DMA window for part1_linux
            compatible = "dma-window";
            guest-addr = <0 0>;
            size = <0x10 0x0>; //64GB
            subwindow-count = <256>;
            sub-window@0 {
                compatible = "dma-subwindow";
                guest-addr = <0 0>;
                size = <0x0 0x10000000>;
            };
            sub-window@1 {
                compatible = "dma-subwindow";
                guest-addr = <0x0 0x10000000>;
                size = <0x0 0x10000000>;
            };
            sub-window-ccsr@2 {
                compatible = "dma-subwindow";
                guest-addr = <0xf 0xf0000000>;
                size = <0x0 0x10000000>;
                srio-ccsr-subwindow;
            };
        };
```

11.1.3.3.3 Byte-channel Multiplexers and Byte-channels

Each partition created by hypervisor can have multiple byte-channel ports which provide a virtualized interrupt-driven character-based I/O channel connected to one of several endpoints:

- A physical UART
- Another byte-channel
- A byte-channel multiplexer (mux)
- A hypervisor GDB stub
- The hypervisor console

A byte-channel multiplexer (mux) multiplexes multiple byte-channels over a single physical link to a host computer, running a mux server which demultiplexes the streams of data.



A byte-channel mux is defined by creating a global node (parent is root) in the configuration device tree that specifies which serial node the mux is attached to.

Partitions that define a byte-channel attached to the mux can then reference the phandle to this node.

The endpoint property specified in the mux node must reference a hypervisor owned UART node.

11.1.3.3.4 Example

In the example below a byte-channel mux is defined. The endpoint value of <&serial1> refers to a physical serial port node that is assigned to the hypervisor.

```
uartmux: uartmux {
        compatible = "byte-channel-mux";
        endpoint = <&serial1>;
};
```

11.1.3.3.5 Doorbells

The hypervisor provides a doorbell mechanism that enables one partition to signal other partitions. This is a one-way signal with no payload which results in an external interrupt (IVOR4) in the destination partition. The interrupt is gated off of MSR[EE].

Flexible configuration of send and receive endpoints are provided. Inter-partition doorbell send and receive endpoints can be configured in any combination of one-to-one, one-to-many, many-to-one, or many-to-many.

A partition is aware of the doorbell endpoints (send or receive) available to it through its device tree which contains a node for each doorbell endpoint.

A doorbell send is done via an hcall.

The partition with the receive endpoint can configure which CPU within the partition receives the doorbell via the hypervisor's VMPIC services.

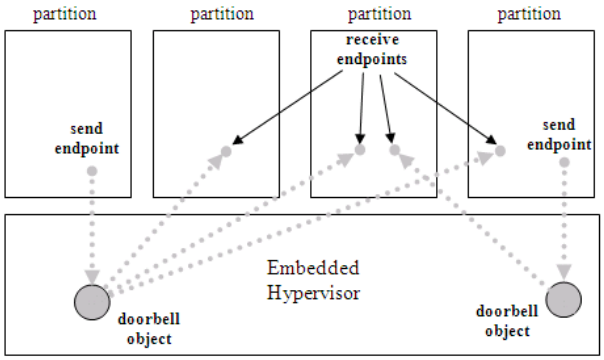


Figure 354. Doorbell connection example

A doorbell is defined by specifying a node (*compatible* = "doorbell") in the configuration device. This node is then referenced by partitions that want send and/or receive endpoints associated with that doorbell object.

11.1.3.3.5.1 Example—Doorbell Configuration

The example below shows the hypervisor configuration tree definition for a doorbell node that is referenced by three partitions —a send endpoint in partition #1, and receive endpoints in partitions #2 and #3. The definition of endpoints is described in Inter-partition Doorbell Configuration on page 2290.

```
doorbells {
        dbell0: doorbell0 {
                compatible = "doorbell";
        };
};

// Partition 1
part1 {
        compatible = "partition";
        ...
        doorbell0_send {
                compatible = "send-doorbell";
                global-doorbell = <&dbell0>;
        };
        ...
};

// Partition 2
part2 {
        compatible = "partition";
        ...
        doorbell0_recv {
                compatible = "receive-doorbell";
                global-doorbell = <&dbell0>;
        };
        ...
};

// Partition 3
part2 {
        compatible = "partition";
```

```
                ...
                doorbell0_recv {
                    compatible = "receive-doorbell";
                    global-doorbell = <&dbell0>;
                };
                ...
        };
```

11.1.3.3.6 Guest device tree node updates

The hypervisor uses the hardware device tree and the hypervisor configuration tree to create partitions and dynamically create guest device trees for the partitions. A "node update" mechanism is provided that allows a system architect to specify a set of directives to customize how the nodes in the guest device tree will look.

Some examples where this feature may be useful:

- adding a new property to a node originating from the hardware device tree
- deleting an unwanted subnode from a sub tree originating from the hardware device tree
- appending the compatible property of a doorbell node with a string that explicitly defines the purpose of the node
- defining a new subtree of nodes that specify custom configuration information

Updates are specified in a node with the name *node-update*. The properties of the *node-update* node specify the directives over the creation of nodes in the guest device tree.

Except for the special property names listed below, all properties under *node-update* will be copied exactly as is to the guest device tree. If the property already exists in the node it will be replaced. Note: phandle references will not be resolved for properties with phandle value. See the *node-update-phandle* node (see How to set the /chosen node for a Partition on page 2276) for node updates to properties with phandle values.

In addition to copying properties, the following special properties have special meaning:

- delete-prop
- delete-node
- delete-subnodes
- prepend-stringlist

Please refer to the *Embedded Hypervisor Software Reference Manual* [] for additional details on the *node-update* properties.

The following nodes in the hypervisor configuration tree support node-update nodes:

- Device nodes
- Doorbell nodes
- Byte-channel nodes
- Partition nodes
- Guest physical memory area nodes

The *node-update* node supports updates to descendants of the node which the node-update modifies. Updates to descendants can be done by replicating the node hierarchy under the *node-update* node using the names of the descendant nodes.

Note: phandles and phandle references in *node-update* nodes are not supported—use a *node-update-phandle* node (see How to set the /chosen node for a Partition on page 2276).

11.1.3.3.6.1 Example: node-update

Suppose a hardware device tree looked like the following:

```
stuff {
        foo@0 {
            reg = <4>;
            bar@10000 {
                    reg = <5>;
            };
        };
};
```

In a hypervisor configuration tree device node, to assign the "/stuff" node to the partition and add a *compatible* property to the "bar" node and delete the *reg* property, the *node-update* node could look like this:

```
stuff {
        device = “/stuff”
        node-update {
                foo@0 {
                    bar@10000 {
                        compatible = “xyz,mystuff”;
                        delete-prop = “reg”;
                    };
                };
        };
};
```

The resulting guest device tree would look like:

```
stuff {
        foo@0 {
                reg = <4>;
                bar@10000 {
                        compatible = “xyz,mystuff”;
                };
        };
};
```

11.1.3.3.6.2 How to set the /chosen node for a Partition

The /chosen node is a special device tree node defined by the ePAPR that may be used to pass configuration parameters to an operating system. In order to set the /chosen node for a partition, use *node-update* on the partition node.

Example

```
node-update {
    chosen {
        linux,stdout-path = "/serial2";
        bootargs = "console=ttyS0,115200";
    };
};
```

11.1.3.3.6.3 node-update-phandle

To support the creation or update of properties that reference other nodes using phandle values a special node with the name *node-update-phandle* is supported. If one property references another node by phandle both properties **must** be assigned to the partition.

Each property listed under *node-update-phandle* has a property name with a value that specifies the phandle of the referenced node in the configuration device tree.

Please refer to the *Embedded Hypervisor Software Reference Manual* [] for additional details on the *node-update-phandle* properties.

11.1.3.3.6.4 Example: node-update-phandle

In the example below a *node-update-phandle* property is used to create a new property with a value consisting of an array of phandles to other nodes from the hardware device tree assigned to this partition.

Configuration Tree

```
partition1 {
    ...
    qman-portalA {
            device = "/qman-portals@f4200000/qman-portal@0";
            vcpu = <0>;

            node-update-phandle {
            fsl,qman-pool-channels = <&qman-pool@1 &qman-pool@2>;
            };
    ...
    qman-pool@1 {
            device = "/qman-portals/qman-pool@1";
    };
    qman-pool@2 {
            device = "/qman-portals/qman-pool@2";
    };
};
```

Guest Device Tree

The guest device tree reflects the node-update-phandle being processed—the fsl,qman-pool-channels property has two phandles corresponding to the nodes being referenced.

```
...
qman-portal@24000 {
    cell-index = <0x9>;
    compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
    ...
    fsl,qman-pool-channels = <99 100>;
    ...
};
...
qpool1: qman-pool@1 {
    cell-index = <1>;
    compatible = "fsl,p4080-qman-pool-channel", "fsl,qman-pool-channel";
    fsl,qman-channel-id = <0x21>;
    phandle = <99>;
};
qpool2: qman-pool@2 {
```

```
            cell-index = <2>;
            compatible = "fsl,p4080-qman-pool-channel", "fsl,qman-pool-channel";
            fsl,qman-channel-id = <0x22>;
            phandle = <100>;
        };
        ...
```

11.1.3.3.7 Partition definition

A partition is defined by declaring a "partition node" in the configuration device tree. At a minimum a partition definition must define the following:

- one CPU
- one region of physical memory (a "guest physical memory area")
- the location where the guest device tree is to be loaded

Beyond those minimum requirements, the following may be defined:

- additional CPUs
- a label string (for use by partition management tools)
- additional memory regions
- assignment of physical I/O devices to the partition
- a specification of any images to be loaded by the hypervisor before starting the partition
- the assignment of virtual resources to the partition (byte-channels, doorbells endpoints)
- debug stub configuration
- node-update and node-update-phandle nodes that specify modifications to the guest device tree.

A partition node has a *compatible* value of "partition".

Example

Below the definition of a simple partition is shown. The partition contains two CPUs, one memory region, and one I/O device (a UART).

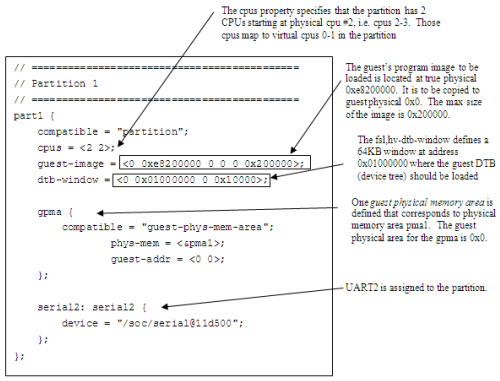


Figure 355.

11.1.3.3.7.1 Assigning CPUs to Partitions

Each physical CPU in the system may be assigned to a partition. The physical CPU assigned to a partition is enumerated logically in the guest device tree. In the guest device tree reg property and the unit address reflect the virtual CPUs index within the partition.

The example below defines two virtual cpus (virtual CPU 0 and 1) that map to physical CPUs 6 and 7.

Example:

```
cpus = <6 2>; // 2 cpus starting at CPU #6
```

On multi-threaded cores (e.g. e6500 dual-threaded core) the hardware threads are individually assigned to the partitions. The hypervisor will expose the vcpu to guests as a single threaded core. As a consequence, it's possible to have two partitions running on the same core but on distinct threads.

Example:

Given this hardware device tree snippet defining two dual-threaded e6500 cores:

```
PowerPC,e6500@0 {
        [...]
        reg = <0 1>;
        [...]
};

PowerPC,e6500@1 {
        [...]
        reg = <2 3>;
        [...]
};
```

this is how one assigns the first three hardware threads to a partition:

```
cpus = <0 3>;
```

The hypervisor will transparently expose three single-threaded e6500 cores to the partition.

11.1.3.3.7.2 Specifying the guest device tree address

The hypervisor creates a guest device tree in a partitions memory before the partition is started. The guest physical address where the DTB should be placed is specified in the *dtb-window* property. The format of the value of *dtb-window* is {guest physical address , size}.

Please refer to the *Embedded Hypervisor Software Reference Manual* [2] for additional details.

Example:

```
dtb-window = <0 0x01000000 0 0x10000>;
```

11.1.3.3.7.3 Specifying a label

A label property can be used to specify an identifying string to name the partition. This string will be displayed when managing the partition.

Example:

```
label = "linux-p2";
```

11.1.3.3.7.4 Loading Images

The hypervisor can load guest images into a partition prior to starting it. Three properties can be used to specify these images.

- **guest-image**—the *guest-image* property specifies the source address, destination address, and size of an image to be executed when a partition is started. The following image types are supported:

- ELF (Executable and Linking Format)—the entry point into the image is calculated from the offset between **e_entry** and the **p_vaddr** field of the program header that contains **e_entry**. If the destination address equals -1, the physical address specified in the ELF program headers (**p_paddr** field) will be used as the destination guest physical address.
- uImage (u-boot Image format)—the entry point into the image is calculated from the offset between the load address and entry point.
- Binary. For binary images, offset 0x0 in the image is treated as the entry point.

- **linux-rootfs**—the linux-rootfs property specifies the source/destination addresses and size of Linux root filesystem. The following image types are supported: 1) uImage (u-boot Image format), 2) binary.

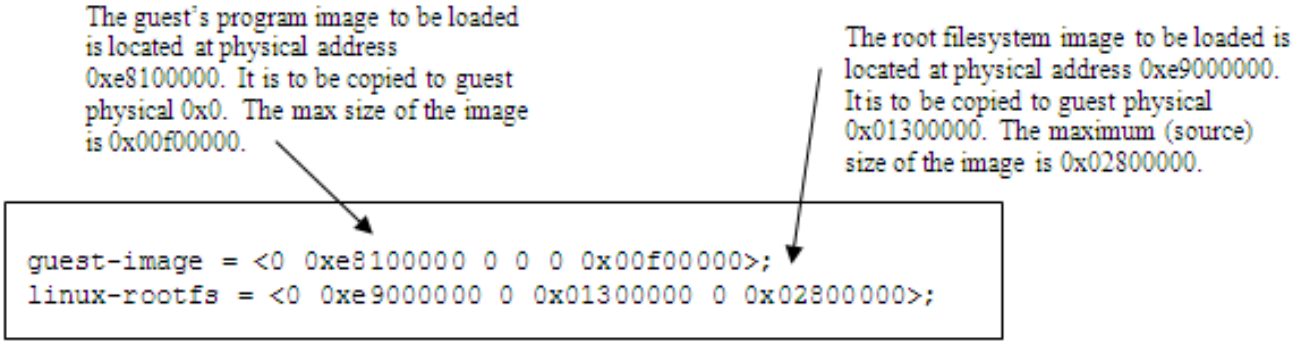
In the guest device tree for this partition, the *linux,initrd-start* and *linux,initrd-end* properties will be set in the /chosen node to reflect the start and end of the rootfs image as guest physical addresses.

- **load-image-table**—specifies the source/destination addresses and size of one or more supplementary images to be loaded by the hypervisor prior to partition start. The value is a prop-encoded array and is a table with one or more rows. The following image types are supported: 1) ELF (Executable and Linking Format), 2) uImage (u-boot Image format), 3) binary.

For ELF files, if the destination address equals -1, the physical address specified in the ELF program headers will be used as the destination guest physical address.

Source addresses are true physical. Destination addresses are guest physical. Please refer to the *Embedded Hypervisor Software Reference Manual* [2] for additional details.

Example



Note, it is not required that the hypervisor load all images. At least 1 "manager" partition must be loaded and started by the hypervisor. The manager partition can then load images from its filesystem and start the managed partition. See Managed Partitions on page 2291.

11.1.3.3.7.5 Guest cache lock mode

By default guest software is permitted to lock lines in the CPU and platform caches. "Guest cache lock mode" can be disabled by specifying the *guest-cache-lock-disable* property on the partition node. This prevents guest software running in the partition from normally executing cache locking instructions (**dcbtls, dcbtstls, dcblc, icbtls, icblc**).

Please refer to the *Embedded Hypervisor Software Reference Manual* [2] for additional details.

11.1.3.3.7.6 Guest debug mode

By default the CPUs debug resources (debug interrupt, IACx, DACx, DBCRx) are owned by the the partition. "Guest debug mode" can be disabled by specifying the guest-debug-disable property on the partition node.

Disabling guest debug mode is required in order to use hypervisor debug stubs for debugging guest operating systems (see Debug stub node on page 2294).

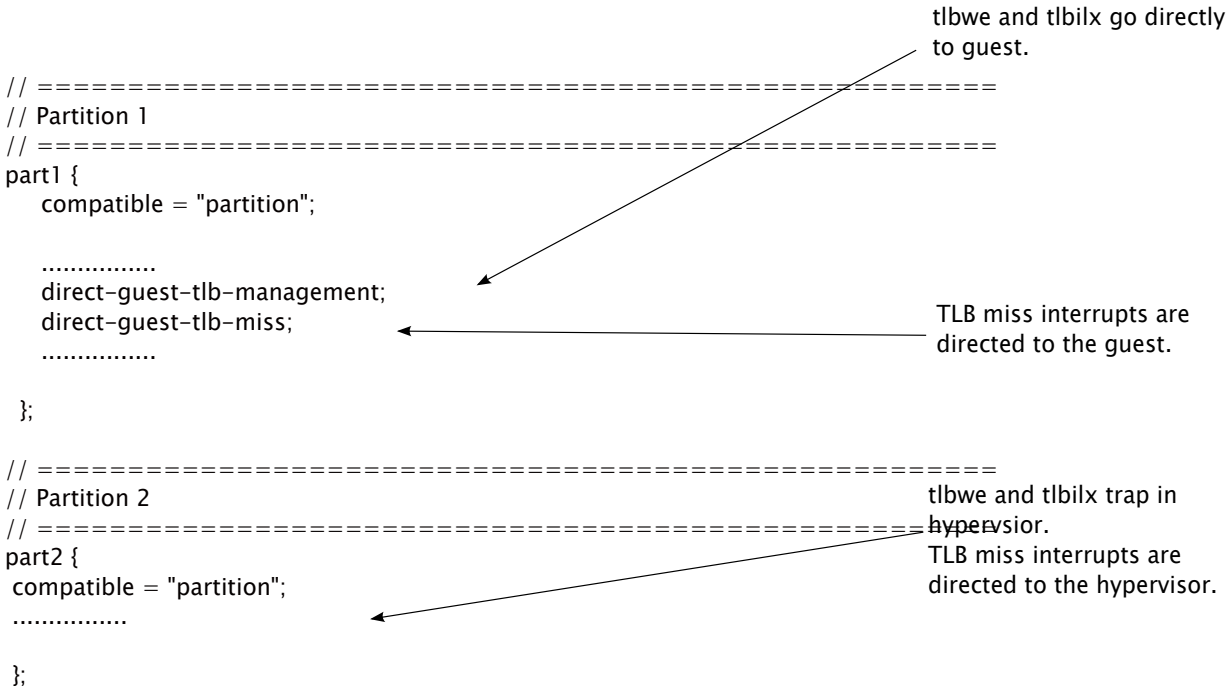
Please refer to the *Embedded Hypervisor Software Reference Manual* [2] for additional details.

11.1.3.3.7.7 Guest TLB Management

e6500 core allows guest TLB management instructions. Two partition configuration options can be used in various combinations:

- direct-guest-tlb-management
- direct-guest-tlb-miss

Please refer to *Embedded Hypervisor Software Reference Manual* [2] for additional details.



11.1.3.3.7.8 Assigning Memory to Partitions

Memory is granted to partitions using "guest physical memory areas" (or GPMAs). GPMAs are defined as child nodes of a partition node. Each GPMA references a PMA (see Physical Memory Areas on page 2267). The GPMA also specifies the guest physical address at which the memory region should be located. A GPMA appears in the guest device tree as a standard ePAPR-compliant memory node.

Memory can be shared between 2 partitions by each partition creating a GPMA that references the same PMA.

Please refer to the *Embedded Hypervisor Software Reference Manual* [2] for additional details.

Example 3-3, guest physical memory area

Hypervisor configuration tree:

```
        partition0 {
                ...
                gpma0 {
                        compatible = "guest-phys-mem-area";
                        pma = <&pma2>;  // 256MB region
                        guest-addr = <0 0>;
                };
                gpma1 {
                        compatible = "guest-phys-mem-area";
```

```
                        pma = <&pma8>;  // 64MB region
                        guest-addr = <0 0x64000000>;
                        node-update {
                            delete-prop = “device-type”;
                            prepend-stringlist = “compatible”,
                                            “fsl,shared-heap”;
                        };
                };
                ...
    };
```

Guest device tree as seen by partition 0:

```
    / {
            #address-cells = <2>;
            #size-cells = <2>;
            ...
            memory@0x0 {
                    device-type = "memory";
                    reg = <0x0 0x0 0x0 0x10000000>;
            };
            memory@0x64000000 {
                    compatible = “fsl,shared-heap”;
                    reg = <0x0 0x64000000 0x0 0x04000000>;
            };
            ...
    };
```

11.1.3.3.7.9 Assigning I/O devices to partitions

I/O devices are granted to partitions with a special "device node" which is a child of the partition node. Device nodes specify a path or alias (see the /aliases node in the ePAPR) in the hardware device tree to the I/O device being assigned.

The assigned device's hardware device tree nodes are copied under the node /devices in the guest device tree.

The hardware device tree expresses the true physical addresses of the system hardware. By default the hypervisor maintains a 1:1 (identity) mapping between true physical and guest physical addresses for all I/O devices.

There are 2 general types of devices represented by device nodes:

- a device that does no DMA (e.g. UART)
- DMA devices—a device that is capable of doing DMA (e.g. DMA controller). Every device that performs DMA expresses a logical I/O device number (LIODN) that the platform IOMMU (PAMU) uses to authorize the access. Certain devices which express multiple LIODNs have special characteristics in the configuration tree.

11.1.3.3.7.9.1 Standard Devices

Standard devices simply have a property that specifies the full path or alias to the node in the hardware device tree that represents the device being assigned to the partition.

Below the assignment of a device in the configuration tree is shown followed by what the node would look like in the guest device tree.

Partition configuration tree:

```
    partition0 {
        ...
        uart2 {
            device = "/soc/serial@11d500";   // serial2
```

```
        };
    };
```

Guest device tree:

```
    uart2 {
        cell-index = <0x2>;
        device_type = "serial";
        compatible = "ns16550";
        reg = <0 0xfe11d500 0 0x100>;
        clock-frequency = <0xe4e1c0>;
        interrupts = <0 0x2>;
        interrupt-parent = <0x564d5043>;
    };
```

11.1.3.3.7.9.2 DMA Devices

DMA devices require an additional property (*dma-window*) to specify the DMA window (see section) which is the guest physical memory regions which this device is allowed to access.

Example:

```
    fman0 {
        device = "fman0";
        dma-window = <&window0>;
     };
```

11.1.3.3.7.9.3 Queue Manager Portals

Qman portals are represented in the hardware device tree with a node containing a compatible value of "fsl,qman-portal." Qman portals express six different logical devices:

- Fman0
- Fman1
- SEC
- PME
- Dequeue data or context stashing
- Dequeue DQRR stashing

Qman portals are typically associated with physical CPUs, and thus a normal configuration would assign one Qman portal to a partition per physical CPU assigned to the partition.

There are 2 components involved in assigning Qman portal devices to a partition:

1. Definition of a *portal-devices* node.

   The portal-devices node configures the Fman0, Fman1, SEC, and PME logical devices. This single configuration is common for all portals within the partition.

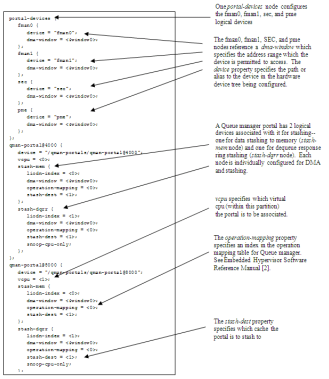
2. Qman portal device assignment node for each portal

Each Qman portal to be assigned is assigned using a device assignment node. In addition, the data/context stashing and DQRR stashing logical devices are configured in these nodes.

Please refer to the *Embedded Hypervisor Software Reference Manual* [2] for a complete definition of all properties related to Qman portal assignment. The example below illustrates how the properties are used.

Example 3-4

Example of the assignment of two queue manager portals to a two CPU partition:



11.1.3.3.7.9.4 Bus types supported

The hypervisor supports partitioning of I/O devices in the hardware device tree that reside on the busses with the following compatible types:

- "simple-bus"
- "fsl,dpaa"

Simple Bus. For device nodes where all ancestors contain a compatible value of "simple-bus", and where no ancestor has been assigned to the partition, the hypervisor copies the node from the hardware device tree node to the guest tree under the node */devices*. Any *reg* property which may have contained an address offset is updated to reflect the full physical address of the devices registers as mapped by any *ranges* properties in the ancestry of the node. Any *ranges* property and unit addresses are also updated to reflect the full physical address.

fsl,dpaa. For device nodes with a parent that contains a compatible value of "fsl,dpaa", both the node and parent are copied to the guest device tree under the node */devices*.

11.1.3.3.7.9.5 ranges

The *ranges* property in an ePAPR-compliant hardware device tree describes the address mapping of a memory-mapped bus. It provides a means of defining a mapping or translation between the physical address space of the bus (the child address space) and the physical address space of the bus node's parent (the parent address space). See the ePAPR for additional information.

When assigning a bus node to a partition, unless the node defines a *map-ranges* property, the hypervisor will not assign the guest physical address regions defined by the *ranges* property to the partition (unless they are also referenced by the *reg* property of the node)

See the example below.

Example

A PCI bus is being assigned to a partition. The PCI memory address space is visible to guest software at 0x80000000 and is 512MB in size (as per the hardware device tree). Fragments of a configuration device tree and hardware device tree are shown below.

Configuration device tree:

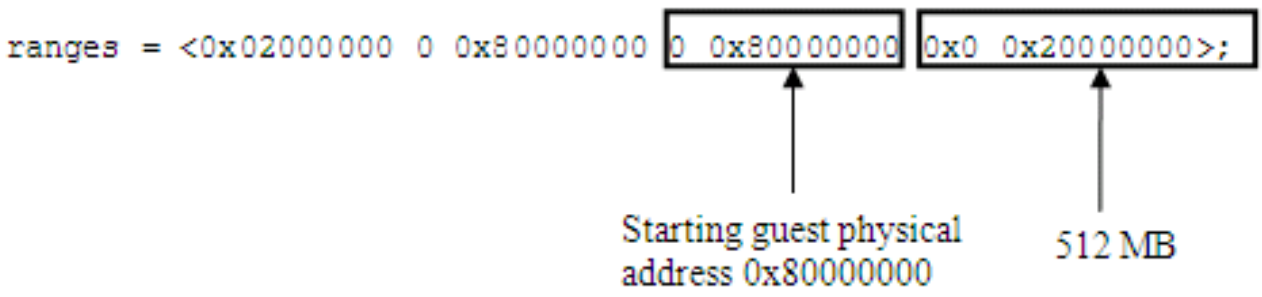
```
        pci-e {
            device = "/soc/pcie@fe200000";
```

```
            map-ranges;
        };
```

Hardware device tree:

```
        / {
                compatible = "fsl,hv-platform-p4080", "simple-bus";
                model = "fsl,hv-linux-p1";
                epapr-version = "ePAPR-1.0";
                #address-cells = <2>;
                #size-cells = <2>;

                pci0: pcie@fe200000 {
                        cell-index = <0>;
                        compatible = "fsl,p4080-pcie", "fsl,mpc8548-pcie";
                        device_type = "pci";
                        #interrupt-cells = <1>;
                        #size-cells = <2>;
                        #address-cells = <3>;
                        reg = <0 0xfe200000 0 0x1000>;
                        bus-range = <0x0 0xff>;
                        ranges = <0x02000000 0 0x80000000 0 0x80000000 0x0 0x20000000>;
                        clock-frequency = <33333333>;
                        interrupt-parent = <&mpic>;
                        interrupts = <24 2>;
        };
```

In the example above, the PCI bus is assigned to the partition with the "pci-e" node. The map-ranges property specifies that hypervisor should treat the defined address ranges in the parent bus's address space as part of the valid guest address space. So in the example, a 512MB region at guest physical 0x80000000 will be treated by the hypervisor as a valid guest physical address. See figure below.



11.1.3.3.7.9.6 Propagation to Descendants

Certain properties of device nodes are applicable to descendants of the device in the hardware device tree to which the device node refers. The *dma-window* property applies to all descendants of the device in the hardware device tree.

Example:

```
        fman0 {
            device = "/soc@fe000000/fman@400000";
            dma-window = <&window2>;
        };
```

In the above example, the dma-window property applies to all child nodes of fman0 that are capable of DMA.

This behavior can be overridden for descendants of the base device by specifying a separate device node for the child node.

Example:

Suppose you want a different dma window for the fman0_rx2 child node of fman0. A separate device node for fman0_rx2 overrides the dma window specified in fman0.

```
fman0 {
        device = "/soc@fe000000/fman@400000";
        dma-window = <&window2>;
};
fman0_rx2 {
        device = "/soc@fe000000/fman@400000/port@8a000";
        dma-window = <&window3>;
};
```

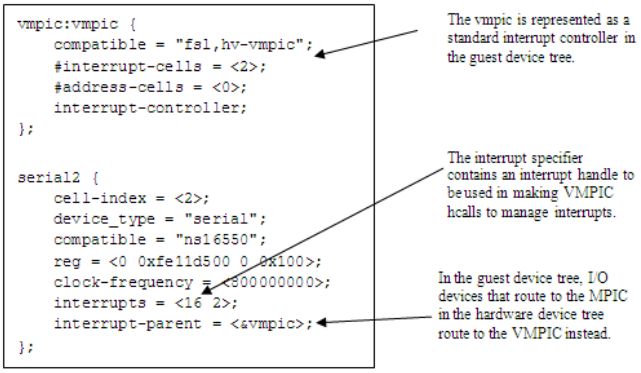
11.1.3.3.7.10 Interrupts and the VMPIC node

External interrupts are processed by guest software executing in a partition with a processing model similar to the hardware platform MPIC (Multiprocessor Programmable Interrupt Controller), with the exception that that MPIC hardware is not directly accessible by guest software. Instead, a virtual MPIC (VMPIC) interface provides interrupt controller services and is accessed via a hypercall interface. See the hypervisor reference manual [2] for a further detail about interrupt processing and the interrupt controller hypercall interface.

All hardware interrupts that route to the MPIC node (as described by the *interrupt-parent* property) in the hardware device tree will instead route to a VMPIC node in the guest device tree. Nodes with the VMPIC as an interrupt parent have interrupt specifiers that contain hypervisor- assigned interrupt numbers that are used as handles in VMPIC hcalls.

Example

Below is an example of a guest device tree that has a UART assigned to it.



Direct EOI—The hypervisor provides an optional mechanism by which in some circumstances an end-of-interrupt can be performed with no hypercall. See the mpic-direct-eoi property in the reference manual [2].

Message signaled interrupts—processing PCI Express MSIs has some special considerations. See the section on PCI Express and Message Signaled Interrupts in the reference manual [2].

NOTE

Hardware interrupts may not be shared between guest OSes or between the hypervisor and a guest. It is recommended that devices that share a hardware interrupt (e.g. UART0 and UART1) be assigned to the same partition.

11.1.3.3.7.11 Byte-channel configuration

Each partition created by the hypervisor can have virtual resources called byte-channels which provide a virtualized interrupt-driven character-based I/O channel connected to one of several endpoints:

- a physical UART
- a byte-channel in another partition
- a byte-channel multiplexer (mux
- the hypervisor console

Hypervisor GDB stubs also each define a byte-channel for communication. The node defining the configuration of the stub also specifies an endpoint to which the stub communicates.

Guest software interacts with byte-channels using a hypercall interface which allows the sending and receiving of character data. Multiplexing is supported where multiple (up to 32) byte-channels can operate over a single serial line. A de-multiplexing terminal server is required to de-multiplex the channels.

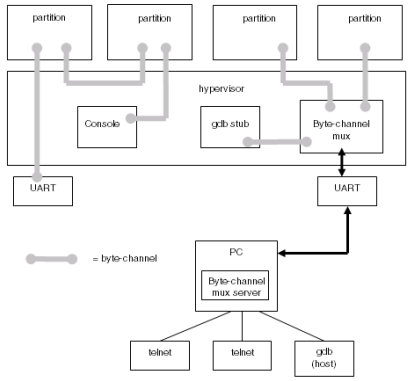


Figure 356. Byte-channel connection example

11.1.3.3.7.11.1 Configuration of Byte-Channels

A byte-channel endpoint is declared by defining a byte-channel endpoint node. This node is a child of the "partition" node. An endpoint property specifies a phandle to another node to which this byte-channel endpoint is attached. If the endpoint is a byte-channel mux, a mux-channel property specifies which channel the byte-channel is to use.

In the guest device tree the byte-channel will have a node under "/hypervisor/handles" which specifies the handle (for use in the hcall API) and interrupt numbers for the byte-channel.

Please refer to the *Embedded Hypervisor Software Reference Manual* [2] for details on how to use the byte-channel hcall API.

The examples below show excerpts from the configuration tree and guest device tree for several byte-channel configurations.

Example 3-5, byte-channel attached to UART

The example below defines a single byte-channel attached to UART1. The *endpoint* property in the partition node associates UART1 with the byte-channel. The guest device tree node contains handles for byte-channel hcalls and interrupt processing.

Hypervisor configuration tree:

```
        hv-config {
                compatible = "hv-config";
                ...
                uart1: uart1 {
                    device = "serial1";
                };
                ...
```

```
        };

        partition1 {
                byte-channel0 {
                    compatible = "byte-channel";
                    endpoint = <&uart1>;
                };
        };
```

Guest device tree:

```
        hypervisor {
            handles {
                byte-channel@6 {
                    compatible = "fsl,hv-byte-channel-handle";
                    reg = <6>;   // handle to byte-channel
                    interrupts = <10 0>; // handle to interrupt
                };
            };
        };
```

Example 3-6, byte-channel attached to byte-channel mux

The example below defines a byte-channel attached to a mux channel 3. The uartmux node associates UART1 with the mux. The byte-channel specifies the mux as the endpoint and a channel number of 3.

Hypervisor configuration tree:

```
        hv-config {
                compatible = "hv-config";
                ...
                uart1: uart1 {
                    device = "serial1";
                };
                ...
        };

        uartmux: uartmux {
                compatible = "byte-channel-mux";
                endpoint = <&uart1>;
        };

        partition1 {
                byte-channel0 {
                    compatible = "byte-channel";
                    endpoint = <&uartmux>;
                    mux-channel = <3>;
                };
        };
```

Guest device tree:

```
        hypervisor {
            handles {
                byte-channel@7 {
```

```
                compatible = "fsl,hv-byte-channel-handle";
                reg = <7>;   // handle to byte-channel
                interrupts = <11 2>;  // handle to interrupt
            };
        };
    };
```

Example 3-7, byte-channel declared for use between 2 partitions

The example below defines two partitions, each with an endpoint connected through the same byte-channel.

Hypervisor configuration tree:

```
    partition1 {
        p1-bc0:byte-channel0 {
            compatible = "byte-channel";
            endpoint = <&p2-bc0>;
        };
    };

    partition2 {
        p2-bc0:byte-channel0 {
            compatible = "byte-channel";
            endpoint = <&p1-bc0>;
        };
    };
```

Guest device tree for part1:

```
    hypervisor {
        handles {
            byte-channel@8 {
                compatible = "fsl,hv-byte-channel-handle";
                reg = <8>;   // handle to byte-channel
                interrupts = <12 2>;
            };
        };
    };
```

Guest device tree for part2:

```
    hypervisor {
        handles {
            byte-channel@8 {
                compatible = "fsl,hv-byte-channel-handle";
                reg = <8>;   // handle to byte-channel
                interrupts = <15 2>;
            };
        };
    };
```

11.1.3.3.7.12 Inter-partition Doorbell Configuration

To define a doorbell endpoint within a partition, a node is created that represents the endpoint. The type of endpoint (send or receive) is specified with the *compatible* property value of "send-doorbell" or "receive-doorbell". A phandle to the global doorbell node is specified with the *global-doorbell* property.

Example 3-8, doorbell with 2 send endpoints (partitions 2 and 3) and 1 receive endpoint (partition 1)

Hypervisor configuration tree:

```
        doorbells {
                dbell0:doorbell0 {
                    compatible = "doorbell";
                };
        };

        partition1 {
                ...
                doorbell0 {
                    compatible = "receive-doorbell";
                    global-doorbell = <&dbell0>;
                    node-update  {
                        prepend-stringlist = “compatible”, “no-mem”;
                    };
                };
                ...
        };
        partition2 {
                ...
                doorbell0 {
                    compatible = "send-doorbell";
                    global-doorbell = <&dbell0>;
                    node-update  {
                        prepend-stringlist = “compatible”, “no-mem”;
                    };
                };
                ...
        };
        partition3 {
                ...
                doorbell0 {
                    compatible = "send-doorbell";
                    global-doorbell = <&dbell0>;
                    node-update  {
                        prepend-stringlist = “compatible”, “no-mem”;
                    };
                };
                ...
        };
```

The doorbells defined in the above example would have the following representation in guest device trees.

Guest device tree for partition1:

```
        handles {
            doorbell@8 {
                compatible = “no-mem”,
                             "fsl,hv-doorbell-receive-handle";
```

```
                interrupts = <12 0>;
        };
};
```

Guest device tree for partition 2:

```
handles {
    doorbell@9 {
        compatible = “no-mem”, "fsl,hv-doorbell-send-handle";
        reg = <9>;   // handle to doorbell

    };
};
```

Guest device tree for partition 3:

```
handles {
    doorbell@8 {
        compatible = “no-mem”, "fsl,hv-doorbell-send-handle";
        reg = <8>;   // handle to doorbell
    };
};
```

11.1.3.3.7.13 Managed Partitions

The hypervisor provides the ability for one partition to manage other partitions. Management involves the capability to:

- Start, stop, restart managed partitions (via hypercall APIs)
- Receive doorbell interrupts from managed partitions for certain events
  - watchdog expiration
  - managed partition state change
    - transition from starting to running
    - transition from stopping to stopped
    - transition from pausing to paused
    - transition from resuming to running
  - restart request
- Send a doorbell interrupt to a managed partition to request that it cleanly shut down

Management ability is granted to the 'manager' partition via a declaration in the manager's partition node. See the reference manual [2] for an overview of partition management and the available partition management hypercalls and interrupts.

To specify the manager/managed relationship between two partitions a partition management node is declared in the partition node of the manager partition. This node contains a partition property which has a value which is a phandle to the managed partition.

Configuration Node Example

The "p1-linux" partition is defined to be the manager of "part2" which is referenced by phandle.

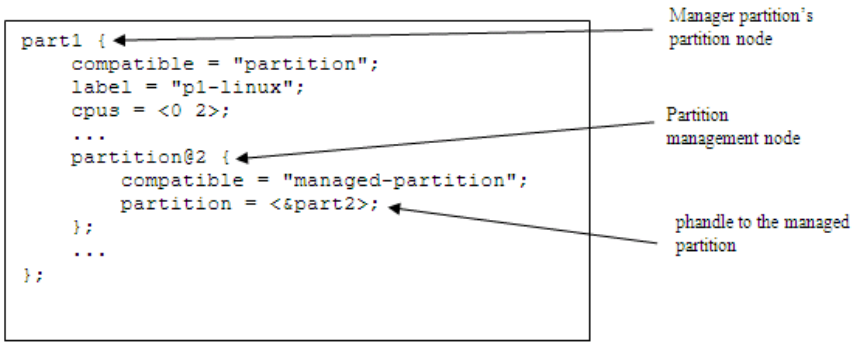


Figure 357. partition node for managing another partition

Guest Device Tree Examples

For the configuration node example in 3-9, the following is what the guest device tree would look like for the manager and managed partition:

Manager partition:

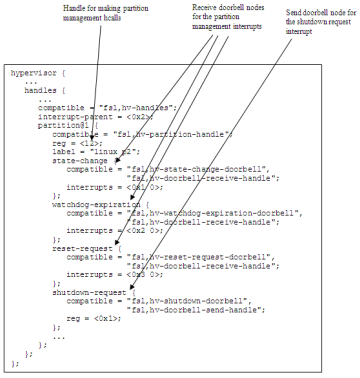


Figure 358. manager partition guest device tree

Managed partition:

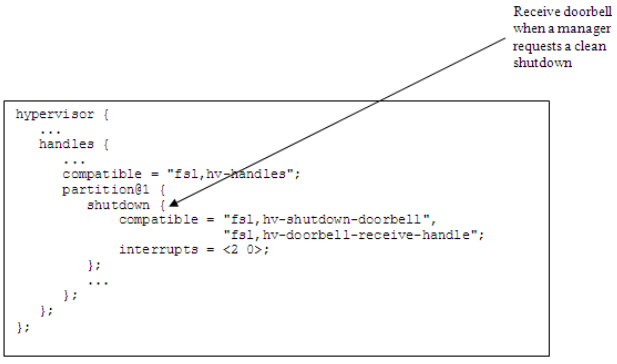


Figure 359. managed partition guest device tree

11.1.3.3.7.14 Guest Aliases

By default all devices assigned to a partition that have an alias in the hardware device tree have the alias copied to the guest device tree. They are copied verbatim, but are updated to correctly reflect the structure of the guest device tree which may be somewhat different than the hardware tree.

In addition, a guest aliases node can be specified under a partition node to add additional aliases to the guest device tree. The name of this node must be *aliases*. The value must be a phandle to a node in the hypervisor configuration tree. The phandle values are resolved to a string value that is the full path to the referenced object in the guest device tree.

Example:

Hypervisor configuration tree:

```
aliases {
    stdout = <&bc2>;
};
```

Guest device tree:

```
aliases {
    ...
    stdout = "/hypervisor/handles/byte-channel2";
};
```

11.1.3.3.7.15 Guest root node properties

The partition node supports a node-update node (see Guest device tree node updates on page 2275) which updates the root node of the guest device tree.

Example:

Hypervisor configuration tree fragment:

```
partition2 {
    ...
    node-update {  // update to the root node
        model = "P4080-Linux";
        chosen {
            foo = "bar";
        };
    };
    ...
};
```

Guest device tree:

```
/ {
    model = "P4080-Linux";
    ...
    chosen {
        foo = "bar";
    };
```

```
        ...
    };
```

11.1.3.3.7.16 Debug stub node

The hypervisor supports debugging guest software using debug stubs within the hypervisor. A host debugger such as GDB can attach to hypervisor debug stubs via a byte-channel. Each virtual CPU being debugged has an instance of a debug stub which performs run control and accesses guest hardware state on behalf of the host debugger.

The hypervisor supports stubs for the GDB and CodeWarrior debuggers.

The hypervisor cannot share the physical CPU's debug resources with the guest operating system. In order to allow debug stubs to use the CPU's debug resources the *guest-debug-disable* property must be specified on the partition node which defines a debug stub.

In order to enable a debug stub a special "debug stub" node must be added to the partition node which specifies the byte-channel and virtual CPU the stub is to debug.

- The compatible string for the node must be "gdb-stub" (for GDB) or "trk-stub" (for CodeWarrior).
- The *endpoint* property specifies a phandle to the character device node which this stub is to use for communication. This value must be a phandle to a hypervisor owned character device (e.g. UART) or a byte-channel mux. In the case of a byte-channel mux the channel must also be specified.
- The *debug-cpus* property specifies which virtual CPUs are managed by this stub instance

See the reference manual [2] for full details.

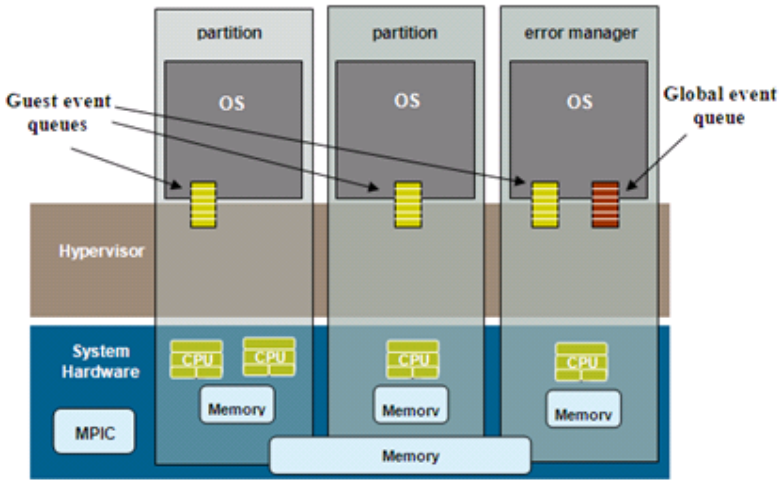
Example

Definition for two gdb stubs debugging a 2 CPU partition—one for debugging virtual cpu 0 (on mux channel 1) and one for debugging virtual cpu 1 (on mux channel 2):

```
    uartmux: uartmux {
        compatible = "byte-channel-mux";
        endpoint = <&uart1>;
    };
    ...
    partition {
        compatible = "partition";
        guest-debug-disable;
        cpus = <0 2>;
        ...
        gdb-stub@0 {
            compatible = "gdb-stub";
            endpoint = <&uartmux>;
            mux-channel = <1>;
            debug-cpus = <0 1>;  //virtual cpu 0
        };
        gdb-stub@1 {
            compatible = "gdb-stub";
            endpoint = <&uartmux>;
            mux-channel = <2>;
            debug-cpus = <1 1>;  // virtual cpu 1
        };
    };
```

11.1.3.3.7.17 Error management

As described in the NXP Embedded Hypervisor Reference manual [2], the hypervisor provides a set of event queues that are used to communicate certain error conditions to partitions.



Guest Event Queue

Each partition has a guest event queue. Partitions will receive notifications of PAMU access violations for a device owned by the partition. Partitions are notified of errors in the guest event queue via a machine check (MCSR[MCP], IVOR1) on virtual CPU #0.

The FH_ERR_GET_INFO hypercall API is used to get the error event information from the queue.

Global Event Queue

There is a single global event queue that is used to communicate certain system errors to a partition designated to be an error manager. The partition designated to be the error manager is specified in the definition of the partition in the hypervisor configuration tree. The notification policy for each system error can be also specified in the hypervisor configuration tree.

The error manager is notified of errors in the global event queue via a critical interrupt (IVOR0). The CPU in the partition to receive the interrupt can be specified in the configuration tree.

The FH_ERR_GET_INFO hypercall API is used to get the error event information from the queue.

11.1.3.3.7.17.1 Global Event Queue Configuration Example

In the hypervisor configuration tree fragment below, *partition1* is designated to be the error manager and the policies for DDR single bit ECC errors are configured.

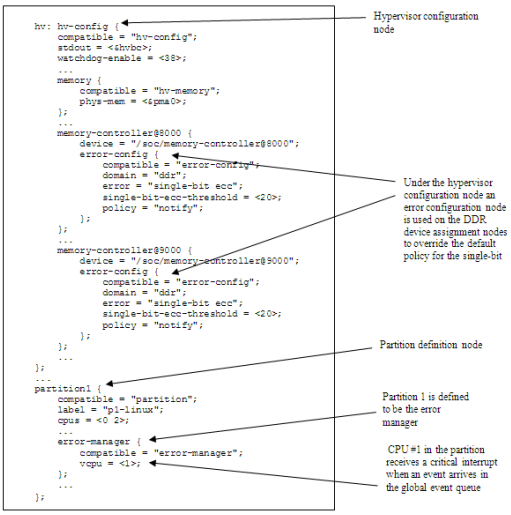


Figure 360. Configuration of Global Event Queue and Error Policies

11.1.3.3.7.17.2 Example - Event Queues in a Guest Device Tree

The global event queue and guest event queue are represented in the guest device tree as shown in the guest device tree fragment shown below :

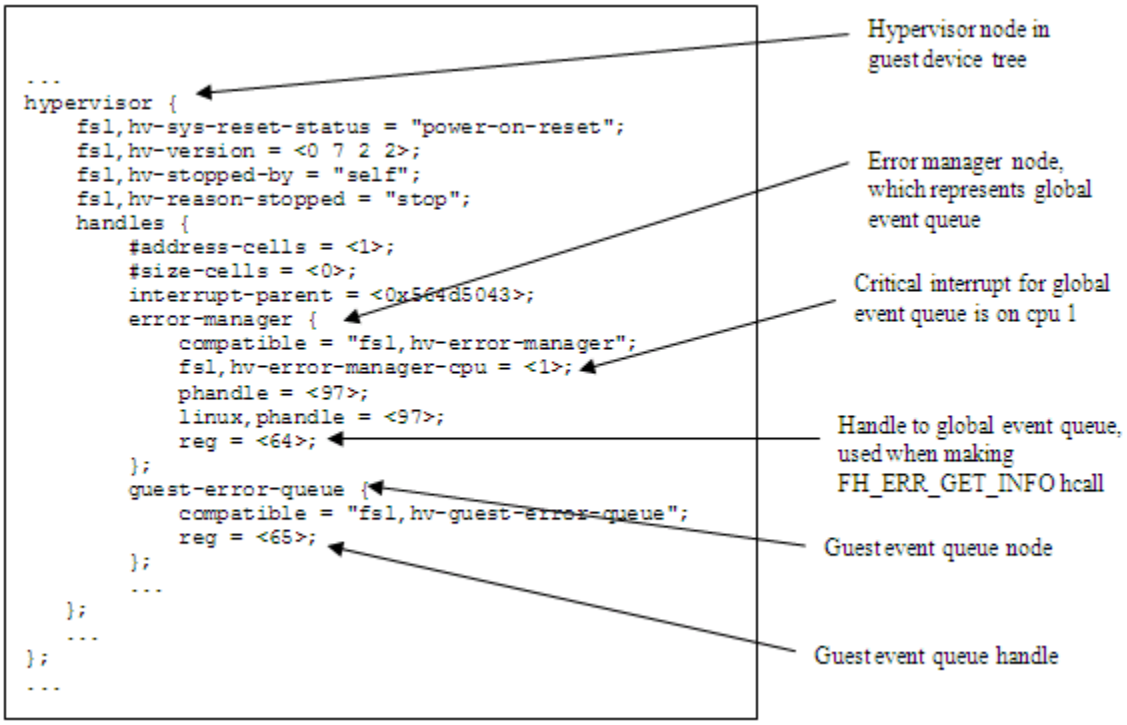


Figure 361. Event Queues in Guest Device Trees

11.1.3.3.7.18 High availability services

The hypervisor provides mechanism to create configurations where partitions can be arranged in an active and standby relationship for high availability. These mechanisms include:

- Notifications on managed partition state changes and watchdog expiration
- Device sharing between active and standby partitions. After an active partition crash, mechanisms exist for the new active partition to claim active ownership of shared devices, include interrupt routing re-configuration and IOMMU updates
- Error manager sharing between active and standby
- System reset if all partitions move into the stopped state (see sysreset-on-partition-stop property).

11.1.3.3.7.18.1 Partition notifications

Partitions can manage receive notification about other partitions using the partition management capabilities of the hypervisor. See Managed Partitions on page 2291 , and the hypervisor reference manual [2] for additional details on partition management features and the partition notifications that may be received.

In order for an active and standby partition to receive notifications they must be configured to manage each other, as shown below.

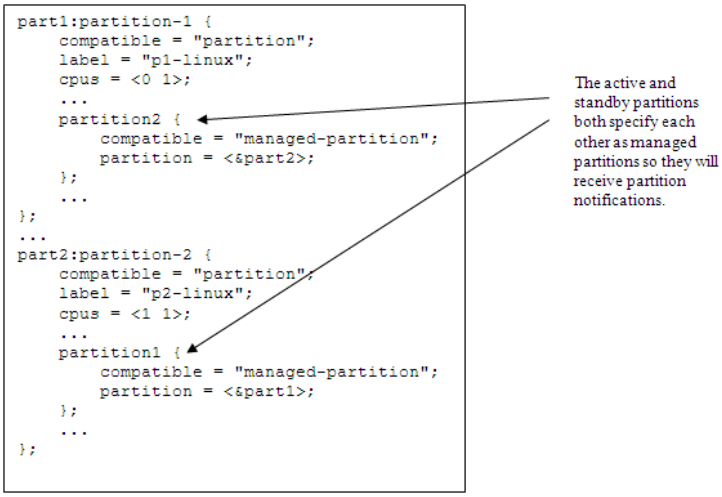


Figure 362. Active/standby partitions and notifications

11.1.3.3.7.18.2 Device Sharing

Devices can be shared between partitions in a failover configuration, whereby one partition has active ownership of a device at a time. Interrupts from the shared device are routed to the active owner. Device errors such PAMU access violations are placed in the error event queue of the active owner of the device.

If the active owner is stopped, another partition configured with access to the device may claim it and become the new active owner. To use this feature, all partitions assigned the shared device must specify the *claimable* property on the device configuration node for the device. At least one partition must have a shared device initially set to be "active". See the hypervisor reference manual [2] for details.

Hypervisor Configuration Tree

The hypervisor configuration tree describes the assignment of devices and which are active and which are standby. In the example below, an SoC-based DMA engine is assigned and shared by two partitions.

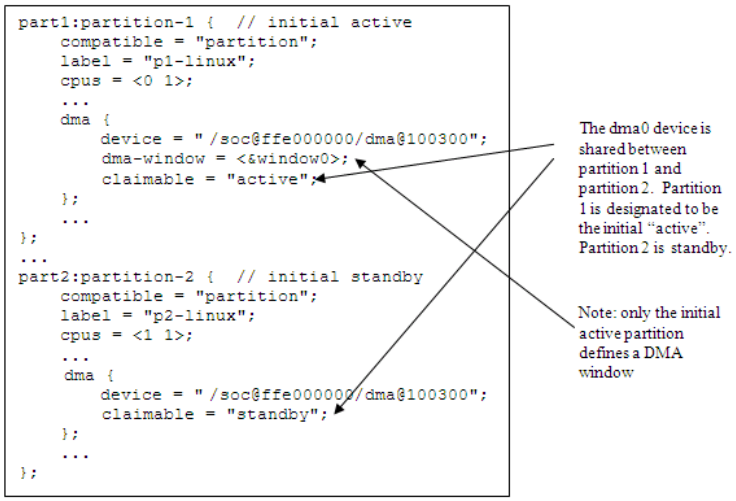


Figure 363. Active/standby Partitions and Device Sharing

Guest Device Trees

The guest device tree passed to the partitions sharing the device reflects the active standby status of a shared device in the node describing the device. The example below shows the guest device trees for a shared DMA engine.

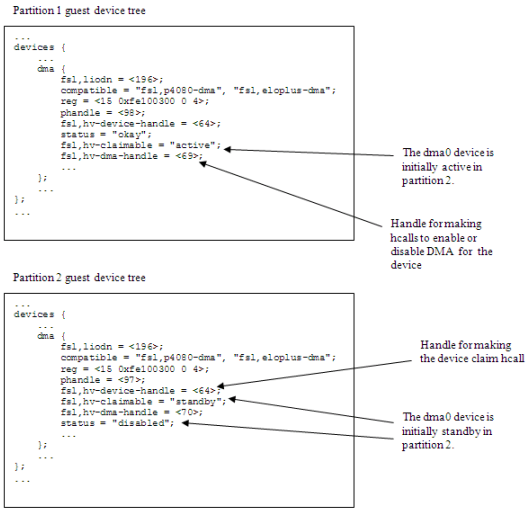


Figure 364. Guest Device Trees and Device Sharing

The partition with the standby device is permitted to map the device's registers, however it must take care to not modify any hardware state as that would potentially interfere with the active partition.

11.1.3.3.7.18.3 Sharing the Error Manager

One partition in the system is designated to be the error manager. This partition receives notification interrupts when there is a system error and has a global event queue containing detailed error information.

The error manager can be shared between active and standby partitions in a manner identical to devices as described in the previous section.

11.1.3.3.7.18.4 Failover Event Flow Example

The example below illustrates a hypothetical event flow for two partitions in an active/standby configuration. The partitions share a device, the SoC-based DMA engine. Partition 1 is the initial active partition with Partition 2 being standby.

Table 577. Failover Event Flow Example

Partition 1	Partition 2
In Partition Definition • Partition 2 is specified as a managed partition • DMA device is assigned with claimable = "active" • Error manager node is specified with claimable = "active" • Partition's watchdog is configured to stop partition on watchdog timeout.	**In Partition Definition** • Partition 2 is specified as a managed partition • DMA device is assigned with claimable = "standby" • Error manager node is specified with claimable = "standby"
Active Partition OS DMA engine driver initializes and manages device normally.	**Standby Partition** OS DMA engine driver sees that device is "standby" and takes care not to touch device registers.
Failover Event Watchdog expires, resulting in partition 1 moving into the “stopped” state. This turns off DMA and disables device interrupts. state change notification interrupt ---------->	**Failover Event** Standby partition receives notification of active partition stopping.
	Failover • Invokes FH_CLAIM_DEVICE to claim active ownership of DMA engine device. • Invokes FH_CLAIM_DEVICE to claim error manager • Invokes interrupt controller APIs to configure and enable interrupts. • Invokes FH_DMA_ENABLE to enable DMA for the DMA engine device • Invokes FH_PARTITION_START to start partition 1
Partition restarts and becomes new standby.	Partition becomes new active.

11.1.3.3.8 Hypervisor Configuration Node

A hypervisor configuration node at the root of the configuration device tree specifies global configuration parameters and the memory and I/O devices assigned to the hypervisor.

- The *compatible* string for the node must be "hv-config"
- A hypervisor memory node (*compatible* = "hv-memory") must be specified which identifies the hypervisor private memory (the PMAs belonging to the hypervisor). The PMA assigned to the hypervisor must provide at least 8MB of memory.
- The *stdout* property specifies the console for the hypervisor. This can refer to a hardware serial port or a byte-channel. This property is only required to override any *stdout* alias specified in the hardware device tree.

See the hypervisor reference manual [2] for additional details.

All I/O devices that the hypervisor manages must be explicitly assigned under the hypervisor in the configuration node.

To have the hypervisor use a UART for a console and/or attached to a byte-channel mux a UART device (compatible = "ns16550") must be assigned to the hypervisor.

NOTE

The following devices must be assigned to the hypervisor for proper operation:

- Interrupt controller (compatible "chrp,open-pic")
- Corenet local access windows (compatible "fsl,corenet-law")
- Corenet coherency fabric (compatible "fsl,corenet-cf")
- Memory controllers (e.g. compatible ""fsl,p4080-memory-controller")
- Platform cache controllers (e.g. compatible "fsl,p4080-l3-cache-controller")
- IOMMU (e.g. compatible "fsl,p4080-pamu")
- Global utilities (compatible "fsl,qoriq-device-config-1.0")
- Specific compatible values may vary depending on the SoC being used.

Example:

```
hv:hypervisor-config {
        compatible = “hv-config”;
        stdout = <&serial0>;

        hv-memory {
            compatible = "hv-memory";
            phys-mem = <&pma0>;
        };

        pamu {
            device = "/soc@fe000000/iommu@20000";
        };

        mpic {
            device = "/soc@fe000000/pic@40000";
        };

        cpc-0 {
            device = "/soc/l3-cache-controller@10000";
        };

        iommu {
            device = "/soc/iommu";
        ;

        cpc-1 {
            device = "/soc/l3-cache-controller@11000";
        };

        corenet-law {
            device = "/soc/corenet-law";
        };

        corenet-cf {
            device = "/soc/corenet-cf";
```

```
                };

                guts {
                    device = "/soc/global-utilities@e0000";
                };

                serial0 {
                    device = "serial0";
                };
        };
```

11.1.3.4 Linux

Two device drivers exist in Linux to support embedded hypervisor functions:

1. A console driver that enables byte-channel based console I/O.
2. A management driver console driver that enables partition management capabilities that can be used by tools like partman.

11.1.3.4.1 Byte-channel Console Driver

The NXP hypervisor byte-channel console driver is part of the **hvc_console** subsystem in Linux.

In order to specify a byte-channel for the Linux console, the *stdout* alias must be set in the configuration device tree to point to a byte-channel endpoint node. Configuring this alias results in the Linux device **/dev/console** being attached to the specified byte-channel.

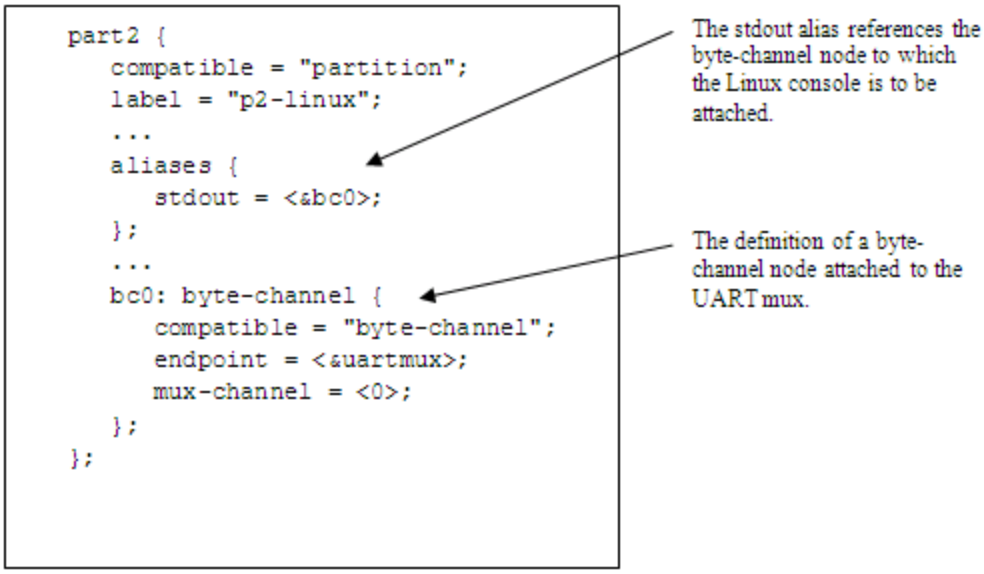


Figure 365. stdout alias for Linux byte-channel console

11.1.3.4.2 Management Driver

The NXP partman utility (described in Partition Management Utility (Linux) on page 2303), is a Linux-based tool for performing partition management tasks. It uses a specialized Linux device driver called **/dev/fsl-hv.**

The management driver provides a character device interface to perform partition management tasks—status, starting, stopping, and image loading for managed partitions. In addition, the partition management receive doorbell interrupts are handled by the driver

The management driver registers itself with the Linux sysfs subsystem. A **/dev/fsl-hv** file must be created with the appropriate major and minor numbers as returned by sysfs. It is recommended that the Linux "udev"or "mdev" facility be used to do this.

Example: To see the major/minor device numbers of fsl-hv:

```
cat /sys/class/misc/fsl-hv/dev
10:63
```

11.1.3.5 Debugging

11.1.3.5.1 Hypervisor Logging

Visibility into hypervisor activity is available through log information written to the hypervisor console. The log level used by the hypervisor is a compile time option and can be set through the hypervisor's Kconfig configuration utility (see Changing the Hypervisor Build Options on page 2308) and the hypervisor shell (see section below).

Log level verbosity can be set with the '*Default console loglevel*' option, with the value zero being no logging and 15 being the most verbose.

In addition, to reduce code size, logging code can be removed from the hypervisor by setting the '*Maximum console loglevel to build for*' option.

11.1.3.5.2 Hypervisor Command Shell

The hypervisor console provides an optional command shell which may be used for debugging. At the **HV**> prompt commands may be entered. See the hypervisor reference manual for details on shell command syntax. The shell also has built-in help to list and describe the supported commands.

Example 5-1

```
HV>
HV> ?
    loglevel (ll) - Control the log level for a specific log type
    crash - Crash the hypervisor.  Used for testing.
    buildconfig (bc) - Display build time configuration
    error_policy - Display error policies
    guestmem (gm) - Dump guest memory
    gtlb - Display guest tlb entries
    resume - Resume a paused partition
    pause - Pause a running partition
    stop - Stop a partition
    restart - Re-start a running partition
    start - Start a stopped partition
    paact - Display PAMU's PAACT table entries
    cdt - Display hypervisor configuration tree
    hdt - Display hardware device tree
    gdt - Guest device tree operation
    reset - Perform a system reset
    info - List partitions and show their status
    version - Print the hypervisor version
    help (?) - Print command usage information
```

```
HV>
```

To display partition info and status:

Example 5-2

```
HV> info
Partition   Name        State         Vcpus
-------------------------------------------
0           /part1      running         3
1           /part2      running         2
```

To display guest TLB entries:

Example 5-3

```
HV> pause 0
HV> gtlb 0 0
TLB 0
                                                          T             SSS UUU I V
Effective                Physical                         S TID   WIMGE XWR XWR P F
------------------------ -------------------------        - ----- ----- --- --- - -


TLB 1
                                                          T             SSS UUU I V
Effective                Physical                         S TID   WIMGE XWR XWR P F
------------------------ -------------------------        - ----- ----- --- --- - -
03  0x40000000 - 0x40000fff 0x020000000 - 0x020000fff    0     0       XWR     1 0
04  0x00100000 - 0x00100fff 0xffe100000 - 0xffe100fff    0     0  I G  XWR     1 0
15  0x20000000 - 0x2fffffff 0x000000000 - 0x00fffffff    0     0   M   XWR     1 0
```

To display the PAMU PAACT table:

Example 5-4

```
HV> paact
        sub                                     sub     stash
        win     base        xlate               win     cache   OMT
liodn   #       addr        addr    size        cnt     id      mode
-------------------------------------------------------------------
196     -       00000000  00000000  1024 MB     0015    000       0
        1       00000000  04000000  0016 MB     -       000       0
        2       01000000  0c000000  0016 MB     -       000       0
        3       02000000  20000000  0064 MB     -       000       0
197     -       04000000  64000000  0064 MB     0000    000       0
```

11.1.3.6 Tools

11.1.3.6.1 Partition Management Utility (Linux)

The hypervisor provides a Linux-based partition management utility called *partman* that works in conjunction with a partition management Linux kernel device driver to provide basic partition management services for a manager partition.

A partition is managed via a handle that the manager partition discovers under the /hypervisor/handles node in its device tree.

A summary of valid combinations of arguments to partman is show below:

Action	Syntax
Show status of a partition	partman [-v] status
Load images into a partition	partman [-v] load -h handle -f file [-a address] [-r]
Start a partition	partman [-v] start -h handle [-f file] [-l] [-e address] [-a address]
Stop a partition	partman [-v] stop -h handle
Listen for doorbells	partman [-v] doorbell -f file
Rings the doorbell of the destination partition	partman [-v] doorbell -h handle
Sets a guest device tree property	partman [-v] setprop -h handle -p dtpath -n propname [-t data [-t data]]
Gets a guest device tree property	partman [-v] getprop -h handle -p dtpath -n propname

Command	Description
`status`	Display the full names and status of all managed partitions. Also shows the handle values to be used for other partman commands.
`load`	Load an image into a partition's memory
`start`	Start a partition. Optionally load a file with -f, or use -l to tell the hypervisor to reload hypervisor-loaded images
`stop`	Stop a partition.
`doorbell`	Listen for doorbells (if -f specified), or ring a doorbell (if -h specified).
`setprop`	Sets a guest device tree property for the specified partition
`getprop`	Gets a guest device tree property for the specified partition.

Option	Description
`-h handle`	For the doorbell command, this specifies the send handle of the doorbell to ring. For all other commands, this specifies the handle of the target partition. Note, the handles used by this command are not the same as the partition numbers used by the hypervisor command shell. Use the status command to list the handles. The partition name call also be used in place of a handle value.

Table continues on the next page...

Table continued from the previous page...

`-f filename`	For the doorbell command, this specifies the program to run when a doorbell is received. This can be any program that can be executed by the shell. partman passes the doorbell handle as a parameter to this script. For all other commands, this specifies the name of the file to load. Three types of images are supported-1) ELF, 2) uImage (u-boot format), 3) binary. If this file is an ELF image, the ELF headers will be parsed and PT_LOAD segments will be extracted and copied to the target partition in the right order. If it is an uncompressed uImage file, the payload of the uImage is extracted and copied to the target partition. Otherwise, the file is treated as a single binary blob.
`-l`	Reload images. Used with the start command to direct the hypervisor to reload hypervisor-loaded images.
`-a address`	Optional guest physical address where the image should go. For ELF images, this parameter defaults to the base physical address in the image. For binary images, the default is 0.
`-e address`	Optional guest physical address of execution entry point. For ELF images, this parameter defaults to the entry point in the image. For binary images, the default is 0.
`-t data`	A string that is that value to write to the specified property using the setprop command. Integer and other binary values are not supported.
`-p dtpath`	Device tree path. A string that specifies the full path to the node for which the property is being accessed.
`-n propname`	Property name. A string that specifies the name of the property which is being accessed.
`-v`	Verbose output. Useful for debugging.
`-q`	Quiet mode. Errors reported via return status only.
`-r`	Specifies that the image is a root filesystem. It is used with the load command to pass the correct size information to the kernel. This parameter is optional.

Example

The examples below show a sequence of loading 2 images into a managed partition, starting the partition, then stopping it.

To see the status of partitions:

```
# partman status
Partition Name                  Handle  Status
----------------------------------------------
p3-linux                        71      stopped
p2-linux                        66      stopped

Byte Channel Name               Handle  RX Int  TX Int
------------------------------------------------------
bc-p3                           85      83      84
bc-p2                           82      80      81
console                         79      77      78

Doorbell Name                   Handle  Type
--------------------------------------------
shutdown                        76      receive
p3-linux/reset-request          74      receive
p3-linux/watchdog-expiration    73      receive
p3-linux/state-change           72      receive
p2-linux/reset-request          69      receive
p2-linux/watchdog-expiration    68      receive
p2-linux/state-change           67      receive
doorbell-in                     65      receive
p3-linux/shutdown               75      send
p2-linux/shutdown               70      send
doorbell-out                    64      send
```

To load a root filesystem image in partition "p2-linux" at guest physical address 0x1300000 in the p2-linux managed partition:

```
# partman load -h p2-linux -f rootfs.ext2.gz -a 0x1300000 -r
```

To load an image in partition "p2-linux" at guest physical 0x0:

```
# partman load -h p2-linux -f vmlinux -a 0x0
```

To start partition "p2-linux" at entry point of guest physical address 0x0:

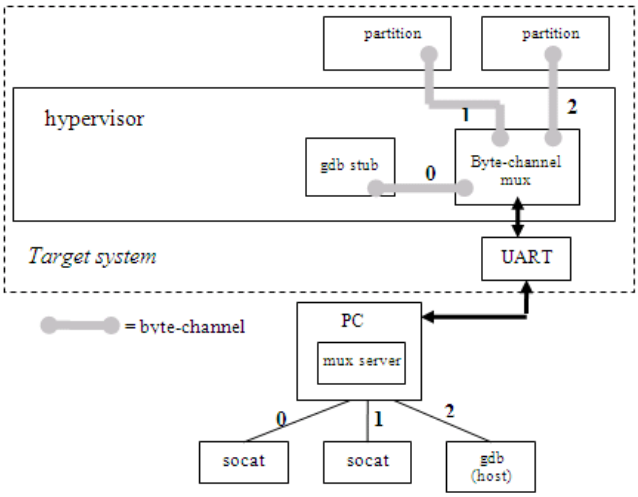
```
# partman start -h p2-linux -e 0x0
```

To stop partition "p2-linux":

```
partman stop -h p2-linux
```

11.1.3.6.2 Mux-server Utility (Linux)

The hypervisor supports multiplexed serial communications via byte-channels. In the configuration device tree each byte-channel that is to be multiplexed specifies a channel number in the definition of the byte-channel end point (see Byte-channel configuration on page 2287). The Linux-based mux_server utility is provided to support multiplexing and demultiplexing services. The mux server runs on a host system and attaches to the target system via a serial or network communications channel. it can be located in the standard Yocto output directory, subdirectory 'hv'.



Syntax: mux_server [-s <baud-rate>] <target> <channel_port> ...

Where:

<**baud-rate**> is the baud rate (default is 115200)

<**target**> is connection to the target board that the mux_server is providing mux/de-mux services to. This would typically be a serial port (e.g. /dev/ttys0) or a host:port in the case of network based connection to the target.

<**channel_port**> is a list of network ports corresponding mux channel numbers, starting with channel 0. So, the first port listed is channel 0, second port is channel 1, etc.

For mux channels that service console output, a variety of tools exist that provide an interactive display of the console data:

- socat (in raw mode)
- putty (in raw mode)

Examples

To attach to a target at /dev/ttys0 and export mux channel 0 on port 8000 and mux channel 1 on port 8001:

```
mux_server /dev/ttys0 8000 8001
```

For this example, to connect to the mux_server with an interactive console on mux channel 0 on host "falcor"

```
socat -,raw,echo=0 tcp:falcor:8000
```

11.1.3.6.3 Switching mux channels without a mux server

The hypervisor and mux_server use a simple channel switch protocol to multiplex the streams of data. ASCII 0x18 (CTRL-x) indicates a channel switch and is followed by a byte specifying the channel number-0x30 indicates channel 0, 0x31 indicates channel 1, etc. See the hypervisor reference manual [2] for a detailed description of the mux protocol.

The channel switch commands allow easy channel switching with only a single console session and no mux server using the CTRL-x character. The character sequence CTRL-x 0 selects channel 0. CTRL-x 1 selects channel 1, and so on.

11.1.3.7 Building and deploying the embedded hypervisor

11.1.3.7.1 Building the Hypervisor with Yocto

The embedded hypervisor software is packaged as part of a NXP SDK which provides the necessary tools needed to build the software. In the context of the *Yocto* package creator, the hypervisor can be built in a standalone manner with the following commands:

```
cd [path-to-yocto-poky-build-folder]
bitbake hypervisor
```

The above commands assume that *Yocto*'s environment had already been configured for a chosen target.

11.1.3.7.2 Changing the Hypervisor Build Options

The hypervisor provides a number of build time parameters that can be changed via the hypervisor's Kconfig utility. These options allow the developer to select only required features which will reduce the hypervisor's code size.

To change the build time configuration options do the following:

```
cd [path-to-yocto-poky-build-folder]
bitbake -c cleansstate hypervisor
bitbake -c menuconfig hypervisor
```

Note: The first bitbake command forces a clean build. This is needed due to how Yocto tracks build dependencies. If it is not provided, the hypervisor won't be rebuilt and thus the new configuration won't be taken into consideration.

A menu will appear that shows the configuration of certain build time parameters. Built in help is provided for these build options. The basic user interface is similar to the Linux kernel's Kconfig tool.

See the screenshot below.

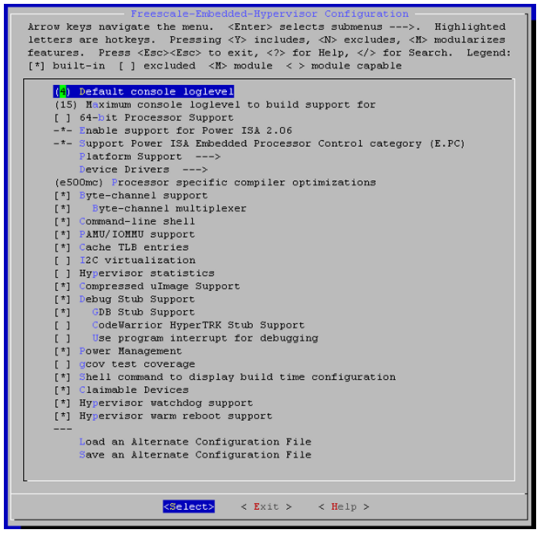


Figure 366.

After making desired build configuration changes, exit and save the configuration.

To build the hypervisor and make Yocto re-deploy the hypervisor image, follow the following steps:

```
cd [path-to-yocto-poky-build-folder]
bitbake hypervisor
```

11.1.3.7.3 Deploying the Hypervisor with U-Boot

The hypervisor must be booted with an ePAPR compliant boot program such as u-boot. In the context of u-boot, five distinct images are required to boot a system with the hypervisor:

- u-boot image (the "boot program")
- hypervisor image (the "client program", typically packaged as a uImage in the context of u-boot)
- hardware device tree (DTB format)
- hypervisor configuration tree (DTB format)
- one or more guest OS images

In the context of u-boot, the hypervisor is the client program to be booted. U-boot passes a pointer to the hardware device tree to the hypervisor as specified in the ePAPR. The **bootargs** property must be set on the /**chosen** node of the hardware device tree to specify the parameter **config-addr** that tells the hypervisor where to find the configuration tree.

Example - Booting the Hypervisor with U-boot

In this example, assume images are at true physical addresses as described below:

hypervisor image: 0xe8700000

hardware device tree: 0xe8800000

hypervisor configuration tree: 0xe8900000

Table 578. booting hypervisor with u-boot

```
=> setenv bootargs config-addr=0xe8900000
=> bootm e8700000 - e8800000
```

The physical addresses of the guest OS images are specified in the partition definition within the configuration tree. See Loading Images on page 2279.

11.1.3.8 Porting an operating system to the NXP Embedded Hypervisor

The required changes that must be made to an operating system in order to have it execute as a guest under the NXP Embedded Hypervisor can be divided into 4 general categories:

1. *Initial state & boot*—the hypervisor provides an boot environment compliant with the ePAPR standard [1]. This defines the initial state of a partition and how secondary CPUs are started. An OS not compliant with the ePAPR may require minimal changes to parse the device tree passed to it.
2. *CPU*—no OS changes should be required for basic CPU operation, however there are some differences between the physical and the virtual CPU that should be considered.
3. *SoC Platform*—the SoC platform consists of all hardware resources outside the e500 family cores. A notable platform-related change involves guest OS use of the interrupt controller.

4. *Additional hypervisor provided resources and services*—in order to take advantage of hypervisor services such as byte-channels, inter-partition doorbells, and partition management, OS modifications are required.

Considerations for each area will be explored in more detail in the sections that follow.

The general philosophy adopted in the hypervisor architecture is to require no OS changes to with respect to the CPU core. That is, the hypervisor uses full virtualization where supervisor- level instructions and registers behave as they would on a native hardware e500mc/e5500.However, OS changes may be required to certain device drivers in order to access hypercall based services offered by the hypervisor.

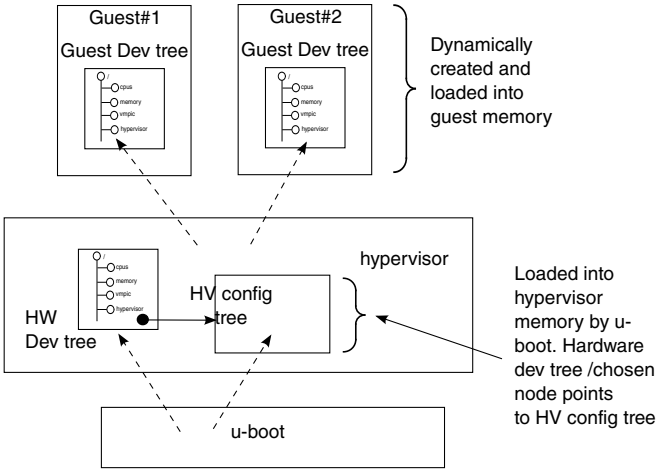
NOTE

Note: One basic assumption is that the operating system has been ported to the native e500mc/e5500 CPU. The hypervisor virtual CPU is identical in most respects to the physical CPU and thus an OS port to the e500mc/e5500 is a foundational step.

11.1.3.8.1 Initial State and Boot

The ePAPR [1] is a standard that specifies the interface between a boot program (e.g. bootloader, firmware, hypervisor) and a client program (e.g. OS, hypervisor). It defines boot-time mechanisms and the state of the platform at boot time.

A key aspect of the ePAPR is the concept of a device tree, which is a data structure that represents the hardware of a system. A device tree is passed to an ePAPR-compliant OS when it is booted. In the context of the NXP Embedded Hypervisor, a "guest device tree" is passed to the partition at boot and describes the subset of hardware resources and virtual resources that have been assigned to the partition.



The guest device tree defines nodes representing:

- CPUs
- Memory
- I/O devices
- Hypervisor specific information including virtual resources such as byte-channels, doorbells, and partition management relationships.

Guest device trees are fully described in the hypervisor reference manual [2].

An OS must be minimally device tree aware. The DTC tool (used to compile device trees from "device tree syntax" (DTS) form to "device tree blob" (DTB) form) provides a C library called libfdt. The libfdt library provides a full range of APIs to parse and manipulate device trees and will make the process of adding device tree awareness to an OS straightforward.

At a minimum an OS must determine the interrupt numbers for each hardware device from the guest device tree passed when the OS is booted. These interrupt numbers are opaque handles used as an argument to the VMPIC services APIs.

If the partition has multiple CPUs in it, the multi-CPU boot mechanisms specified by the ePAPR must be used. The guest device tree and spin table must be used in the ePAPR to release secondary CPUs.

The ePAPR specifies the initial state of the system when control is transferred to an OS.

11.1.3.8.2 CPU Related Changes

When running on the hypervisor, operating systems run in a hypervisor-controlled environment referred to as a partition. The OS view of CPUs and other hardware resources is identical or nearly identical to that of the native hardware, except that the hypervisor controls and manages the privileged resources of the system. This virtualized hardware environment is sometimes referred to as a virtual machine. One aspect of the virtual machine is the CPU. A "virtual CPU" looks nearly identical to the native CPU minus the hypervisor extensions (category Embedded.Hypervisor in the Power ISA 2.06). This means few, if any, changes are required to guest OS software with respect to basic CPU operation. Supervisor level instructions and behave as expected, with a few minor exceptions (see below).

Guest operating systems run in supervisor mode (MSR[PR]=0). This means that guest OS access to hypervisor privileged resources such as the MMU will cause a hypervisor privilege exception. The hypervisor then performs the expected behavior of the instruction or register access for the OS in a process referred to as emulation. Emulation is mostly transparent to the guest and few OS modifications should be required.

However, there are some subtle differences between a virtual CPU and a native CPU running in supervisor mode—mostly in terms of some additional restrictions. These differences are detailed in the NXP Embedded Hypervisor Software Reference Manual [2], and are summarized below. As previously mentioned a first step in porting an OS to run on the hypervisor is to port the OS to the e500mc/e5500.

The list below is a summary of differences between the physical CPU and the hypervisor virtual CPU:

- **Timebase**. The TBU and TBL SPRs are read-only. The hypervisor takes care of timebase synchronization across all CPUs.
- **Processor ID register**. The PIR is read-only to guest software. The hypervisor sets PIR to the vcpu number within the partition.
- **Cache locking**. If an operating system relies on locking cache lines, the behavior of **dcbtls,dcbtstls, dcblc, icbtls**, and **icblc** depends on the **guest cache lock** mode configuration setting in the hypervisor. Guest cache lock mode grants software in the partition the ability to lock the cache.
- **Cache Control Registers**. Certain cache control register bits may not be written. For example, an OS may not turn off caches. See the hypervisor reference manual [2] for complete details.
- **Reservations.** The ability to emulate an atomic operation using **lwarx** and stwcx is based on the conditional behavior of stwcx, the reservation set by lwarx, and the clearing of that reservation if the target location is modified by another processor or mechanism before the stwcx performs its store. In the virtual CPU a reservation is lost for any of the reasons described in the physical CPU specification and additionally any of the following reasons:
  - Invocation of a hypercall
  - Execution of certain supervisor privileged instructions which are emulated (such as TLB operations). See the hypervisor reference manual [2] for further details. This means that locks cannot be held across these emulated instructions.
  - Access to certain supervisor privileged SPRs. See the hypervisor reference manual [2] for further details.
- **Watchdog timer**. The definition of the TCR[WRC] bits is different than a native CPU. The only watchdog reset permitted is a partition reset. For managed partitions, a watchdog timeout results in an interrupt being sent to the manager partition.
- **BUCSR** (branch prediction control). BUCSR is read-only.
- **CDCSR0** (Core Device Control and Status Register 0). CDCSR0 is read-only.

- **MMU**. The number of TLB entries in TLB0 and TLB1 is different than that of a native CPU. Iterating over TLB0 using tlbre is not supported. The tlbilx instruction is present (even though it is part of category Embedded.Hypervisor) and is recommended for performing TLB invalidations.
- **Hypervisor resources**. The features and capabilities of the physical CPU defined by the category Embedded.Hypervisor in the 2.06 Power ISA™ are **not** features of the virtual CPU. These differences include:
  - The MSR[GS] bit is always set and cannot be changed.
  - The GS bit and LPID are not part of the virtual CPU's virtual address
  - Accesses to Embedded.Hypervisor category SPRs such as MSRP, LPIDR, MAS5, MAS8 are boundedly undefined. See the hypervisor reference manual [2] for further details.
  - If 64-bit category is implemented, only ICM field of EPCR register is accessible. For 32 bit guests the register is not accessible.
  - The EGS and ELPID fields in EPLC and EPSC are not writeable
  - For msgsnd, the LPIDTAG field is ignored. The G_DBELL, G_DBELL_CRIT, and G_DBELL_MC message types are not supported.
- **Thread management instructions, SPRs, TMRs.** All the thread management resources are emulated by the hypervisor: reads act as if the core has a single hardware thread (e.g. a dual-threaded e6500 core will be exposed to a guest as two single-threaded e6500 cores) writes will trigger a warning in the Hypervisor console.

Refer to the hypervisor reference manual [2] for complete details.

An operating system developer must evaluate the use of each of the above features and modify the OS to comply with the

11.1.3.8.3 SoC Platform Changes

The system-on-a-chip platform consists of the non-CPU resources such as memory-mapped I/O devices and registers. Operating systems have restricted ability to access the platform.

Most peripherals and I/O devices can be assigned directly to partitions, and thus the OS can use unmodified, native device drivers to access the devices.

However, the following devices are hypervisor-owned and should be only accessed using hypervisor provided services:

- Interrupt controller (see vmpic services)
  - The MPIC timers are not available for guest use
- Global Utilities
  - Power management
  - Clock control
  - Reset control
- Peripheral Access Management Unit (PAMU)
- DDR memory controllers (for error management)

Any guest OS access to the above resources will require modifications to use the hypervisor services provided by hypercalls.

Interrupt Controller Services

A notable platform change required in an OS will be to use the hypervisor interrupt controller services. The hypervisor interrupt controller services use a processing model identical to that of the native MPIC hardware, except that hypercall APIs are used to access manage interrupts. The table below shows the interrupt controller APIs.

Table 579. Interupt Controller Hypercalls

Hypercall	Description
EV_INT_SET_CONFIG	Configures the specified interrupt
EV_INT_GET_CONFIG	Returns the configuration of the specified interrupt
EV_INT_SET_MASK	Sets the mask for the specified interrupt source
EV_INT_GET_MASK	Returns the mask for the specified interrupt source
EV_INT_EOI	Signals the end of processing for the highest-priority interrupt currently in service.
FH_VMPIC_GET_MSIR	Gets the MPIC MSIRx (Shared Message Signaled Interrupt Register) associated with the specified interrupt source

Each interrupt generating device is dynamically assigned an opaque interrupt handle that is set in the device's interrupt specifier in the guest device tree. Interrupt handles are used in the in the API to manage the interrupt source.

The QorIQ supports an improved interface between the MPIC and CPUs called Coreint. With Coreint mode, normal external interrupts (defined as not *critical*, not *machine check,* or not *debug*) are delivered via a new interface between the MPIC and the processor core(s), called CoreInt. The CoreInt interface "pushes" the interrupt vector into the core where it can be read through a special purpose register. This reduces interrupt service latency since the interrupted processor core is no longer required to query the MPIC's IACK register to acknowledge obtain the interrupt vector.

The hypervisor supports an optional mode called *direct EOI* that in some circumstances, where CPUs and devices are dedicated to a partition, allows an EOI with no hcall. See the hypervisor reference manual [2] for additional details.

11.1.3.8.4 Additional hypervisor provided resources and services

The hypervisor provides a number of other optional services that may be used as needed by operating systems. Some services are hyper-call based and thus require OS modifications to create device drivers to use the services.

These services include the following:

- Byte-channels
- Inter-partition doorbell interrupts
- Partition management
- Shared memory

Byte-channels

Byte-channels are an interrupt-driven character-based I/O channel with functionality is similar to a UART. The hypervisor can be configured to multiplex multiple streams of character over a single physical UART. In a system with a limited number of UARTs byte-channels provide a flexible mechanism by which OSes can each get one (or more) character I/O interfaces for use as a console or debugging. The table below shows the interrupt controller APIs.

Table 580. Byte-channel Hypercalls

Hypercall	Description
EV_BYTE_CHANNEL_SEND	Sends data to a byte-channel (up to 16 characters).

Table continues on the next page...

Table 580. Byte-channel Hypercalls (continued)

EV_BYTE_CHANNEL_RECEIVE	Receives bytes of data from a byte-channel (up to 16 characters)
EV_BYTE_CHANNEL_POLL	Returns the status of the byte-channel send and receive buffers

A device driver must be written for an operating system to use the byte-channel service. NXP provides a reference Linux device drivers.

Interpartition Doorbell Interrupts

Interpartition doorbell interrupts provide a mechanism for a sending partition to interrupt one or more destination partitions. Operating system driver code must be developed to use this service. The table below shows the doorbell API.

Table 581. Interpartition Doorbell Hypercalls

Hypercall	Description
EV_PARTITION_SEND_DBELL	Sends a doorbell interrupt to the destination partitions which are associated with the specified doorbell.

The partition that receives the doorbell will have a "receive" doorbell node in its guest device tree containing an interrupt specifier for the doorbell. The doorbell is handled through the normal external handling mechanisms of the OS, but a doorbell specific driver must be created to handle the doorbell.

Partition Management

The hypervisor architecture supports the concept of a manager partition which can use hypervisor provided partition management services. A manager partition has the capability to start, stop, and restart other partitions using hyper calls. For partitions in the stopped state a manager partition can copy data to and from the managed partition, thus enabling the manager to load OS and other images into the managed partition. The table below shows the partition management APIs.

Table 582. Partition Management Hypercalls

Hypercall	Description
FH_PARTITION_START	Boots and starts execution of the specified partition.
FH_PARTITION_STOP	Stops a partition.
FH_PARTITION_RESTART	Requests that the current partition be reinitialized and restarted.
FH_PARTITION_GET_ STATUS	Gets the status of a partition.
FH_PARTITION_MEMCPY	Copies data to/from another partition's memory.
FH_PARTITION_GET_DTPROP	Gets a guest device tree property from the specified partition
FH_PARTITION_SET_DTPROP	Sets a guest device tree property in the specified partition

Shared Memory

The hypervisor provides a configuration mechanism that allows a system architect to define memory that may be shared between partitions. The existence of shared memory is communicated to a guest OS through the guest device tree. Shared memory appears as memory nodes and OS software must be developed to discover and utilize the nodes.

It may be necessary to use the node-update feature of the configuration device tree to update the memory nodes so they will be not be treated as belonging to the available general purpose memory.

Appendix A - Using a gdb stub to Debug a Partition

The hypervisor supports debugging guest operating systems via a debug stub. Host debuggers supported include gdb and CodeWarrior. The debug stub is resident inside the hypervisor and communicates to the host debugger via a byte-channel (which is normally attached directly to a UART, or a byte-channel multiplexer which is attached to a UART).

The example below shows how gdb may be used to attach to a partition with a running guest operating system.

A stub and communications channel must be specified in the partition node definition for each CPU participating in the debug session. See section .

Step 1 - Partition Configuration. Each debug stub requires a debug stub node per virtual CPU to be defined in the configuration device tree. The stub node specifies the communications channel to be used and which virtual CPU number is handled by the stub. The *guest-debug-disable* property must be set on the partition node in order to grant the CPU debug resources to the debug stub. In the example below, partition 1 which has 1 CPU defines one "gdb-stub" node which specifies uart1 as the communications channel.

Table 583. Configuration tree fragments defining debug stub node

```
/* hypervisor configuration node */
hv-config {
    compatible = "hv-config";
    ...
    uart0: uart0 {
        device = "/soc@fe000000/serial@11c500";
    };
    ...
};

uartmux: uartmux {
    compatible = "byte-channel-mux";
    endpoint = <&uart0>;
};

/* partition node for partition 1 */
partition1 {
    compatible = "partition";
    label = “partition 1”;
    guest-debug-disable;
    cpus = <1 1>;
    ...
    gdb-stub@0 {
        compatible = "gdb-stub", "debug-stub";
        endpoint = <&uartmux>;  // byte-channel endpoint
        mux-channel = <1>;
        debug-cpus = <0 1>; //virtual cpu 0
    };
};
```

Step 2 -Attaching with gdb.

After booting the target system, start the host gdb debugger and attach to the configured communications channel. In this example, assume that the target system's UART is connected to /dev/ttys0 of the host system. A hypervisor mux is used and the gdb stub mux is on channel 1.

On the host system start the mux server (mux channel 1 is on port 8001)

```
$ mux_server /dev/ttys0 8000 8001 &
```

Start gdb and attach to port 8001

```
$ powerpc-linux-gnu-gdb

GNU gdb (Sourcery G++ Lite 4.2-171) 6.7.50.20080107-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-rh73-linux-gnu --target=powerpc.
For bug reporting instructions, please see:
<https://support.codesourcery.com/GNUToolchain/>.

(gdb) target remote :8001
```

At this point gdb connects to the hypervisor and a normal gdb debug session can begin.

Selected GDB commands supported by the gdb stub:

- Run control:
  - next
  - step
  - stepi
- Breakpoints
  - To set a software breakpoint: ***break***
  - To set a hardware breakpoint (up to 2 supported): ***hbreak***
  - To set a data breakpoint: ***watch*** **,** ***awatch***
- Reset
  - To reset the partition: **monitor restart**

11.2 KVM/QEMU

11.2.1 KVM/QEMU Release Notes

```
Copyright (C) 2013-2016 Freescale Semiconductor, Inc.

Freescale KVM/QEMU Release Notes
05/29/2016

Overview
--------
This document describes new features, current limitations, and
```

```
known issues in KVM and QEMU for NXP QorIQ SDK 2.0

New Features
------------

Linux and QEMU versions:
  o  KVM is based on the SDK 2.0 Linux kernel
  o  QEMU is based on QEMU 2.4.0

Limitations Power based Platforms
-----------------------------------------

The following items describe known limitations with this release
of KVM/QEMU for Power based platforms.

  o Memory

       -Because QEMU maps all guest memory (and the mappings are
        cacheable because of Linux limitations), to avoid inconsistent
        TLB mappings guest OSes must not map memory as cache-inhibited (I=1).

  o QEMU
       -the only kernel image type supported are files in uimage format,
        created with the u-boot mkimage tool.

  o virtual CPU features
       -the fixed interval timer is not supported
       -hardware implementation dependent registers (HIDx)
        are not supported
       -L1 and L2 cache control registers are not supported
       -cache locking instructions (dcbtls, dcbtstls, dcblc, icbtls,
        icblc) are not supported
       -the CPU debug resources (debug registers, interrupt) are not
        implemented
       -the CPU performance monitor resources (performance counters
        and interrupt)
       -a hardware machine check that occurs while a virtual machine
        is running will cause QEMU to exit

  o e6500 CPU Features

       -On e6500 CPUs, hardware threads are exposed to virtual machines
        as cores, not threads.   Guest software will see a single
        threaded e6500 virtual CPU.

  o vfio-PCI on e500mc/e5500/e6500

       -Some current limitations in the PCI passthrough implementation are:

          1. It is mandatory that hugetlbfs-allocated memory be used
             when assigning PCI devices to a virtual machine

          2. PCI devices attached to one PCI controller cannot be assigned
             to different virtual machines. The supported isolation granularity
             is that of the PCI controller.

          3. This also means that PCI devices attached to one PCI controller
             cannot be used by the host kernel and a virtual machine at the
             same time. When doing PCI device assignment, all devices attached
             to a PCI controller should be "unbound" from the host.
```

```
          4. Only PCI devices using MSI interrupts are supported -- legacy
             PCI INTx interrupts are not supported.

  o Linux Kernel Build Limitations

       -It is required that the in-kernel MPIC configuration
        option (CONFIG_KVM_MPIC, "KVM in-kernel MPIC emulation") be enabled
        when building the Linux kernel to use vfio-pci or vhost support

Limitations ARM based Platforms
---------------------------------------

The following items describe known limitations with this release
of KVM/QEMU for ARM based platforms.

  o VFIO-PCI is not supported
  o PMU counters are not supported in the guest
```

11.2.2 KVM/QEMU Release Notes

Copyright (C) 2013-2016 Freescale Semiconductor, Inc.

Freescale KVM/QEMU Release Notes

06/23/2016

Overview

This document describes new features, current limitations, and

known issues in KVM and QEMU for NXP QorIQ LS1012A release.

New Features

Linux and QEMU versions:

o KVM is based on the LS1012A release Linux kernel

o QEMU is based on QEMU 2.4.0

Only basic functionality is supported: basic guest boot without I/O.

Limitations

The following items describe known limitations with this release of KVM/QEMU:

o VFIO-PCI is not supported

o 64K page size in the host kernel is not supported.

Privacy Terms of Use Terms of Sale Feedback

11.2.3 KVM/QEMU Release Notes

This document describes current limitations in the release of KVM and QEMU for NXP SoCs. Copyright (C) 2013-2016 Freescale Semiconductor, Inc.

Freescale KVM/QEMU Release Notes 19/12/2016

Overview

This document describes new features, current limitations, and known issues in KVM and QEMU for NXP QorIQ releases.

New Features

Linux and QEMU versions:

- KVM is based on the current release Linux kernel
- QEMU is based on QEMU 2.6.0

Limitations

The following items describe known limitations with this release of KVM/QEMU for ARM based platforms.

- VFIO-PCI is not supported
- PMU counters are not supported in the guest

11.2.4 Power Architecture Book E Virtual CPU Specification

```
Freescale Power Architecture Book E Virtual CPU Specification
-------------------------------------------------------------
Version 1.6
Copyright 2011-2013, Freescale Semiconductor, Inc.
May 28, 2013

                              CONTENTS

      1. OVERVIEW
          1.1 Introduction
          1.2 References
          1.3 Definitions
              1.3.1 Virtual CPU (vcpu) and Emulated CPU
              1.3.2 Guest Operating System
              1.3.3 Boundedly Undefined
              1.3.4 Volatile
          1.4 Revision History
      2. IMPLEMENTED CATEGORIES
      3. REGISTERS
          3.1 Version Registers
          3.2 Machine State Register (MSR)
          3.3  CPU Index and Processor ID Register (PIR)
          3.4  External PID Load Context (EPLC) and External PID Store Context
               (EPSC) Registers
          3.5  Timebase (TLB and TBU)
          3.6  L1 and L2 Cache Control Registers
          3.7  Branch Unit Control and Status Register (BUCSR)
          3.8  Core Device Control and Status Register 0 (CDCSR0)
          3.9  Debug Registers
          3.10 Embedded Processor Control Register (EPCR)
          3.11 Hardware Threads
      4. INSTRUCTIONS
          4.1 Cache Locking Instructions
```

```
           4.3 System Call instruction
           4.4 Wait for Interrupt Instruction
           4.5 Reservations
           4.5 tlbilx
           4.6 msgsnd/msgclr
       5. MMU
           5.1 Overview
           5.2 TLBnCFG NENTRY and ASSOC
           5.3 IPROT=0
           5.4 MMUCFG
       6. EXCEPTIONS
           6.1 Debug Interrupts
       7. HYPERVISOR SPECIFIC CONSIDERATIONS
           7.1 KVM  & Cacheable and Cache-inhibited Mappings


1.  OVERVIEW

    1.1 Introduction

        Virtualization enables multiple operating systems to run on a system,
        each in their own isolated virtual machine.  Hypervisors create and
        manage virtual machines, one part of which is a virtual CPU (or vcpu).
        A vcpu emulates a physical CPU and the behavior of instructions,
        registers, and exceptions on the vcpu is nearly identical to
        the physical CPU being emulated.

        This document defines a virtual implementation of a CPU based
        on Freescale's implementation of Book III E of the Power ISA.

        In this document, the vcpu architecture is defined in terms of
        differences between the vcpu and the following physical
        CPUs which an implementation may emulate:
           -e500mc
           -e5500
           -e6500

        The differences between the vcpu and the cpu being emulated
        should be understood by operating systems developers.

    1.2 References

        1.  Power ISA - Version 2.06 Revision B
        http://www.power.org/documentation/power-isa-version-2-06-revision-b/

        2.  EREF 2.0: A Programmer’s Reference Manual for Freescale Power
        Architecture® Processors
        http://www.freescale.com/files/32bit/doc/ref_manual/EREF_RM.pdf

        2.  PowerPC(tm) e500 Core Family Reference Manual
        http://www.freescale.com/files/32bit/doc/ref_manual/E500CORERM.pdf

        3.  e500mc Core Reference Manual, Freescale Semiconductor.
        http://www.freescale.com/files/32bit/doc/ref_manual/E500MCRM.pdf

        4.  e5500 Core Reference Manual, Freescale Semiconductors.
        Download at freescale.com with document e5500RM

        5.  ePAPR (Embedded Power Architecture Platform Requirements)
        version 1.1
```

```
        https://www.power.org/documentation/epapr-version-1-1/

    1.3 Definitions

        1.3.1 Virtual CPU (vcpu) and Emulated CPU

              A 'virtual CPU' (or vcpu) is the CPU as seen by
              software running in a virtual machine.  The vcpu
              emulates or behaves similar to some physical CPU--
              the 'emulated CPU'.

        1.3.2 Guest Operating System

              A 'guest operating system' (or 'guest') is an operating
              system running in a virtual machine created by a hypervisor.

        1.3.3 Boundedly Undefined

              The definition of the term 'boundedly undefined' used in this
              specification is identical to the Power ISA:

                  The results of executing a given instruction are said to be
                  boundedly undefined if they could have been achieved by
                  executing an arbitrary finite sequence of instructions
                  (none of which yields boundedly undefined results) in the
                  state the processor was in before executing the given
                  instruction. Boundedly undefined results for a given
                  instruction may vary between implementations, and between
                  different executions on the same implementation.

        1.3.4 Volatile

              The definition of the term 'volatile' used in this
              specification is identical to the Power ISA:

                  Bits in a register or array (e.g., TLB) are considered
                  volatile if they may change even if not explicitly
                  modified by software.

    1.4 Revision History

        Version     Date      Change
        -------------------------------------------------
        1.2        10/26/2011  updated references, msgsnd/msgclr,
                               cache control registers
        1.3        1/5/2012    Added e6500 CPU definitions
        1.4        5/2/2012    Added definitions for EPCR and MMUCFG
        1.5        6/25/2012   Category table updates for categories
                               that were missing
        1.6        5/28/2013   Added missing references to e6500,
                               updated references links, minor
                               clarifications
        1.7        5/16/2016   Drop e500v2 support

2. IMPLEMENTED CATEGORIES

   Table 2-1 below identifies the categories of the Power Architecture
   and EREF implemented by vcpu implementations for e500mc, e5500,
   and e6500.
```

```
   X indicates category is supported in vcpu
   - indicates category is not supported in vcpu
     (Note: any categories not listed are not supported in vcpu)

                              Table 2-1
-----------------------------------------------------------------
                                          Virtual CPU
Feature/Category                  Abrv.   e500mc   e5500  e6500
-----------------------------------------------------------------
Base                              B          X       X      X
Embedded                          E          X       X      X
Alternate Timebase                ATB        X       X      X
Cache Specification               CS         X       X      X
Decorated Storage                 DS         X       X      X
Embedded.Enhanced Debug           E.ED       X       X      X
Embedded.External PID             E.PD       X       X      X
Embedded.Hypervisor               E.HV       -       -      -
Embedded.Little-Endian            E.LE      [1]     [1]    [1]
Embedded.Performance Monitor      E.PM       X       X      X
Embedded.Processor Control        E.PC       X       X      X
Embedded.Cache Locking            E.CL       X       X      X
External Proxy                    EXP        X       X      X
Floating Point                    FP         X       X      X
   Floating Point.Record          FP.R       X       X      X
Memory Coherence                  MMC        X       X      X
Signal Processing Engine          SP         -       -      -
    Embedded Float Scalar Double  SP.FD      -       -      -
    Embedded Float Scalar Single  SP.FS      -       -      -
    Embedded Float Vector         SP.FV      -       -      -
Store Conditional Page Mobility   SCPM       X       X      X
Wait                              WT         X       X      X
64-bit                            64         -       X      X
Embedded.Page Table               E.PT       -       -      X
Embedded.Hypervisor.LRAT          E.HV.LRAT  -       -      -
Embedded Multi-Threading          E.EM       -       -      -
Vector (AltiVec)                  V          -       -      X
Enhanced Reservations             ER         -       -      X
 (Load and Reserve and Store Cond.)
Data Cache Extended Operations    DEO        X       X      X
Cache Stashing                    CS         X       X      X
-----------------------------------------------------------------
[1] Little-Endian mappings are supported for data but not instructions.

The ePAPR 1.1 [3] specification defines "power-isa-*" properties on
CPU nodes that specify which Power Architecture categories are
implemented.

    Property: power-isa-*
    Usage: optional
    Value: <empty>
    Description:
       If the power-isa-version property exists, then for each
       category from the Categories section of Book I of the Power
       ISA version indicated, the existence of a property named
       power-isa-[CAT], where [CAT] is the abbreviated category
       name with all uppercase letters converted to lowercase,
       indicates that the category is supported by the implementation.

       For example, if the power-isa-version property exists and
       its value is "2.06" and the power-isa-e.hv property exists,
```

```
            then the implementation supports [Category:Embedded.Hypervisor]
            as defined in Power ISA Version 2.06.

    A hypervisor should advertise implemented CPU categories on
    CPU nodes.

    An operating system should examine these properties
    to determine categories implemented by a virtual CPU.

3.  REGISTERS

    This section describes differences between registers in a virtual CPU
    compared to the CPU being emulated.

    3.1 Version Registers

        The Processor Version Register (PVR) and System Version Register (SVR)
        return the values of the CPU being emulated.

        Note: a guest should take care regarding what assumptions are made
        based on PVR as there are differences between the virtual
        CPU and the CPU being emulated as described in this specfication.

    3.2 Machine State Register (MSR)

        The machine state register (MSR) in the vcpu has differences in
        the following MSR fields as defined in Table 2-1.

             Bits    Name              Description
            ---------------------------------------------------------------
              35      GS       Guest state. MSR[GS] is read-only and is
                               always '1'. (See Note1)

              37     UCLE      User-mode cache lock enable.  Is writeable
                               and behaves as per the architecture if the
                               vcpu implements category "Embedded.Cache Locking".
                               Otherwise is '0' and is read-only.

              54      DE       Debug interrupt enable.  Is writeable and
                               and behaves as per the architecture if
                               DBCR0[EDM]=0.  If DBCR0[EDM]=1, then MSR[DE]=0
                               and is read-only. (see Note2)

            ---------------------------------------------------------------

          Notes
          -----

              Note1 - The MSR[GS] bit is defined only when the CPU being
                      emulated implements Category: Embedded.Hypervisor.

              Note2 - If the vcpu implements category "Embedded.Enhanced Debug",
                      when MSR[DE]=1, the registers SPRG9, DSRR0, and DSRR1
                      are volatile.

    3.3  CPU Index and Processor ID Register (PIR)

          The Processor ID Register is read-only.

          At virtual machine initialization, each vcpu in the virtual machine is
```

```
        assigned a unique index (within the partition) that can be used to
        distinguish the CPU from other CPUs in the partition.

        This CPU index value can be read by using the mfspr instruction to
        read the processor ID register (PIR).

        The CPU index is used in several instances:

           -The index enables software to detect whether a CPU is the boot
            CPU in an SMP configuration. The CPU index of the boot CPU is
            set by software in the device tree header (see ePAPR [3]).

           -If the vcpu implements Category: Embedded.Processor Control, the
            index is used as a parameter to the msgsnd and msgclr instructions
            to specify the targeted CPU for intra-partition signaling.

           -Interrupt source configuration in the VMPIC interrupt controller
            allows specifying the index of the CPU that is configured to
            receive the interrupt.

         Each CPU node is described in the device tree. The reg property
         for each CPU node has a value that matches the CPU index.

   3.4  External PID Load Context (EPLC) and External PID Store Context
        (EPSC) Registers

        A virtual CPU may implement [Category: Embedded.External PID]
        of the Power ISA.  EPLC and EPSC specify the context for external
        PID loads and stores as defined by the Power ISA.

        The EGS and ELPID fields in EPLC and EPSC specify the hypervisor
        context and are not accessible by supervisor level software on
        the vcpu.  Values written to the EGS and ELPID fields are
        ignored.

   3.5  Timebase (TLB and TBU)

        The TBU and TBL are read-only.

   3.6  L1 and L2 Cache Control Registers

       The behavior of the L1 and L2 cache control registers is dependent
       on whether the virtual CPU implements category "Embedded.Cache Locking".

       All L1 and L2 cache control registers and L2 error registers
       can be read regardless of whether category "Embedded.Cache Locking"
       is implemented.

       When category "Embedded.Cache Locking" is _not_ implemented:
          -The L1CSR0[CUL] and L1CSR1[ICUL] fields can be written.  For all
           other fields writes have no effect.

       When category "Embedded.Cache Locking" is implemented:
          -Writes to the flash lock clearing bits are supported--
           L1CSR0[CLFR], L1CSR1[ICLFR], L2CSR0[L2FLC]
          -Writes to the L1CSR0 sticky status bits are supported--
           L1CSR0[CUL], L1CSR0[CSLC], L1CSR0[CLO]
          -Writes to the L1CSR1 sticky status bits are supported--
           L1CSR1[ICUL], L1CSR1[ICSLC], L1CSR1[ICLO]
          -Writes to the L2CSR0 sticky status bits are supported--
```

```
          L2SCR0[L2LO]
         -Support for L1CSR0[DCBZ32] is implementation defined
         -For all other fields writes have no effect.

3.7  Branch Unit Control and Status Register (BUCSR)

     If the cpu being emulated implements BUCSR, the BUCSR fields are
     identical to those of the cpu being emulated. The BUCSR can be read,
     but is not writeable on the vcpu. Writes are NOPs and do not affect
     architectural state.

3.8  Core Device Control and Status Register 0 (CDCSR0)

     If the emulated CPU implements CDCSR0, the CDCSR0 fields are
     identical to those of the CPU being emulated. CDCSR0 can be read but
     is not writeable on the vcpu. Writes are a NOP and result in no
     architectural state changes.

3.9  Debug Registers

     The DBCR0 register in the vcpu is always readable.

     If DBCR0[EDM]=1, then the implementation has not granted debug
     resources to the vcpu.  In this case all accesses to debug
     registers (except reading DBCR0) are boundedly undefined.

     If DBCR0[EDM]=0, then the debug registers implemented by
     the CPU being emulated are supported excepted as noted below.

     Writes to the debug registers and fields in Table
     2-7 are not supported and ignored.  Reads return 0x0.

                        Table 2-7
           ------------------------------------------------
           Debug        Register fields
           Register     not supported
           ------------------------------------------------
            DBCR0       IDM - internal debug mode
                        FT - freeze timers
                        IRPT - interrupt taken
                        RET - return debug event
                        RST - reset

            DBSRWR      all fields

            DDAM        all fields

            DEVENT      all fields
           ------------------------------------------------

    When MSR[DE]=1, the registers SPRG9, DSRR0, and DSRR1 are
    volatile.

3.10 Embedded Processor Control Register (EPCR)

     EPCR is implented if category 64-bit is supported.  All bits
     in EPCR are reserved except for ICM.

         Bits    Name              Description
        ---------------------------------------------------------------
```

```
              38      ICM      Controls the computation mode for all interrupts.
                               At interrupt time, EPCR[ICM] is copied into
                               MSR[CM].

                                   0 Interrupts will execute in 32-bit mode.
                                   1 Interrupts will execute in 64-bit mode.
           -------------------------------------------------------------

    3.11 Hardware Threads

        e6500 CPUs have multiple hardware threads, but threads may
        not be exposed in a virtual machine.  No assumption should
        be made about the number of threads available based mechanisms
        such as PIR.  A e6500 vcpu may only have one thread.

        The Thread Management Configuration Register 0 (TMCFG0) should
        be used to determine the number of threads in a virtual CPU.

4.  INSTRUCTIONS

    4.1 Cache Locking Instructions

        The behavior of cache locking instructions (dcbtls, dcbtstls,
        dcblc, icbtls, icblc) is dependent on whether the virtual CPU
        implements category "Embedded.Cache Locking".  When category
        "Embedded.Cache Locking" is implemented cache locking instructions
        behave as per the architecture.   If cache locking is not implemented,
        executing cache-locking instructions is effectively a nop-- the
        operation is ignored.

    4.3 System Call instruction

        The sc instruction behaves as per the architecture except for the
        following: in user mode MSR[PR=1], sc with LEV == 1 results in a
        program exception with ESR[PPR] set (privileged instruction exception).

    4.4 Wait for Interrupt Instruction

        The 'wait' instruction stops synchronous processor activity
        including the fetching of instructions until an asynchronous
        interrupt occurs.  It is possible that a spurious 'wakeup'
        could occur where instruction fetching is resumed even
        when no vcpu interrupt or no loss of reservation occurred.

    4.5 Reservations

        The ability to emulate an atomic operation using "load with
        reservation" instructions and "store conditional" instructions
        is based on the conditional behavior of "store conditional",
        the reservation set by "load with reservation", and the clearing
        of that reservation if the target location is modified by another
        processor or mechanism before the "store condtitional" performs
        its store.

        The following considerations should be understood regarding potential
        reservation loss. With the vcpu, a reservation may be broken for
        the following reasons:

           -The Power ISA lists reasons where reservation may be lost
```

```
            -An asynchronous interrupt in the physical CPU may cause a loss
             of a reservation, including interrupts not visible to or caused
             by guest software.

            -A reservation may be broken if software executes a privileged
             instruction or utilizes a privileged facility. Privileged
             instructions and facilities are defined by the Power ISA.

    4.5 tlbilx

        The tlbilx instruction is supported on e500mc, e5500, and e6500
        virtual CPU implementations even though category E.HV is not
        supported.

    4.6 msgsnd/msgclr

        The msgsnd and msgclr instructions are defined by cateogry
        "Embedded.Processor Control" and are supported if the category
        is implemented on the cpu being emulated.

        The vcpu does not implement category "Embedded.Hypervisor".
        An attempt to use the E.HV features of msgsnd/msgclr is
        boundedly undefined.

5.  MMU

    5.1 Overview

        Software running on an vcpu implementation should not make
        assumptions about the configuration or geometry of the vcpu's
        MMU based on the PIR register.  Instead, software should determine
        MMU configuration from the MMUCFG and TLBnCFG registers.  The vcpu's
        MMU configuration may be different from the CPU being emulated.

    5.2 TLBnCFG NENTRY and ASSOC

        The Power ISA [1] specifies how TLBnCFG[NENTRY] and TLBnCFG[ASSOC]
        should be interpreted.  This definition is summarized in
        the table below:

          NENTRY  ASSOC      Meaning
          ----------------------------------------------------------------------
           0       0         no TLB present

           0       1         TLB geometry is completely implementation-defined.
                             MAS0[ESEL] is ignored

           0       >1        TLB geometry and number of entries is
                             implementation defined, but has known
                             associativity.
                             For tlbre and tlbwe, a set of TLB entries is
                             selected by an implementation dependent function
                             of MAS8[TGS][TLPID], MAS1[TS][TID][TSIZE], and
                             MAS2[EPN]. MAS0[ESEL] is used to select among
                             entries in this set, except on tlbwe if
                             MAS0[HES]=1.

           n > 0   n or 0    TLB is fully associative
          ----------------------------------------------------------------------
```

```
    5.3 IPROT=0

        A TLB entry with IPROT=0 may be evicted at any time.

    5.4 MMUCFG

        The LPIDSIZE field (bits 36-39) can be used by software to
        detect whether category E.HV is present.  A value of 0 indicates
        that E.HV functionality is not present.

6.  EXCEPTIONS

    6.1 Debug Interrupts

        The vcpu does not support delayed/deferred debug interrupts:

            -If MSR[DE]=0 and a debug condition occurs in the
             vcpu, no bit is set in DBSR.

            -Writes to DBSRWR have no effect.

            -Imprecise debug events (DBSR[IDE]) and unconditional debug
             events (DBSR[UDE]) are not supported.

            -If a debug event happens with MSR[DE] = 1, and the software
             running on the vcpu fails to clear DBSR before re-enabling
             MSR[DE] another debug interrupt will not occur.

7.  HYPERVISOR SPECIFIC CONSIDERATIONS

    7.1 Cacheable and Cache-inhibited Mappings on KVM

        As part of virtual machine initialization and setup, QEMU (the virtual
        machine manager) also creates mappings to a guest address regions.
        A guest must have I=1 for RAM mappings and I=0 for other mappings
        in order to avoid creating an architecture-violating alias with QEMU's
        mapping.
```

11.2.5 KVM for Power Architecture Users Guide and Reference

11.2.5.1 Introduction to KVM and QEMU

11.2.5.1.1 Overview

This document is a guide and tutorial to building and using KVM (Kernel-based Virtual Machine) on NXP QorIQ SoCs.

Virtualization provides an environment that enables running multiple operating systems on a single computer system. Virtualization uses hardware and software technologies together to enable this by providing an abstraction layer between system hardware and the OS. The isolated environment in which OSes run is known as a *virtual machine* (or VM). The abstraction layer that manages all this is referred to as a *hypervisor or virtual machine manager*. The hypervisor layer operates at a privilege level higher than that of the operating systems, thus enabling it to enforce system security, ensure that virtual machines cannot interfere with each other, and transparently provide other services such as I/O sharing to the VM.

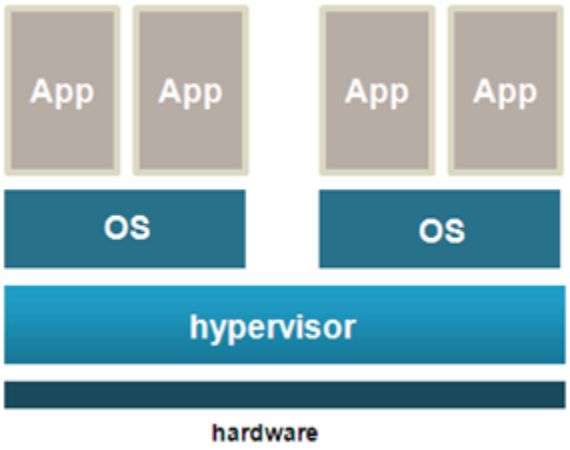


Figure 367.

KVM is a Linux kernel driver that together with QEMU, an open source machine emulator, provides an open source virtualization platfiorm based on Linux. KVM and QEMU together act as a virtual machine manager that can boot and run operating systems in virtual machines. See Figure below.

In this document the term *host* kernel refers to the underlying instance of Linux with the KVM driver that acts as the hypervisor. The term *guest* refers to the operating system, such as Linux, that runs in a virtual machine. A virtual machine will be referred to as a "VM".

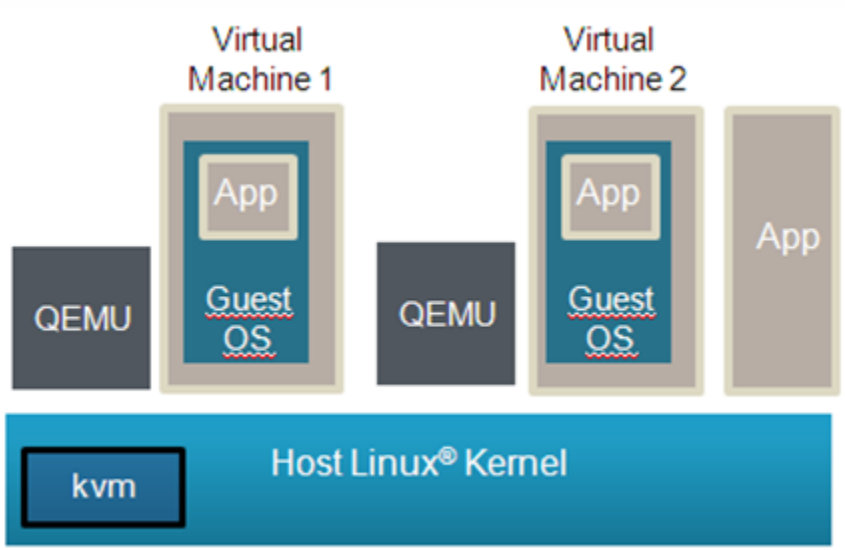


Figure 368.

NXP QorIQ SoCs based on the PowerPC e500mc, e5500 and e6500 CPUs are supported.

11.2.5.1.2 Organization of this Document

This document is organized as follows:

- Introduction to KVM and QEMU on page 2328 provides an introduction to KVM/QEMU, including overview information and references.
- Building QEMU and KVM on page 2334 provides information on how to build QEMU and the Linux kernel with KVM.

- Using QEMU and KVM on page 2338 describes how to use KVM/QEMU, including how to invoke QEMU to start virtual machines and how to set up virtual I/O and passthrough I/O devices.
- Virtual machine reference on page 2347 provides a reference for virtual machines-- details about initial VM state, virtual CPUs, and virtual I/O devices. This information is relevant when porting an OS or device driver to a KVM-based virtual machine.
- Debugging virtual machines on page 2355 describes facilities available for debugging software running in a virtual machine.
- Using KVM/QEMU with libvirt on page 2345 describes how to use the libvirt toolkit to manage KVM/QEMU virtual machines.
- KVM/QEMU How-to's on page 2357 provides a set of examples for common tasks.

11.2.5.1.3 Virtual Machine Overview

A guest OS running in a KVM/QEMU virtual machine "sees" a hardware environment similar to running on a physical board. The guest sees CPUs, memory, and a number of I/O devices. Some aspects of this environment are virtualized (emulated in software by KVM/QEMU) but this virtualization is mostly transparent to the guest, and changes to the guest are typically not required to run in a virtual machine.

The number of virtual machines that can be run simultaneously is only limited by the amount of available resources (like any other application on Linux).

There is limited support for passthrough of physical I/O devices to virtual machines. PCI and USB devices can be directly assigned to virtual machines, making the device a private resource of the virtual machine.

NXP's KVM/QEMU implements an e500-based virtual machine containing the following resources:

- One or more e500 virtual CPUs
- Memory
- Virtual PCI bus (used for virtual devices such as block and network devices)
- Virtual UART
- Virtual MPIC interrupt controller
- Virtual Global Utilities (used for guest OS initiated reset)

11.2.5.1.4 Introduction to KVM and QEMU

QEMU (pronounced KYOO-em-yoo) is a software-based machine emulator that emulates a variety of CPUs and hardware systems. KVM is a Linux kernel device driver that provides virtual CPU services to QEMU. The two software components work together as a virtual machine manager.

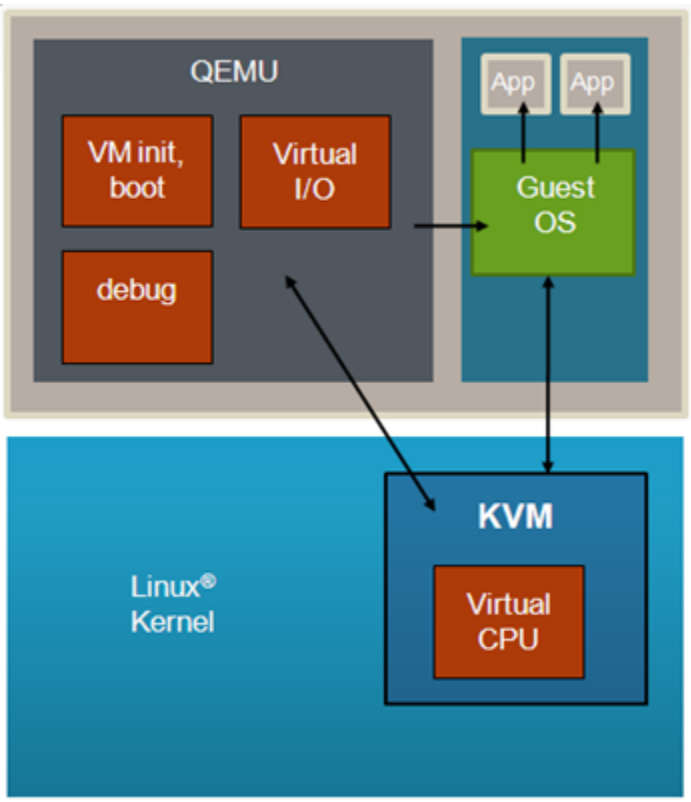


Figure 369.

QEMU is a Linux user-space application that runs on the host Linux instance and is used to start and manage a virtual machine. QEMU provides the following:

- A command line interface that provides extensive customization and configuration of a virtual machine when it is started-- e.g. type of VM, which images to load, and how virtual devices are configured
- Loading of all images needed by the guest-- e.g kernel images, root filesystem, guest device tree
- Setting the initial state of the VM and booting the guest
- Virtual I/O services, such as virtual network interfaces and virtual disks
- Debug services-which provide the capability to debug a guest OS using GDB (similar to a virtual JTAG)

KVM is a device driver in the Linux kernel whose key role in the VM architecture is to provide virtual CPU services. These services involve two aspects:

1. First, KVM provides an API set that QEMU uses to set and get the state of virtual CPUs and run them. For example, QEMU sets the initial values of the CPU's registers before starting the VM.
2. Second, after KVM starts a guest OS, certain operations (such as privileged instructions) performed by the OS cause an exception (or exit) into the host Linux kernel that must be handled and processed by KVM. This handling of traps is referred to as "emulation". These traps are transparent to the guest. For example if a guest executes a **tlbwe** instruction to update a TLB, which is a privileged resource, this results in a trap to KVM which transparently manages the physical TLB on behalf of the guest.

The KVM API is documented in the Linux kernel-- Documentation/virtual/kvm/api.txt.

KVM/QEMU supports virtual I/O which allows sharing of physical I/O devices by multiple VMs. Virtual network and block I/O are supported. See For More Information on page 2333 for references that provide additional information on virtio.

11.2.5.1.5 Device Tree Overview

A device tree is a data structure defined by the ePAPR that describes hardware resources such as CPUs, memory, and I/O devices. An ePAPR-compliant OS is passed a device tree which it reads to determine what hardware resources are available.

The host Linux kernel is booted first, typically by u-boot (an open source bootloader). U-boot passes the kernel a **hardware** device tree that lists and describes all system hardware resources available to the host kernel (CPUs/cores, memory, interrupt controller and I/O).

Similarly, when a guest OS is booted in a KVM/QEMU virtual machine, QEMU passes it a **guest** device tree that describes all the hardware resources in the VM. See Figure below.

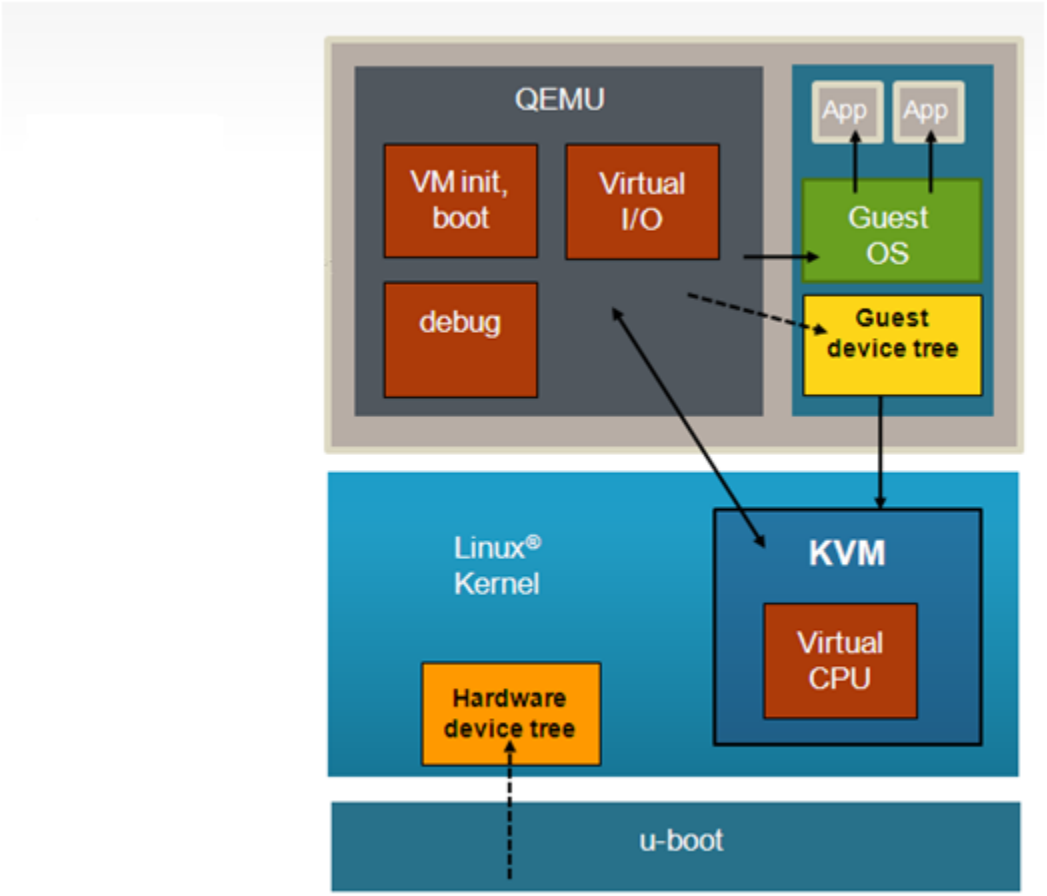


Figure 370.

The guest device tree is generated by QEMU and is used to define the resources a virtual machine will see. The guest device tree defines CPUs, memory, and I/O devices. QEMU places the guest device tree in the virtual machine's memory prior to starting the virtual machine.

11.2.5.1.6 ePAPR

The ePAPR (Embedded Power Architecture Platform Requirements) [2] specification defines platform standards for embedded Power Architecture systems. The ePAPR standard focuses on how a *boot program* (such as boot firmware, or a hypervisor) starts a *client program* (such as an operating system). The ePAPR is oriented to the requirements of embedded systems.

Standards specified in the ePAPR include the following:

- Devices trees, which describe the physical resources in a system. The standard includes:

- The syntax of device tree files (DTS syntax)
- The physical structure of compiled device trees (DTB format)
- Device tree semantics, such as requirements and conventions for the names of nodes and properties
- A description of how hardware busses and interrupts are represented.
- Specific definitions or bindings for the representation of standard hardware resources
- Initial state of a machine when a client program is started including the state of registers and initial memory mappings

- How a boot CPU starts secondary CPUs or threads.

11.2.5.1.7 References

[1] QEMU Emulator User Documentation: http://qemu.weilnetz.de/qemu-doc.html

[2] ePAPR (Embedded Power Architecture Platform Requirements) version 1.1. https://www.power.org/documentation/epapr-version-1-1/

[3] Power Architecture Book E Virtual CPU Specification on page 2319, version 1.4

11.2.5.1.8 For More Information

KVM

- KVM website: http://www.linux-kvm.org

QEMU

- QEMU website: http://www.qemu.org/

Libvirt

- ***Libvirt Users Guide*** (section in SDK documentation under Linux User Space)
- What benefits does libvirt offer to developers targeting QEMU+KVM? : https://www.berrange.com/posts/2011/06/07/what-benefits-does-libvirt-offer-to-developers-targetting-qemukvm/

Device Trees

- devicetree.org website: http://devicetree.org
- DTC, the device tree compiler is available at: http://git.jdl.com. DTC also includes a library called libfdt which can be used by software to parse device trees.

Virtio-- a framework for doing virtual I/O using KVM/QEMU

- http://www.ibm.com/developerworks/linux/library/l-virtio/
- http://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf
- http://www.linux-kvm.org/wiki/images/d/dd/KvmForum2007%24kvm_pv_drv.pdf

Virtual Networking with QEMU

- http://wiki.qemu.org/Documentation/Networking
- http://www.linux-kvm.org/page/Networking

vfio -- a framework for passing through PCI devices to a virtual machine

- https://github.com/torvalds/linux/blob/master/Documentation/vfio.txt

11.2.5.2 Building QEMU and KVM

11.2.5.2.1 Overview

Linux with KVM enabled and QEMU can be built as part of the standard build process used to build the NXP SDK using Yocto.

They can also be built in a standalone manner outside of Yocto.

The build instructions in the sections that follow assume a working understanding of how to use Yocto to build the NXP SDK. Please refer to the Yocto documentation in the SDK.

11.2.5.2.2 Building Linux with KVM

11.2.5.2.2.1 Overview

KVM is a component in the Linux kernel. KVM is not enabled in the default kernel configuration in the NXP SDK and KVM features must be enabled using the kernel's menuconfig configuration utility prior to building the kernel.

In the sections that follow configuration options are described for both the host and guest Linux kernel. The host and guest kernels can be built separately, but it is possible to build a single Linux kernel image that can be used for both the host and the guest.

The kernel configuration options described below would be the same if building the kernel standalone (outside of Yocto).

The following sections provide high level build information:

- Running menuconfig with Yocto - describes how to configure the kernel under Yocto
- Quick Start - Recommended Configuration Options - in a single step shows all the recommended configuration options to enable to build a kernel with virtual I/O enabled with the same kernel image serving as both host and guest.

The following sections provide more detailed information on each KVM-related configuration option for host and guest:

- Host Kernel: Enabling KVM - describes the configuration options to enable KVM in the host kernel.
- Host Kernel: Enabling Virtual Networking - describes how to enable bridging and tun/tap in the host kernel which enables virtual networking.
- Host Kernel: Enabling vfio-pci - describes how to pass through PCI devices to VMs
- Guest Kernel: Enabling Network and Block Virtual I/O -- describes how to enable virtual I/O in the guest kernel.

11.2.5.2.2.2 Running menuconfig with Yocto

The prerequisite and starting point for building the Linux kernel with KVM enabled is performing a standard kernel build with Yocto.

```
$ bitbake virtual/kernel
```

To change the kernel configuration options use the Linux standard menuconfig utility. To invoke menuconfig under Yocto do the following from the Yocto build environment:

```
$ bitbake -c menuconfig virtual/kernel
```

Note: Depending on what build steps may have been done previously, it may be necessary to invoke the command 'bitbake -c clean virtual/kernel' prior to running the menuconfig command.

The result will be an xterm that appears that displays the menuconfig screen:

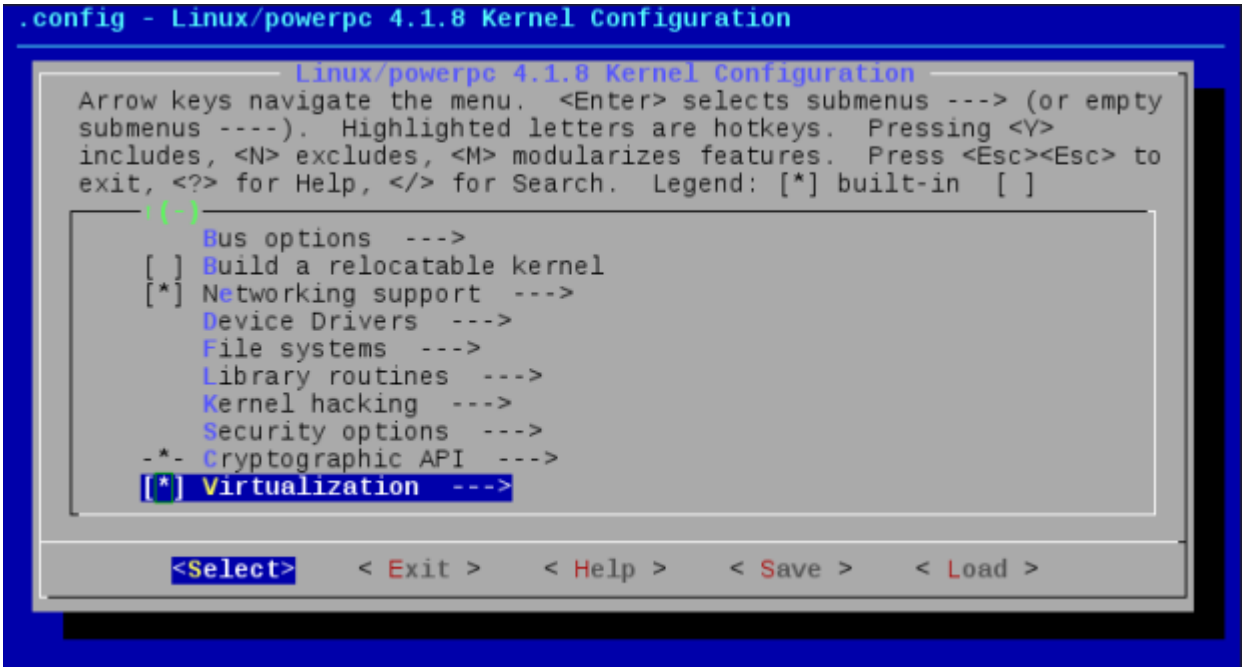


Figure 371.

11.2.5.2.2.3 Quick Start - Recommended Configuration Options

The steps below show all the recommended configuration options to enable to build a kernel with virtual I/O enabled with the same kernel image serving as both host and guest. The sections that follow explain these options in further detail.

1. From the main menuconfig window enable virtualization:

```
[*] Virtualization
```

2. In the virtualization menu enable the following options:

   For e500mc-based systems:

```
[*] KVM support for PowerPC E500MC/E5500/E6500 processors
```

   For all systems:

```
[*] KVM in-kernel MPIC emulation
<*> Host kernel accelerator for virtio net (EXPERIMENTAL)
```

3. Enable network bridging

```
Networking support  --->
    Networking options  --->
        <*> 802.1d Ethernet Bridging
```

4. Enable virtio PCI

```
Device Drivers  --->
    Virtio drivers  --->
        <*> PCI driver for virtio devices (EXPERIMENTAL)
```

5. Enable virtio for block devices

```
Device Drivers  --->
    [*] Block devices  --->
        <*>   Virtio block driver
```

6. Enable virtio for network devices

```
Device Drivers  --->
    [*] Network device support  --->
        <*>   Universal TUN/TAP device driver support
        <*>   Virtio network driver
```

11.2.5.2.2.4 Host Kernel: Enabling KVM

This section describes the core, basic options needed to enable KVM in the host kernel. KVM is enabled in the host kernel under the virtualization menu of the main kernel menuconfig window.

```
[*] Virtualization
```

Core KVM support is enabled as follows:

For e500mc-based (and e5500/e6500 systems) systems:

```
[*] KVM support for PowerPC E500MC/E5500/E6500 processors
```

For all systems enable MPIC emulation

```
 [*] KVM in-kernel MPIC emulation
```

11.2.5.2.2.5 Host Kernel: Enabling Virtual Networking

Virtual network interfaces on page 2342 describes how virtual networking can be used to give each VMs a virtual network interface which share physical network interfaces in Linux.

To get the best available performance enable the Linux kernel-based vhost network acceleration:

```
Virtualization  --->
         <*> Host kernel accelerator for virtio net (EXPERIMENTAL)
```

One common approach to configuring virtual networking is for QEMU to use a tun/tap interface bridged to a physical network interface. To do this, Ethernet bridging and the kernel's tun/tap features must be enabled in the host kernel:

```
Networking support  --->
          Networking options  --->
                    <*> 802.1d Ethernet Bridging
Device Drivers  --->
          [*] Network device support  --->
                    <*>   Universal TUN/TAP device driver support
```

11.2.5.2.2.6 Host Kernel: Enabling vfio-pci

Vfio-pci is a framework that allows PCI devices to be safely passed through to user space applications, and this framework can be used to assign PCI devices to KVM/QEMU virtual machines.

Host Kernel Config -- e500mc/e5500/e6500 based Processors

Processors based on the e500mc/e5500/e6500 all have an IOMMU (PAMU) which is used to provide isolation so that PCI devices assigned to virtual machines are restricted to accessing only a virtual machine's memory.

To enable vfio-pci on e500mc/e5500/e6500:

```
Device Drivers  --->
        <*> VFIO Non-Privileged userspace driver framework  --->
                <*> VFIO support for PCI devices
```

11.2.5.2.2.7 Guest Kernel: Enabling Network and Block Virtual I/O

Virtio is a framework for doing virtual I/O using QEMU/KVM. A virtual machine sees a virtual PCI bus with a special "virtio" virtual network and/or virtual block device on it. Guest software must have virtio drivers to use the devices.

Below the kernel configuration options are shown to enable virtio in the Linux kernel to support networking I/O and block (disk) I/O.

```
Device Drivers
          Virtio drivers  --->
                  <*>   PCI driver for virtio devices
Device Drivers  --->
          [*] Network device support  --->
                    <*>   Virtio network driver
Device Drivers  --->
          [*] Block devices  --->
                    <*>   Virtio block driver
```

11.2.5.2.3 Building QEMU

QEMU is a standard package in the QorIQ SDK and will be built for the following Yocto build configurations:

- fsl-image-virt
- fsl-image-full

To add QEMU to another Yocto build configuration, edit the conf/local.conf file and add the IMAGE_INSTALL_append variable:

```
IMAGE_INSTALL_append = " qemu"
```

After a Yocto build is complete, the target root filesystem will contain the following QEMU components that are required for using KVM/QEMU:

```
/usr/bin/qemu-system-ppc                # QEMU executable (e500mc, e5500 32 bit builds)
/usr/bin/qemu-system-ppc64              # QEMU executable (e5500, e6500 builds)
```

QEMU can be built as an individual component in the Yocto environment as well, it's package name is "qemu".

To clean and rebuild QEMU from the Yocto build environment:

```
$ bitbake -c clean qemu
$ bitbake qemu
```

11.2.5.2.4 Creating a host Linux root filesystem

11.2.5.2.4.1 Overview

Creating a Linux root filesystem is out of the scope of this document. Please reference the NXP QorIQ DPAA SDK for more information on how to create root filesystems with Yocto. This section describes the software components needed on a root filesystem to use KVM/QEMU.

The host root filesystem is the filesystem booted by the host kernel. The host rootfs is distinct from a guest root filesystem which may be needed by certain guest such as Linux.

A host root filesystem capable of running Linux as a guest needs the following components:

- Guest Linux kernel image
- QEMU executable
- Guest root filesystem
- Dynamic libraries needed by QEMU (libfdt, libz, glib2.0). These libraries are standard components in a Yocto-created rootfs.

Example host root filesystem layout with the required components to boot a Linux guest (excluding shared libraries):

```
/root/uImage                                # guest Linux kernel
/root/rootfs.ext2.gz                        # guest rootfs
/usr/bin/qemu-system-ppc                    # QEMU
```

11.2.5.2.4.2 Adding Images to a Root Filesystem with Yocto

If using Yocto, as described in Building QEMU on page 2337, the root filesystem produced by the build process will contain QEMU and the example guest device trees provided by the SDK.

A feature is also available with Yocto and the SDK to add arbitrary additional images to the root filesystem. This is done using the merge-files component in the fsl-image-core and fsl-image-full build image types in Yocto.

Any files and directories copied to the "merge" directory (see path below) will be copied to the root filesystem created by Yocto:

```
fsl-qoriq-sdk/meta-fsl-ppc/recipes-extended/merge-files/merge-files/merge
```

After populating the merge directory with the desired files, clean and rebuild the rootfs. See example below for the fsl-image-core image type:

```
$ bitbake -c install -f merge-files
$ bitbake merge-files
$ bitbake fsl-image-core
```

See the how-to article Quick-start Steps to Build and Deploy KVM Using Yocto on page 2357 for a more detailed example.

11.2.5.3 Using QEMU and KVM

11.2.5.3.1 Overview of Using QEMU

QEMU is used to start virtual machines and is built and included in the rootfs created by Yocto. The QEMU application is named **qemu-system-ppc** for e500mc builds and **qemu-system-ppc64** for e5500/e6500 builds.

In addition to the QEMU executable itself, the following is a list of the minimum components that must be available on the target system to launch a virtual machine using QEMU:

- The host Linux kernel on the target must be built with virtualization support for KVM enabled as described in Building Linux with KVM on page 2334.
- A guest OS kernel image (e.g. uImage for Linux)
- A guest root filesystem (If needed by the guest OS. For example, a Linux guest requires a rootfs.)
- Recommended: A working network interface (to interface to the guest's console and the QEMU monitor)

See the article Quick-start Steps to Run KVM Using Hugetlbfs on page 2359 for an example of how to boot a virtual machine with a rootfs created by Yocto.

The QEMU Emulator User Documentation [1] (see References on page 2333) contains complete documentation for all QEMU command line arguments. The Table below summarizes some of the flags and arguments for basic operation.

Table 584.

Argument	Descriptions
-enable-kvm	Specifies that the Linux KVM should be used for the virtual machine's CPUs
-nographic	Disables graphical output-console will be on emulated serial port.
-M machine	Specifies the type of virtual machine. One value is supported: • ppce500
-smp *cpu_count*	Specifies the number of CPUs for the virtual machine. The number of virtual CPUs allowed is the same as the value of the CONFIG_NR_CPUS config option in the host Linux kernel. To see this value issue the following command from Linux on the target board: `zcat /proc/config.gz | grep NR_CPUS`
-kernel *file*	Specifies the guest OS image. The image must be in uimage format, created with the u-boot mkimage tool. See How to Create a uImage Format Program Image on page 2367.
-initrd *file*	Specifies a root filesystem image
-append *cmdline*	Use *cmdline* as the guest OS kernel command line (passed in the bootargs property of the /chosen node in the guest device tree)
-serial *dev*	Redirects the virtual serial port to the host device *dev*. QEMU supports many possible host devices. Please refer to the QEMU User Documentation [1] (see References on page 2333) for complete details. Note: if using a tcp device with the **server** option QEMU will wait for a connection to the device before continuing unless the **nowait** option is used.

Table continues on the next page...

Table 584. (continued)

Argument	Descriptions
-m megs	Specifies the size of the VM's RAM in megabytes. This option is ignored if using direct mapped memory. See Virtual Machine Memory on page 2341 for further details on options for allocating memory.
-mem-path *path*	Specifies the path to a file from which to allocate memory for the virtual machine. This option should be used to allocate memory from hugetlbfs. See Virtual Machine Memory on page 2341 for further details on options for allocating memory.
-monitor *dev*	Redirects the QEMU monitor to the host device *dev*. QEMU supports many possible host devices. Please refer to the QEMU User Documentation [1] (see References on page 2333) for complete details. Note: if using a tcp device with the **server** option QEMU will wait for a connection to the device before continuing unless the **nowait** option is used.
-S	Do not start CPU at startup (you must type 'c' in the monitor). This can be useful if debugging.
-gdb dev	Wait for gdb connection on device dev
-drive *[args]*	Used to create a virtual disk in a virtual machine. See Virtual block devices on page 2342 for additional information.
-netdev *[args]* -device virtio-net-pci *[args]*	The -netdev and -device virtio-net-pci arguments specify the network backend and front end for creating virtual network devices in virtual machines. See Virtual network interfaces on page 2342 for additional information.
-device usb-ehci *[args]* -device usb-host *[args]*	Assigns the specified host USB device to the virtual machine. The **-device usb-ehci** argument specifies that a PCI-based EHCI USB controller should be added to the PCI bus. The **-device usb-host** identifies the specific USB device being passed through. See Passthrough of USB Devices on page 2343 for additional information.
-device vfio-pci *[args]*	Used to assign host PCI devices to virtual machines. See Passthrough of PCI Devices on page 2343 for additional information.

Table continues on the next page...

Table 584. (continued)

Argument	Descriptions
-watchdog-action *action*	The *action* value specifies the policy for when a CPU watchdog final timeout occurs. Valid values are: • **reset** // the default action, specifies that the VM should be rebooted • **poweroff** // exits QEMU • **pause** // pauses the VM • **debug** // prints a message a continues execution • **none** // no action, continues execution The **shutdown** action is not supported. See section VCPU Watchdog on page 2350.

Below is an example command line a user would run from the host Linux to start an e500mc virtual machine booting a Linux guest:

```
qemu-system-ppc -enable-kvm -nographic -m 256 -M ppce500 -kernel uImage -initrd rootfs.ext2.gz -
append "root=/dev/ram rw console=ttyS0,115200" -serial tcp::4444,server
```

11.2.5.3.2 Virtual Machine Memory

QEMU allocates and loads images into a VM's memory prior to starting the VM. The amount of memory needed for a virtual machine will be dependent on the workload to be run in the VM. There are two ways to allocate memory:

1. Allocation via hugetlbfs

   Hugetlbfs is a Linux mechanism that allows applications to allocate memory backed large physically contiguous regions of memory. QEMU can take advantage of hugetlbfs for allocation of memory for virtual machines, which can provide a significant performance improvement over malloc allocated memory. Hugetlbfs allocated memory provides the flexibility of memory that can be allocated and freed with performance comparable to direct mapped memory.

   See the document Linux HugeTLBFS in the SDK documentation which provides the technical background and detailed information on how to enable and configure hugetlbfs. The Book E implementation in Linux supports 4MB, 16MB, 64MB, 256MB, and 1GB sized huge pages. (Note: 1G huge pages are not supported on 32-bit processors).

   The **-mem-path** argument to QEMU specifies the path to the hugetlbfs mount point where the huge pages should be allocated from.

   The **-m** argument to QEMU specifies the amount of memory to allocate to the virtual machine. There are no constraints on the size passed to this argument other than that the amount of memory must fit within the constraints of the system and be enough for the workload in the VM.

   The base guest physical address of the memory node will be 0x0.

   See the how-to article Quick-start Steps to Run KVM Using Hugetlbfs on page 2359 for an example of how to use hugetlbfs.

   Note: If directly assigning a dma-capable I/O device to a virtual machine, hugetlbfs allocated memory cannot be used. In this case direct mapped memory must be used.

2. Allocation via malloc

   The default for QEMU is to allocate guest memory by the standard malloc facility available to user space applications in Linux. The amount of memory is specified with the -m command line argument. Malloc'ed memory has the flexibility of being allocated and freed by QEMU as needed. However, malloc'ed memory is backed by 4KB physical pages that are

not contiguous and emulation is required by KVM to present a contiguous guest physical memory region to the VM. This approach is discouraged since the emulation can result in a substantial performance penalty for certain workloads.

The base guest physical address of the memory node will be 0x0.

The guest device tree generated by QEMU will contain a memory node that specifies the total amount of memory.

NOTE

A virtual machine's memory is part of the address space of the QEMU process. This means that the amount of memory allocated to a VM is limited by the standard limits that exist for Linux processes. A 32-bit host kernel has a 3GiB virtual address space used for stack, text, and other data, and this limits the amount of memory that can be allocated to a VM.

11.2.5.3.3 Virtual network interfaces

QEMU provides a number of options for creating virtual network interfaces in virtual machines. Virtual network interfaces are specified using the QEMU command line and guest software discovers them by probing the virtual PCI bus.

There are two aspects of virtual network interfaces with QEMU:

1. The network “front-end”, which is the network card as seen by the guest on the PCI bus. This is specified with the **-device** QEMU argument. The argument to specify a virtio network front end would look like: **-device virtio-net-pci**
2. The network "backend", which connects the network card to some network. Network backend options include user mode networking, a host TAP interface, sockets, or virtual distributed Ethernet. The network backend is specified using the **-netdev** command line argument of QEMU. Note: It is possible to connect two virtual machines using virtual network interfaces.

For example, to use a virtio NIC card with a TAP interface back-end the QEMU command line argument would look like:

```
-netdev tap,id=tap0,script=/root/qemu-ifup,downscript=/home/root/qemu-ifdown,vhost=on -device
virtio-net-pci,netdev=tap0
```

The script “/root/qemu-ifup” is a script that QEMU invokes and passes the TAP interface name as an argument. For example, the script could add the TAP interface to an Ethernet bridge. The script "/root/qemu-ifdown" is called with the TAP interface as parameter, except this time it is called when the virtual machine stops. Following the same example, this script should remove the TAP interface from the Ethernet bridge.

See the QEMU Users Manual [1] (see References on page 2333) for detailed information about command line options and the types of network interfaces and backends. For best performance, the virtio front-end is recommended.

For additional information about QEMU networking see the references in For More Information on page 2333.

For a detailed example, see the how-to article How to Use Virtual Network Interfaces Using Virtio on page 2360 .

11.2.5.3.4 Virtual block devices

There are a number of approaches to provide a virtual disk to a KVM/QEMU virtual machine. A guest disk image can be a single raw file on the host filesystem, a file in a virtual disk format such as qcow2 and vdi, or a block device on the host Linux system. The virtual disk is assigned on the QEMU command line. In the example below, the file **my_guest_disk** is a disk image and is assigned to the VM when QEMU is launched:

```
-drive file=my_guest_disk,cache=none,if=virtio
```

Refer to the QEMU Users manual [1] (see References on page 2333) for details on the types of virtual disk images that may be created and the related arguments to QEMU.

For a detailed example, see the how to article How to Use Virtual Disks Using Virtio on page 2362.

11.2.5.3.5 Passthrough of USB Devices

USB devices can be assigned to virtual machines. When the device is assigned to the virtual machine it becomes the private resource of the VM and cannot be used by the host Linux. The virtual machine sees a EHCI USB controller on its PCI bus. The EHCI controller supports USB 2.0 devices.

There are two approaches for passing through a USB devices-- 1) by specifying the USB vendor ID and product ID of the device, or 2) by specifying the USB bus and port number.

In the examples below, the **-device usb-ehci** argument specifies that a PCI-based EHCI USB controller should be added to the PCI bus. The **-device usb-host** identifies the specific USB device being passed through.

To assign the device by vendor and product ID, first identify the device using the lsusb command. For example:

```
# lsusb
Bus 002 Device 002: ID 13fe:1e23 Kingston Technology Company Inc.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

To assign the Kingston USB disk, specify the following -device arguments to QEMU:

```
-device usb-ehci,id=ehci
-device usb-host,bus=ehci.0,vendorid=0x13fe,productid=0x1e23
```

To assign the device by USB bus and host number, use the lsusb command:

```
# lsusb -t
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=fsl-ehci/1p, 480M
    |__ Port 1: Dev 2, If 0, Class=Mass Storage, Driver=usbfs, 480M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=fsl-ehci/1p, 480M
```

In this example, the storage device can be seen on bus 2, port 1. The info usbhost in the QEMU monitor can also be used to display the host USB bus and port numbers for all USB devices.

To assign the Kingston USB disk by bus and port number, specify the following -device arguments to QEMU:

```
-device usb-ehci,id=ehci
-device usb-host,bus=ehci.0,hostbus=2,hostport=1
```

11.2.5.3.6 Passthrough of PCI Devices

PCI devices can be assigned or passed through to virtual machines. This makes the assigned device the private resource of the VM, and the VM can directly access the device's registers.

NOTE

Some current limitations in the PCI passthrough implementation are:

1. PCI devices attached to one PCI controller cannot be assigned to different virtual machines. The supported isolation granularity is that of the PCI controller.
2. This also means that PCI devices attached to one PCI controller cannot be used by the host kernel and a virtual machine at the same time. When doing PCI device assignment, all devices attached to a PCI controller should be "unbound" from the host.
3. A PCI device that is passed through to a virtual machine cannot be bound back to the host-- a system reboot is required to do this

See the article Host Kernel: Enabling vfio-pci on page 2336 for information on how to build the host and guest Linux kernels to enable vfio-pci.

All VMs have a virtual PCI bus where they discover virtual devices such as network interfaces, virtual disks, etc. Assigned PCI devices appear as another device on the VM's PCI bus.

One requirement for assigning PCI devices is to use hugetlbfs allocated memory for the virtual machine. This provides the virtual machine with larger physically contiguous regions. See the article Virtual Machine Memory on page 2341 for additional information

Follow the steps below to assign a PCI device to a virtual machine.

Step 1. **Identify the device**. To assign a PCI device, first identify the PCI manufacturer and device id and bus:dev:func number of the device using the lspci command. In addition, processors may have more than one PCI controller and in Linux a PCI "domain" identifies the PCI controller involved. The domain number can be determined from sysfs.

Example:

```
# lspci -nn
00:00.0 PCI bridge [0604]: Freescale Semiconductor Inc Device [1957:0410] (rev 10)
01:00.0 Ethernet controller [0200]: Intel Corporation 82574L Gigabit Network Connection [8086:10d3]

# ls /sys/bus/pci/devices/
0000:00:00.0  0000:01:00.0
```

In this example, to assign the Ethernet controller the needed information is:

- manufacturer/device-id: 8086:10d3
- bus:dev:func: 01:00.0
- domain: 0x0000

Step 2. Disable D3 state switching: From Linux-v4.1 onwards there is a feature added on switching devices into their D3 state when they are not in use but on some platforms and devices switching D3 state does not work and we need to disable D3 state. Use below command to disable D3 state switching:

```
# echo 1 > /sys/module/vfio_pci/parameters/disable_idle_d3
```

Step 3. Unbind the device from the host kernel. In order to assign the device to the virtual machine it is necessary to explicitly unbind the device from the host Linux kernel so that the device will no longer be in use by a host Linux device driver. To do this echo the *bus:dev:func* into the *unbind* file in sysfs for the PCI device. Example:

```
# echo 0000:01:00.0 > /sys/bus/pci/devices/0000:01:00.0/driver/unbin
```

Step 4. Bind the PCI device to vfio. This step signals the Linux vfio subsystem to start handling device driver functions for the manufacturer and device type of the device. This action triggers a rescan of the PCI bus and results in the device being bound to vfio. Example:

```
# echo 0001:01:00.0 > /sys/bus/pci/drivers/vfio-pci/bind
```

The echo into the new_id file in sysfs tells vfio that it is to handle devices of vendor 0x8086 and device ID 0x10d3.

Step 5. Launch QEMU.

Launch QEMU with the **-device vfio-pci** argument specifying the *domain:bus:dev:func* for the assigned device. It is required that the **mem-path** argument be used to specify hugetlbfs memory allocation. The syntax of the vfio-pci argument is:

```
-device vfio-pci,host=[domain:bus:dev:func]
```

```
Where: domain - identifies the PCI domain bus - specifies the bus number dev - specifies the PCI
device number func - specifies the device function number
```

e500mc Example:

```
qemu-system-ppc -m 512 [...] -device vfio-pci,host=0000:01:00.0 -mem-path /var/lib/hugetlbfs/
pagesize-4M
```

11.2.5.3.7 Using KVM/QEMU with libvirt

Libvirt is an open source toolkit that enables the management of Linux-based virtualization technologies such as KVM/QEMU virtual machines and Linux containers.

See the chapter ***Libvirt Users Guide*** (section under Linux User Space) for an overview of libvirt and its capabilties, features, and basic usage information.

The libvirt users guide describes the lifecycle of a virtual machine being managed by libvirt. To manage a KVM/QEMU virtual machine as a libvirt domain, it is first defined in an XML file and then a "define" operation (such as **virsh define**) is used to make libvirt aware of the domain. The domain can then be started, which causes QEMU to be invoked to boot the virtual machine. Libvirt tools can then be used to manage the running domain.

Libvirt URIs

To manage KVM/QEMU domains in libvirt, a QEMU URI is used:

- For a local node: *qemu:///system*
- For a remote node: *qemu[+transport]:///[hostname]/system*

See the libvirt users guide for more information on URIs.

Creating a KVM/QEMU Domain

The recommended procedure for creating a KVM/QEMU domain is to:

1. Identify the QEMU command line that fully specifies the desired capabilities and image to boot the virtual machine. Place the command line in a text file, including the full path to the QEMU executable and any images.
2. Convert the QEMU command line to libvirt XML with the *virsh domxml-from-native* command.
3. Use *virsh define* and *virsh start* to define and start the domain.

See Libvirt KVM/QEMU Example (Power Architecture) on page 2367 for an example of creating and managing a KVM/QEMU domain.

See Libvirt KVM/QEMU -- Adding Devices Example (Power Architecture) on page 2369 for an example of assigning devices to a KVM/QEMU domain.

Managing libvirt Domains

See the libvirt users guide for more information on URIs.

A running domain can then be managed by virsh (or other libvirt compatible development tools). Below are some examples of supported management operations:

- *virsh console* - connects to the console of the VM
- *virsh suspend* - suspends a running VM. After *suspend* the domain is in the "paused" state.
- *virsh resume* - resumes a suspended domain. After *resume* the domain is in the "running" state
- *virsh list* - lists active and running VMs
- *virsh reset* - resets the VM
- *qemu-monitor-command* - allows interactions with a VMs QEMU monitor

Libvirt XML

As described above the *virsh domxml-from-native* command is used to convert a QEMU command line into a libvirt XML file.

The full definition of the XML format and all XML tags is specified at: http://libvirt.org/formatdomain.html

11.2.5.3.8 VMs and the Linux Scheduler

Each virtual machine appears to the host Linux as a process with each virtual CPU in the VM implemented as a thread. A VM appears as an instance of QEMU when looking at Linux processes as can be seen in the example below:

```
$ ps -ef
                o
                o
 2814 root      2856 S    -/bin/sh
 2830 root      2872 S    -/bin/sh
 2831 root      595m S    ../images/qemu-system-ppc -enable-kvm -nographic -
 2838 root      2640 S    telnetd
 2839 user      3372 S    -sh
 2840 root      2864 S    -sh
                o
                o
```

CPUs appear as threads. To see thread IDs use the `info cpus` command in the QEMU monitor. Example of a VM with 4 virtual CPUs:

```
(qemu) info cpus
* CPU #0: nip=0x00000000c001451c thread_id=1977
  CPU #1: nip=0x00000000c001451c thread_id=1978
  CPU #2: nip=0x00000000c001451c thread_id=1979
  CPU #3: nip=0x00000000c001451c thread_id=1980
```

To see the QEMU threads using the ps command:

```
# ps -eL | grep qemu
 1976  1976 ttyS0    00:00:00 qemu-system-ppc
 1976  1977 ttyS0    00:00:02 qemu-system-ppc
 1976  1978 ttyS0    00:00:01 qemu-system-ppc
 1976  1979 ttyS0    00:00:06 qemu-system-ppc
 1976  1980 ttyS0    00:00:01 qemu-system-ppc
```

Being a Linux thread means that standard Linux mechanisms can be used to control aspects of how the threads are scheduled relative to other threads/processes. These mechanisms include:

- process priority
- CPU affinity
- isolcpus
- cgroups

11.2.5.4 Virtual machine reference

11.2.5.4.1 VM Overview

In general the architecture of KVM/QEMU is such that few changes should be needed to guest software to run in a VM-- i.e. a full virtualization approach is used, which means that virtual CPUs and virtual I/O devices behave like the physical hardware they are emulating.

However, there are some differences between virtual machines and native hardware that should be considered when targeting an OS to a KVM virtual machine. These differences can be divided into 3 general categories that will be discussed in further detail in this section:

1. Initial state and boot
2. CPUs
3. I/O devices

11.2.5.4.2 Memory Map of Virtual I/O Devices

The ppce500 VM contain a small subset of the devices found on an SoC. The available devices will be represented in the device tree passed to the guest at boot. See the table below for a summary of the virtual I/O devices in the ppce500 VM:

Table 585.

Virtual I/O Devices for ppce500	
ppce500v2 Address	**Descriptions**
0xe0004500	UART
0xe0040000	MPIC
0xe00e0000	Global utilities. This device is used for guest initiated reset of the VM.
0xe0008000	PCI controller. The virtual PCI bus is used for other virtual I/O devices such as virtual network and block devices.
0xe1000000	PCI IO space

11.2.5.4.3 Virtual machine state at initialization

11.2.5.4.3.1 Initial State and Boot

KVM/QEMU are compliant with the ePAPR [2] which specifies the interface between a boot program and an OS. QEMU acts as a boot program and thus the ePAPR defines the initial state of a VM.

It is recommended that a guest OS be minimally device tree aware. The libfdt library (available with the DTC tool) provides a full range of APIs to parse and manipulate device trees and will make the process of adding device tree awareness to an OS straightforward.

If the partition has multiple CPUs in it, the multi-CPU boot mechanisms specified by the ePAPR must be used to release secondary CPUs.

11.2.5.4.3.2 Initial Mapped Areas

The ePAPR [2] defines the concept of an *initial mapped area* or IMA. A partition's IMA is the region of memory mapped in the MMU by the boot program that contains the entry point for the guest. Both boot CPUs and secondary CPUs begin client

program execution in an IMA. The terms *Boot IMA* and *Secondary IMA* are used to distinguish the IMAs for boot CPUs and secondary CPUs where necessary.

11.2.5.4.3.2.1 Boot IMA

By default QEMU will set up a boot IMA that spans the OS kernel image and device tree. As per the ePAPR, the MMU will be configured with the effective address of the boot IMA to be 0x0.

The size of the boot IMA is passed to the client in r7 as per the ePAPR.

11.2.5.4.3.2.2 Secondary IMA

The guest physical address of secondary IMAs can be determined by the **entry_addr** field in the spin table used to release secondary vcpus. The effective address will be 0x0. The secondary IMA will always be 4KiB in size.

11.2.5.4.3.3 Initial State of Virtual CPUs

In a VM with multiple virtual CPUs, CPU #0 is the boot CPU and all other vcpus in the partition are considered secondary. See the ePAPR for additional details on boot and secondary CPUs.

11.2.5.4.3.3.1 CPU Initial State

Within a partition, the initial state of the boot and secondary vcpus complies with the entry conditions specified in the ePAPR. The virtual CPU state is summarized below:

Table 586. vcpu Initial State

Register	Value
MSR	PR=0, supervisor state EE=0, interrupts disabled ME=0, machine check interrupt disabled CE=0, critical interrupts disabled GS=1, guest state (if host CPU implements category E.HV)
R3	For boot CPUs will be effective address of the guest device tree image. For secondary CPUs will be value of R3 field in the ePAPR spin table.
R4	0
R5	0
R6	For boot CPUs: ePAPR magic value-0x45504150 For secondary CPUs: zero.
R7	Size of the boot or secondary IMA
R8	0
R9	0
R10-R31	undefined

Table continues on the next page...

Table 586. vcpu Initial State (continued)

Register	Value
F0-F31	undefined (floating point registers)
TCR	WRC=0, no watchdog timer reset will occur WIE=0, watchdog timer interrupt disabled FIE=0, fixed interval timer disabled DIE=0, decrementer interrupt disabled
PIR	Contains the partition-relative CPU index for the current CPU.
other registers	Undefined
TLB	Has a mapping for the IMA as per the ePAPR. The initial mapping will be in TLB1.

11.2.5.4.3.3.2 Secondary CPUs

At partition start, all secondary vcpus are in a quiescent state and spinning in the spin-table as per the ePAPR. They can be released from the spin by writing the location specified in the *cpu-release-addr* property in the guest device tree as defined in the ePAPR.

Vcpus do not support a writeable PIR register and thus the *pir* field in the spin table is not supported.

11.2.5.4.3.4 Initial State of I/O Devices

The initial state of direct mapped I/O devices is undefined, except that all interrupt sources have interrupts masked at the virtual MPIC.

11.2.5.4.4 Virtual CPUs

11.2.5.4.4.1 Virtual CPU Specification

Software running in a virtual machine sees a virtual CPU that emulates an e500 CPU. The behavior of instructions, registers, and exceptions on the vcpu is nearly identical to the physical CPU being emulated.

The virtual CPU type will match that of the host hardware platform. On an e500mc system, the vcpu looks like an e500mc CPU, and the same will be true for e5500, and e6500 CPUs.

Differences between the vcpu and emulated CPU are defined in the *NXP Power Architecture Book E Virtual CPU Specification* [3].

See the KVM/QEMU release notes for any specific limitations in CPU features supported for a given release of KVM/QEMU.

11.2.5.4.4.2 Time in the Virtual CPU

Guest software is able to directly read the physical timebase registers of the CPU without hypervisor mediation. This means the VM's view of time with respect to the timebase is real, wall clock time. However, since a VM may share physical CPUs with other Linux processes the actual time the VM runs is some percentage of the available CPU time.

Suppose a VM performs a task for 50 ms and receives 50% of the physical CPU time. In this case the VM would get approximately 25 ms of work on the task done during an elapsed time of 50ms. Time appears to have been 'stolen' from the VM. There is no mechanism currently available for a VM to determine how much time is being stolen.

The decrementer in the vcpu operates in terms of real, wall clock time. A decrementer timeout of 100 ms will occur in 100 ms of wall clock time. But note, there may be additional jitter to the decrementer timeout because of other processes or VMs that share the CPU.

11.2.5.4.4.3 VCPU Watchdog

The vcpu in the e500mc, e5500 and e6500 VMs has a standard e500 watchdog timer.

When a final watchdog expiration occurs, and if TCR[WRC] is non-zero, the QEMU command line argument **-watchdog-action** controls the policy.

From the QEMU User's Manual [1]:

- The **action** controls what QEMU will do when the watchdog timer expires. The default is **reset** (forcefully reset the guest).
- Other actions are **poweroff** (exits QEMU), **pause** (pause the guest), **debug** (print a debug message and continue), or **none** (do nothing).
- The **shutdown** action is not supported.

11.2.5.4.5 Virtual MPIC

The virtual MPIC is used to manage device interrupts to the VM. The virtual MPIC implements a subset of the functionality of the standard MPIC interrupt controller found on e500-based SoCs. The implemented subset is the functionality needed to manage interrupts.

The following MPIC features are not implemented:

- Global timers
- MSIs
- Message interrupts
- Performance monitor mask registers
- Configuring sources as critical or machine check

Accesses made to unimplemented registers are ignored.

Guest software does not need to do a readback from the virtual MPIC after writing EOI. This is needed in hardware on some implementations (particularly with external proxy), but under KVM it is a wasteful trap. Likewise, under KVM, no read-back is required after masking an interrupt before it takes effect.

The table below shows the implemented registers with differences between the hardware MPIC noted.

Table 587. Virtual MPIC Registers

Global Registers			
Offset	**Register**	**Access**	**Notes**
0x4_0080	CTPR-Current task priority register	RW	
0x4_0090	WHOAMI-Who am I register	R	
0x4_00A0	IACK-Interrupt acknowledge register	R	
0x4_00B0	EOI-End of interrupt register	W	
0x4_1000	FRR-Feature reporting register	R	

Table continues on the next page...

Table 587. Virtual MPIC Registers (continued)

Global Registers			
Offset	**Register**	**Access**	**Notes**
0x4_1020	GCR-Global configuration register	RW	Only the RST bit is implemented which resets the MPIC.
0x4_1080	VIR-Vendor identification register	R	VIR is always 0x0
0x4_1090	PIR-Processor core initialization register	R	Not supported
0x4_10E0	SVR-Spurious vector register	RW	Initial value is 0xffff.
Global Timer Registers			
	Note: Only the interrupt source configuration registers for the global timers are implemented-the global timers are not implemented.		
0x4_1120	GTVPRA0 - timer A0 vector/priority register	RW	
0x4_1130	GTDRA0 - timer A0 destination register	RW	
0x4_1160	GTVPRA1 - timer A1 vector/priority register	RW	
0x4_1170	GTDRA1- timer A1 destination register	RW	
0x4_11A0	GTVPRA2- timer A2 vector/priority register	RW	
0x4_11B0	GTDRA2- timer A2 destination register	RW	
0x4_11E0	GTVPRA3- timer A3 vector/priority register	RW	
0x4_11F0	GTDRA3- timer A3 destination register	RW	
0x4_2120	GTVPRA0 timer A0 vector/priority register	RW	
0x4_2130	GTDRA0 timer A0 destination register	RW	
0x4_2160	GTVPRA1 timer A1 vector/priority register	RW	
0x4_2170	GTDRA1 timer A1 destination register	RW	
0x4_21A0	GTVPRA2 timer A2 vector/priority register	RW	
0x4_21B0	GTDRA2 timer A2 destination register	RW	
0x4_11E0	GTVPRA3 timer A3 vector/priority register	RW	
0x4_11F0	GTDRA3 timer A3 destination register	RW	

Table continues on the next page...

Table 587. Virtual MPIC Registers (continued)

Global Registers			
Offset	**Register**	**Access**	**Notes**
Interrupt Source Configuration Registers			
0x5_0000	Interrupt source 0 vector/priority register	RW	
0x5_0010	Interrupt source 0 destination register	RW	
0x5_0020	Interrupt source 1 vector/priority register	RW	
0x5_0030	Interrupt source 1 destination register	RW	
0x5_0040 - 0x5_fff0	Interrupt source 2 through interrupt source 2048 configuration registers.	RW	
Per-CPU Registers			
0x6_0080	CTPR-Current task priority register	RW	
0x6_0090	WHOAMI-Who am I register	R	
0x6_00A0	IACK-Interrupt acknowledge register	R	
0x6_00B0	EOI-End of interrupt register	W	

The device tree node for the virtual MPIC is shown below.

```
pic@40000 {
        clock-frequency = <0>;
        interrupt-controller;
        #address-cells = <0>;
        #interrupt-cells = <4>;
        reg = <0x40000 0x40000>;
        compatible = "fsl,mpic", "chrp,open-pic";
        device_type = "open-pic";
        big-endian;
};
```

11.2.5.4.6 Virtual PCI

The virtual PCI controller and bus are used for virtual devices such as virtual network devices, virtual block devices, virtual USB controller, etc. The virtual PCI controller implements a subset of the functionality of the PCI controller sufficient for an unmodified guest PCI device driver to probe the virtual PCI bus.

The table below shows the implemented registers.

Table 588. Virtual PCI Controller Registers

PCI Configuration Access Registers			
Offset	**Register**	**Access**	**Notes**
0x000	CFG_ADDR-PCI configuration address	RW	
0x004	CFG_DATA-PCI configuration data	RW	
PCI Express ATMU Registers			
0xC20 - 0xDF0	Outbound Windows 1 - 4 and Inbound Windows 1 - 4.	RW	The translation address, translation extended address, base address, and window attribute registers for outbound and inbound windows 1 - 4 can be read and written, but perform no function.

11.2.5.4.7 Virtual UART

The virtual machine implements a standard ns16550 UART which can be used by guest software for character I/O.

Table 589. Virtual UART Registers

Block Base Address (Address: 0x11c500)			
Offset	**Register**	**Access**	**Notes**
0x00	URBR-ULCR[DLAB] = 0 , Receiver buffer register	R	
0x00	UTHR-ULCR[DLAB] = 0, Transmitter holding register	W	
0x00	UDLB-ULCR[DLAB] = 1, Divisor least significant byte register	RW	
0x01	UIER-ULCR[DLAB] = 0, Interrupt enable register	RW	
0x01	UDMB-ULCR[DLAB] = 1, Divisor most significant byte register	RW	
0x02	Interrupt ID register	R	
0x02	FIFO control register	W	
0x03	Line control register	RW	
0x04	Modem control register	RW	
0x05	Line status register	R	

Table continues on the next page...

Table 589. Virtual UART Registers (continued)

Block Base Address (Address: 0x11c500)			
Offset	**Register**	**Access**	**Notes**
0x06	Modem status register	R	
0x07	Scratch register	RW	

The device tree node for the virtual UART is shown below.

```
serial@11c500 {
        cell-index = <0>;
        device_type = "serial";
        compatible = "ns16550";
        reg = <0x11c500 0x100>;
        clock-frequency = <0>;
        interrupts = <36 2 0 0>;
};
```

11.2.5.4.8 Virtual Global Utilities

A reset of a virtual machine is done via the reset control register in an emulated global utilities device. The supported registers in the global utilities are shown in the table below.

Table 590. Virtual Global Utilities Registers

Global Utilities			
Offset	**Register**	**Access**	**Notes**
Version Registers			
0xE_00A0	PVR-Processor version register	R	
0xE_00A4	SVR-System version register	R	
Status Registers			
0xE_00B0	RSTCR-Reset control register	W	A value of 0x2 will cause a virtual machine reset.

The device tree node for the virtual GUTS is shown below.

```
global-utilities@e0000 {
        compatible = "fsl,guts-rstcr";
        reg = <0xe0000 0x1000>;
        fsl,has-rstcr;
};
```

11.2.5.5 Debugging virtual machines

11.2.5.5.1 QEMU Monitor

When starting QEMU, a monitor shell is available that can be used to control and see the state of VM. By default this monitor is started in the Linux shell where QEMU is invoked.

See example below of the output when starting QEMU. The user can interact with the monitor at the (qemu) prompt.

```
QEMU waiting for connection on: telnet::0.0.0.04445,server
QEMU 2.2.0 monitor - type 'help' for more information
(qemu)
```

The monitor can also be exposed over a network port by using the `-monitor dev` command line option. See Overview of Using QEMU on page 2338 and the QEMU user's manual [1] (see References on page 2333).

Refer to the QEMU user's manual [1] for a complete listing of the monitor commands available. Below is a list of some useful commands supported in the NXP SDK implementation of QEMU:

- **help** - lists all the available commands with usage information
- **info cpus** - displays the state and thread ID of all virtual CPUs
- **info** *registers* - displays the contents of the default vcpu's registers
- **cpucpu_number** - sets the default vcpu number
- **info** *tlb* - displays the vcpu's TLBs
- **system_reset** - resets the VM
- **x**/*fmt addr* -- virtual memory dump starting at 'addr'
- **xp**/*fmt addr* -- physical memory dump starting at 'addr'

11.2.5.5.2 QEMU GDB Stub

QEMU supports debugging of a VM using gdb. QEMU contains a gdb stub that can be attached to from a host system and allows standard source level debugging capabilities to examine the state of the VM and do run control.

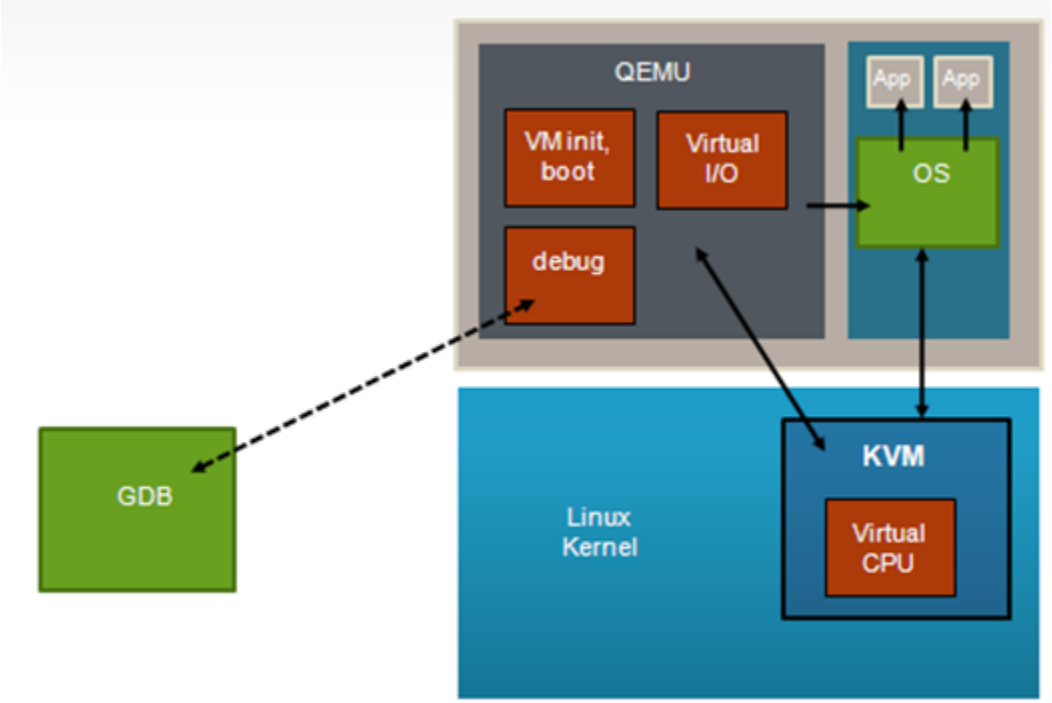


Figure 372.

To use the gdb stub, start QEMU with the -gdb *dev* option where dev specifies the type of connection to be used. See the QEMU user's manual [1] (see References on page 2333) for details.

One useful option when debugging is the -S argument to QEMU which causes QEMU to wait to start the first instruction of the guest until told to start using the monitor (**continue** command).

In the example below the tcp device type is used. A gdb stub will be active on port 4445 of the host Linux kernel when starting QEMU:

```
$ qemu-system-ppc -enable-kvm -nographic -M ppce500 -kernel uImage-p4080ds.bin -initrd devel-
image-p4080ds.ext2.gz -append "root=/dev/ram rw console=ttyS0,115200" -serial tcp::
4444,server,telnet -gdb tcp::4445

QEMU waiting for connection on: telnet::0.0.0.04444,server
QEMU 2.2.0 monitor - type 'help' for more information
(qemu)
```

After the guest has been started normally, gdb can be used to connect to the VM (in this example the host kernel has an ip address of 192.168.1.43):

```
(gdb) target remote 192.168.1.43:4445
Remote debugging using 192.168.1.43:4445
0xc000db38 in ?? ()
```

Debugging with gdb can then proceed normally:

```
(gdb) p/x $r3
$1 = 0xc075a000
```

11.2.5.6 KVM/QEMU How-to's

11.2.5.6.1 Quick-start Steps to Build and Deploy KVM Using Yocto

The following steps show how to build host and guest root filesystems, Linux kernel, and QEMU in the Yocto environment.

There are two possibilities to deploy KVM using Yocto:

- Using the preconfigured **fsl-image-virt** Yocto image as the base for host root filesystem. This image type will generate a host root filesystem which contains: QEMU, guest root filesystem and guest kernel image.
- Using other Yocto images as the base for host rootfilesystem . If the user wants to use other Yocto images to enable KVM there will be additional steps needed in order to add all the needed pieces into the host root filesystem: QEMU, guest root filesystem, guest kernel image.

11.2.5.6.1.1 Deploy KVM using fsl-image-virt Image

The host roofs is based on the **fsl-image-virt** image type and the guest rootfs is based on **fsl-image-core**.

The steps outlined below assume that the Yocto build environment is configured so that the **bitbake** command can be run.

1. Build the base host root filesystem using the **fsl-image-virt** image type.

   ```
   $ bitbake fsl-image-virt
   ```

2. Build a kernel with KVM enabled. In this case the same kernel image will be used for both host and guest.

   Configure the Linux kernel to enable KVM-related features.

   ```
   $ bitbake -c menuconfig virtual/kernel
   ```

   Follow the steps described in section Quick Start - Recommended Configuration Options on page 2335 to enable KVM in the Linux kernel.

   Then rebuild the kernel based on the new configuration options:

   ```
   $ bitbake virtual/kernel
   ```

3. Re-build the fsl-image-virt image.

   ```
   bitbake fsl-image-virt
   ```

   The resulting host rootfs will contain:

   - Linux kernel image
   - Guest root filesystem: based on **fsl-image-core**
   - QEMU

For steps to run QEMU see the article Quick-start Steps to Run KVM Using Hugetlbfs on page 2359.

11.2.5.6.1.2 Deploy KVM using fsl-image-core Image

The host roofs is based on the **fsl-image-core** image type and the guest rootfs is based on **fsl-image-minimal**.

The steps outlined below assume that the Yocto build environment is configured so that the **bitbake** command can be run.

1. Build the base host root filesystem using the **fsl-image-core** image type.

   ```
   $ bitbake fsl-image-core
   ```

2. Build a host kernel with KVM-enabled.

   Configure the Linux kernel to enable KVM-related features.

   ```
   $ bitbake -c menuconfig virtual/kernel
   ```

   Follow the steps described in section Quick Start - Recommended Configuration Options on page 2335 to enable KVM in the Linux kernel.

   Then rebuild the kernel based on the new configuration options:

   ```
   $ bitbake virtual/kernel
   ```

3. Add QEMU to the packages built by **fsl-image-core**.

   Edit the `conf/local.conf` file and append the following line which adds the QEMU package:

   ```
   IMAGE_INSTALL_append = " qemu"
   ```

4. Build a guest root filesystem and add it to the host rootfs.

   A. Build the guest roofs using the **fsl-image-minimal** image type. This creates a small rootfs sufficient for booting a Linux guest:

   ```
   $ bitbake fsl-image-minimal
   ```

   This command results in the minimal rootfs being created in tmp/deploy/images. In this example the minimal rootfs built is named: fsl-image-minimal.rootfs.ext2.gz

   B. Use the merge-files feature of Yocto (see Adding Images to a Root Filesystem with Yocto on page 2338) to copy the guest rootfs to the host rootfs.

   ```
   $ mkdir -p ../sources/meta-freescale/recipes-extended/merge-files/merge/home/root

   $ cp tmp/deploy/images/<platform>/fsl-image-minimal.rootfs.ext2.gz ../sources/meta-freescale/
   recipes-extended/merge-files/merge/home/root/guest.rootfs.ext2.gz

   $ bitbake -c install -f merge-files

   $ bitbake merge-files
   ```

5. Re-build the fsl-image-core image.

   ```
   bitbake fsl-image-core
   ```

   The resulting host rootfs will contain:

   - Linux kernel image
   - Guest root filesystem
   - QEMU, including the example guest device trees

For steps to run QEMU see the article Quick-start Steps to Run KVM Using Hugetlbfs on page 2359.

11.2.5.6.2 Quick-start Steps to Run KVM Using Hugetlbfs

The pre-requisite to this example is completing the steps in the article Quick-start Steps to Build and Deploy KVM Using Yocto on page 2357.

This example assumes that the host Linux kernel is booted, has a working network interface, and the following images are present in the host root filesystem:

- Guest kernel image (/boot/uImage)
- Guest root filesystem (/home/root/guest.rootfs.ext2.gz)
- QEMU (/usr/bin/qemu-system-ppc or qemu-system-ppc64)

There are a number of mechanisms for allocating huge pages and making them accessible via a mount point. Refer to the SDK documentation for details. This example assume allocating pages using the hugeadm command. Create a 1GB pool of 4MB huge pages, which can be used by QEMU for allocating VM memory:

```
$ hugeadm --pool-pages-min 4M:256

$ hugeadm --pool-list
      Size  Minimum  Current  Maximum  Default
   1048576        0        0        0
   4194304      256      256      256        *
  16777216        0        0        0
  67108864        0        0        0
 268435456        0        0        0
1073741824        0        0        0
```

Create a mount point to access the huge pages:

```
$ hugeadm --create-mounts

$  ls -1 /var/lib/hugetlbfs/
pagesize-16MB
pagesize-1GB
pagesize-256MB
pagesize-4MB
pagesize-64MB
```

Start QEMU specifying the 4MB huge page pool as the file from which to allocate memory. In this example 512MB of memory is allocated to the VM:

```
$ qemu-system-ppc -enable-kvm -m 512 -mem-path /var/lib/hugetlbfs/pagesize-4MB -nographic -M
ppce500 -kernel /boot/uImage -initrd ./guest.rootfs.ext2.gz -append "root=/dev/ram rw
console=ttyS0,115200" -serial tcp::4444,server,telnet

QEMU waiting for connection on: telnet::0.0.0.04444,server
```

Explanation of the command line options:

- **-enable-kvm** : specifies that KVM should be used
- **-m 512** : the amount of memory for the VM
- **-mem-path /var/lib/hugetlbfs/pagesize-4MB** : allocates from hugetlbfs based memory
- **-nographic** : don't instantiate a graphics card, this is the only option supported for the SDK
- **-M ppce500** : the type of virtual machine
- **-kernel /boot/uImage** : name of guest Linux kernel

- **-initrd ./guest.rootfs.ext2.gz** : name of guest roofs
- **-append "root=/dev/ram rw console=ttyS0,115200"** : guest Linux bootargs
- **-serial tcp::4444,server,telnet** : provide an emulated serial port (telnet server) on port 4444 on the host Linux system. Default behavior will be for QEMU to wait until the user connects to this port before booting the VM.

At this point QEMU is waiting for a telnet connection to the virtual machine's console (port 4444 of the target board) prior to starting the virtual machine.

Connect to QEMU via telnet to start the virtual machine booting. In this example the target board has IP address 192.168.1.234.

```
$ telnet 192.168.1.234 4444

Using P5020 DS machine description
Memory CAM mapping: 256/256 Mb, residual: 0Mb
Linux version 3.0.30-rt50-02082-g40c70aa (b08248@right.am.nxp.net) (gcc ve
rsion 4.6.2 (GCC) ) #2 SMP Tue May 1 17:00:37 CDT 2012
Found initrd at 0xc2000000:0xc25ab30a

...

INIT: Entering runlevel: 5
Starting syslogd/klogd: done
Stopping Bootlog daemon: bootlogd.

Yocto (Built by Poky 6.0) 1.1 p4080ds ttyS0

p4080ds login:
```

11.2.5.6.3 How to Use Virtual Network Interfaces Using Virtio

As discussed in Virtual network interfaces on page 2342, there are two aspects of virtual network interfaces-- 1) the "front end" (the device as seen by the guest OS) and 2) the "backend" (the means by the virtual device is connected to the network).

This example uses a "virtio" model NIC card and a tap network backend. The virtual network interface is bridged via a TAP interface to the physical network. The guest OS is Linux.

When starting QEMU we will add the following arguments to create the virtual network interface:

```
-netdev tap,id=tap0,script=/root/qemu-ifup,downscript=/home/root/qemu-ifdown,vhost=on -device
virtio-net-pci,netdev=tap0
```

Perform the following steps:

1. Enable virtio networking in the host and guest Linux kernels (see Host Kernel: Enabling Virtual Networking on page 2336 and Guest Kernel: Enabling Network and Block Virtual I/O on page 2337).
2. On the host Linux create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine using the **brctl** command. In this example the physical interface being used is fm1-gb1:

   ```
   $ brctl addbr br0
   $ ifconfig br0 192.168.1.139
   $ ifconfig fm1-gb1 up
   $ brctl addif br0 fm1-gb1
   ```

3. Create a qemu-ifup script on the host Linux system. For the TAP backend type, when QEMU creates the virtual network interface it invokes a user-created script that allows customization of how the TAP interface is to be handled.

The name of the TAP interface created by QEMU is passed as an argument. In this example we will bridge the the TAP inteface to the bridge created in step #2. See the example qemu-ifup script below:

```
#!/bin/sh
# TAP interface will be passed in $1
bridge=br0
guest_device=$1
ifconfig $guest_device up
brctl addif $bridge $guest_device
```

4. In order to have a smooth experience when stopping/restarting the virtual machine, you should also create a symmetrical qemu-ifdown script, that will remove the TAP interface from the bridge once the guest stops. This is not mandatory, but if it's not specified, QEMU will issue a warning on guest exit. Similarly, the name of the TAP interface created by QEMU is passed as an argument.

```

# TAP interface will be passed in $1
bridge=br0
guest_device=$1
ifconfig $guest_device down
brctl delif $bridge $guest_device
```

5. When starting QEMU specify that the network device type is "virtio" and specify the path to the script created in step #3:

```
qemu-system-ppc -enable-kvm -nographic -m 256 -M ppce500 -kernel uImage -initrd rootfs.ext2.gz -
append "root=/dev/ram rw console=ttyS0,115200"  -serial tcp::4444,server -netdev
tap,id=tap0,script=/root/qemu-ifup,downscript=/home/root/qemu-ifdown,vhost=on -device virtio-
net-pci,netdev=tap0
```

6. In the guest OS the virtual network interface will appear and can be brought up and assigned an IP address in the normal way. In the example below (the commands are run from the guest command shell) the virtio interface is eth0.

```

$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: dummy0: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether 1e:6e:b8:9a:81:32 brd ff:ff:ff:ff:ff:ff
3: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop qlen 1000
    link/ether 52:54:00:12:34:56 brd ff:ff:ff:ff:ff:ff
4: tunl0: <NOARP> mtu 1480 qdisc noop
    link/ipip 0.0.0.0 brd 0.0.0.0
5: sit0: <NOARP> mtu 1480 qdisc noop
    link/sit 0.0.0.0 brd 0.0.0.0

$ ethtool -i eth0
driver: virtio_net
version:
firmware-version:
bus-info: virtio0

$ ifconfig eth0 192.168.1.126
```

11.2.5.6.4 How to Use Virtual Disks Using Virtio

As discussed in Virtual block devices on page 2342, there are a number of formats available for virtual disk images.

The example below uses a raw file. The steps below go through the process of creating a virtual disk image, assigning it to a VM, partitioning the disk, creating a filesystem on it, and mounting it.

1. On the host Linux, create a binary image to represent the guest disk. For example to create a 16MB disk:

```
$ dd if=/dev/zero of=my_guest_disk bs=4K count=4K
```

2. Start QEMU, specifying the name of the virtual disk file for the -drive argument:

```
$ qemu-system-ppc -enable-kvm -nographic -m 256 -M ppce500 -kernel uImage
-initrd rootfs.ext2.gz -append "root=/dev/ram rw console=ttyS0,115200"
-serial tcp::4444,server,telnet -drive file=my_guest_disk,cache=none,if=virtio
```

3. After the guest has booted the virtual disk is visible as a block device in /dev with the name vda, vdb, etc.

```
$ ls -l /dev/vda
brw-r-----    1 root     disk     254,   0 Jan  1 00:02 /dev/vda
```

   A virtual block device can be treated like any other hard disk. It can be partitioned, formatted, and mounted.

4. Configure a partition on the disk with fdisk:

```
$ fdisk /dev/vda
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.
Command (m for help):
```

   Display the partition table:

```
Command (m for help): p
Disk /dev/vda: 16 MB, 16777216 bytes
16 heads, 63 sectors/track, 32 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes
   Device Boot      Start         End      Blocks  Id System
Command (m for help):
```

   Create a new partition:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-32, default 1): 1
Last cylinder or +size or +sizeM or +sizeK (1-32, default 32): Using default value 32
Command (m for help):
```

   Display the new partition:

```
Command (m for help): p
Disk /dev/vda: 16 MB, 16777216 bytes
16 heads, 63 sectors/track, 32 cylinders
```

```
Units = cylinders of 1008 * 512 = 516096 bytes
   Device Boot      Start         End      Blocks  Id System
/dev/vda1               1          32       16096+ 83 Linux
Command (m for help)
```

Write the partition table to disk and exit:

```
Command (m for help): w
The partition table has been altered!
Calling ioctl() to re-read partition table
 vda: vda1
```

5. Create a filesystem on the new partition:

```
$ mkfs.ext3 /dev/vda1
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
4032 inodes, 16096 blocks
804 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=16515072
2 block groups
8192 blocks per group, 8192 fragments per group
2016 inodes per group
Superblock backups stored on blocks:
        8193
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
This filesystem will be automatically checked every 35 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

6. Mount the filesystem:

```
$ mount /dev/vda1 /mnt/src
EXT3-fs: barriers not enabled
kjournald starting.  Commit interval 5 seconds
EXT3-fs (vda1): using internal journal
EXT3-fs (vda1): mounted filesystem with writeback data mode
```

11.2.5.6.5 Debugging: How to Examine Initial Virtual Machine State with QEMU

It can be helpful when debugging to examine the state of the virtual machine prior to executing the first instruction of the guest OS.

To do this, start QEMU with the -S option.

Example:

```
qemu-system-ppc -enable-kvm -nographic -m 256 -M ppce500 -kernel uImage -initrd rootfs.ext2.gz -
append "root=/dev/ram rw console=ttyS0,115200"  -serial tcp::4444,server,telnet -S
```

The console was started with the "-serial tcp::4444,server,telnet" option so QEMU waits for a connection prior to starting initialization. Use telnet to connect to port 4444 of the target.

At this point QEMU initializes the VM, but does not execute the entry point to the guest OS. The monitor prompt can now be used to examine initial state:

```
CQEMU waiting for connection on: telnet::0.0.0.04444,server
QEMU 1.4.0 monitor - type 'help' for more information
(qemu)
```

In this example, the VM was assigned direct-mapped memory at physical address 0x40000000.

To see where boot images are loaded and placed by QEMU use the **info roms** command:

```
(qemu) info roms
addr=0000000000000000 size=0x857540 mem=ram name="uImage"
addr=0000000002000000 size=0x010000 mem=ram name="mpc8544ds.dtb"
addr=0000000003000000 size=0x5782aa8 mem=ram name="rootfs.ext2.gz"
```

The kernel image (uImage) is loaded at the start of guest memory at 0x00000000. The guest device tree is loaded at 0x40c00000. The ramdisk is loaded at 0x42000000.

To examine the initial state of the TLBs use the **info tlb** command:

```
(qemu) info tlb

TLB0:
Effective          Physical           Size TID   TS SRWX URWX WIMGE U0123

TLB1:
Effective          Physical           Size TID   TS SRWX URWX WIMGE U0123
0x0000000000000000 0x0000000000000000  64M 0     0  SRWXURWX ----- U----
```

One TLB entry is present which maps the ePAPR defined initial mapped area (IMA). As per the ePAPR the effective address of the IMA is 0x0.

To examine the initial state of registers use the info registers command:

```
(qemu) info registers
NIP 00000000   LR 00000000 CTR 00000000 XER 00000000
MSR 10000000 HID0 00000000  HF 00000000 idx 1
TB 00000501 3593879695 DECR 00000000
GPR00 0000000000000000 0000000000000000 0000000000000000 0000000000c00000
GPR04 0000000000000000 0000000000000000 0000000045504150 0000000010000000
GPR08 0000000000000000 0000000000000000 0000000000000000 0000000000000000
GPR12 0000000000000000 0000000000000000 0000000000000000 0000000000000000
GPR16 0000000000000000 0000000000000000 0000000000000000 0000000000000000
GPR20 0000000000000000 0000000000000000 0000000000000000 0000000000000000
GPR24 0000000000000000 0000000000000000 0000000000000000 0000000000000000
GPR28 0000000000000000 0000000000000000 0000000000000000 0000000000000000
CR 00000000  [ -  -  -  -  -  -  -  -  ]             RES ffffffff
```

The program counter (NIP) is set to 0x00000000 which is the effective address of the entry point of the kernel. Other registers contain initial state as defined by the ePAPR. R3 contains the address of the guest device tree. R7 contains the size of the IMA.

11.2.5.6.6 Debugging: How to Profile Virtualization Overhead with KVM

Running software in a virtual machine can cause additional overhead that affects performance. The virtualization overhead is directly related to the number of time the hypervisor (KVM) is invoked to handle exception conditions that may occur in the virtual machine. These exception handling events are referred to as 'exits', because guest context is exited.

Examples of exits include things such the guest executing a privileged instruction, access a privileged CPU register, accessing a virtual I/O device, or a hardware interrupt such as a decrementer interrupt.

The type and number of exits that occur is workload dependent.

Counting Exits

A count of a subset of KVM exits that occur can be seen under debugfs. To see this first mount debugfs:

```
mount -t debugfs none /sys/kernel/debug
```

The exit statistics can be seen under /sys/kernel/debug/kvm. A simple script (called kvm_stat in this example) can display the statistics:

```
#!/bin/bash

IFS=$(echo -en "\n\b")

for i in $(ls -1 /sys/kernel/debug/kvm/)
do
    echo -n "$i: "
    file=`echo $i | sed 's/ /\\ /g'`
    cat "/sys/kernel/debug/kvm/$file"
done
```

Example of running the kvm_stat script above:

```
# ./kvm_stat
dcr: 0
dec: 4828
doorbell: 0
dsi: 0
dtlb_r: 627339
dtlb_v: 1475
ext_intr: 87
guest doorbell: 4142
halt_wakeup: 69
inst_emu: 984745
isi: 43
itlb_r: 544461
itlb_v: 1352
mmio: 17914
sig: 0
sysc: 0
```

Getting Exit Timings

To understand the exits that occur and how much overhead (in terms of time) that they add, a kernel configuration option is available in the host kernel that makes statistics available in /sys.

To enable exit timings enable the following option in the kernel:

```
Virtualization  --->
         [*]   Detailed exit timing
```

Then boot the host kernel normally and start the virtual machine that is to be profiled.

In the host kernel mount debugfs where the exit timing statistics will be shown:

```
mount -t debugfs none /sys/kernel/debug
```

KVM related statics for each active virtual CPU can be seen under /sys/kernel/debug/kvm in a file named vm[vcpu-thread-id]_vcpu[vcpu#]_timing. For example:

```
# ls -1 /sys/kernel/debug/kvm/vm*
/sys/kernel/debug/kvm/vm1660_vcpu0_timing
/sys/kernel/debug/kvm/vm1661_vcpu1_timing
```

To see the exit timings, use the cat command on the timing file:

```
# cat /sys/kernel/debug/kvm/vm1660_vcpu0_timing

type    count   min     max     sum     sum_squared
        MMIO    14764           1      36620         199893        43199875902
         DCR        0  238609294          0              0                  0
      SIGNAL        0  238609294          0              0                  0
    ITLBREAL   276599           0         32         255240            4883933
    ITLBVIRT      743           0         19            941              29987
    DTLBREAL   318284           0         47         314193            6337030
    DTLBVIRT      878           0          3           1076              24710
     SYSCALL        0  238609294          0              0                  0
         ISI       24           0          1             19                293
         DSI        0  238609294          0              0                  0
    EMULINST    41768           0      17099         392833        46501697994
   EMUL_WAIT        0  238609294          0              0                  0
  EMUL_WRTEE        0  238609294          0              0                  0
  EMUL_MTSPR     6912           0    6249920       96624050  10587713631926250
  EMUL_MFSPR       12           0          3             14                493
  EMUL_MTMSR        0  238609294          0              0                  0
  EMUL_MFMSR        0  238609294          0              0                  0
  EMUL_TLBSX        2           1          4              5                355
  EMUL_TLBWE   453548           0       3666         901499         1634321517
    EMUL_RFI        0  238609294          0              0                  0
   EMUL_RFCI        0  238609294          0              0                  0
      (null)        0  238609294          0              0                  0
   EMUL_RFDI        0  238609294          0              0                  0
        DEC     3607           2        119          55498           18550038
     EXTINT      101           6         47           2774            1512955
       HALT        0  238609294          0              0                  0
 USR_PR_INST      720           0         10            658              12616
  FP_UNAVAIL     1120           0          1            878              12505
      DEBUG        0  238609294          0              0                  0
    PERFMON        0  238609294          0              0                  0
 TIMEINGUEST  1124002           0       4161        7291506       246400568633
      DBELL     1009           1          5           2043              81017
 GUEST_DBELL     3911           0      37587         408794        60625818916
 PROGRAM_INT        0  238609294          0              0                  0
```

The first column identifies the exit counter, the second column identifies the count of exits, and the 5th column identifies total accumulated time for each exit type in nanoseconds (ns).

To zero the counters, echo the character 'c' to the exit timings file in debugfs. For example:

```
echo -n 'c' >/sys/kernel/debug/kvm/vm1660_vcpu0_timing
```

11.2.5.6.7 How to Create a uImage Format Program Image

QEMU can load OS kernel images passed using the **-kernel** command line flag. The image type supported is a uImage, which is an image format with its origins from the u-boot bootloader. A uImage is binary program image that with a 64-byte header that has information about the type of image.

This article shows how to take an ELF program image, convert it to a binary program image, and then use the mkimage utility to convert the program to a uImage format file. In this example the ELF program executable is called *hello*.

1. The first step is to create a binary image from the ELF executable using objcopy. The objcopy is available in the cross compile toolchain provided by the NXP SDK.

   ```
   ${CROSS_COMPILE}objcopy -O binary hello hello.bin
   ```

   Where $CROSS_COMPILE is the path to the cross compile toolchain.

2. Next convert the binary image to a uimage with the mkimage tool available in u-boot.

   ```
   mkimage -A ppc -T kernel -C none -a 00000000 -e 00000000 -d hello.bin hello.uimage
   ```

   The -a flag specfiies the physical address where to load the image. In this example, the load address is 0x0.

   The -e flag specifies the physical address of the entry point. In this example the entry point address is 0x0.

3. The parameters and values used in creating the uimage can be viewed with mkimage -l

   ```
   $ mkimage -l hello.uimage
   Image Name:
   Created:      Tue May 22 12:38:25 2012
   Image Type:   PowerPC Linux Kernel Image (uncompressed)
   Data Size:    49012 Bytes = 47.86 kB = 0.05 MB
   Load Address: 0x00000000
   Entry Point:  0x00000000
   ```

11.2.5.6.8 Libvirt KVM/QEMU Example (Power Architecture)

The following example shows the lifecycle of a simple KVM/QEMU libvirt domain called *kvm1*.

In this example the default URI is qemu:///system, and because of this default an explicit URI is not used in the virsh commands.

1. We begin with a simple QEMU command line in a text file named kvm1.args:

```
$ cat kvm1.args
/usr/bin/qemu-system-ppc -m 256 -nographic -M ppce500 -kernel /boot/uImage -initrd /home/root/
my.rootfs.ext2.gz -append "root=/dev/ram rw console=ttyS0,115200" -serial pty -enable-kvm -name kvm1
$
```

NOTE

The serial console is a tty, not a telnet server. The -name option is required and specifies the name of the virtual machine.

2. Before defining the domain the QEMU command line must be converted to libvirt XML format:

```
$ virsh domxml-from-native qemu-argv kvm1.args  > kvm1.xml
$
```

The content of the newly created domain XML file is shown below:

```
$ cat kvm1.xml
<domain type='kvm'>
  <name>kvm1</name>
  <uuid>f5b7cf86-41eb-eb78-4284-16501ff9f0e1</uuid>
  <memory unit='KiB'>262144</memory>
  <currentMemory unit='KiB'>262144</currentMemory>
  <vcpu placement='static'>1</vcpu>
  <os>
    <type arch='ppc' machine='ppce500'>hvm</type>
    <kernel>/boot/uImage</kernel>
    <initrd>/home/root/my.rootfs.ext2.gz</initrd>
    <cmdline>root=/dev/ram rw console=ttyS0,115200</cmdline>
  </os>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-ppc</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='virtio'/>
  </devices>
</domain>
```

3. Now the domain can be defined:

```
# virsh define kvm1.xml
Domain kvm1 defined from kvm1.xml

# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm1                           shut off
```

4. Next start the domain. This starts the VM and boots the guest Linux.

```
# virsh start kvm1
Domain kvm1 started
```

```
# virsh list
 Id    Name                           State
----------------------------------------------------
 3     kvm1                           running
```

5. The virsh console command can be used to connect to the console of the running Linux domain.

```
# virsh console kvm1
Connected to domain kvm1
Escape character is ^]

Poky 9.0 (Yocto Project 1.4 Reference Distro) 1.4 model : qemu ppce500 ttyS0

model : qemu ppce500 login:
```

Press CTRL +] to exit the console.

6. To stop the domain use the destroy command:

```
# virsh destroy kvm1
Domain kvm1 destroyed

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm1                           shut off
```

7. To remove the domain from libvirt, use the undefine command:

```
# virsh undefine kvm1
Domain kvm1 has been undefined

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
```

11.2.5.6.9 Libvirt KVM/QEMU -- Adding Devices Example (Power Architecture)

This example shows how devices (virtual network, USB, and PCI) can be added to a libvirt domain. The steps are identical to the simple domain example shown in Libvirt KVM/QEMU Example (Power Architecture) on page 2367 , (except some additional preparation is needed for step #1):

1. Create a text file containing the QEMU command line
2. Use **virsh domxml-from-native** to create the libvirt XML
3. Use **virsh define** to define the QEMU domain
4. Use **virsh start** to start the domain

When defining the QEMU command line options in step #1 to add the network, USB, and PCI devices it is necessary to explicitly specify the PCI slot number for each device (all the devices in this example appear on the VM's virtual PCI bus). The remainder of this example explains how to do this.

The normal QEMU command line arguments to assign these devices to a virtual machine would look something like:

- for a virtual network interface

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/root/qemu-ifdown,vhost=on -
device virtio-net-pci,netdev=tap0
```

- for the passthrough of USB device on bus #2, port #1

```
-device usb-ehci,id=ehci -device usb-host,bus=ehci.0,hostbus=2,hostport=1
```

- for the passthrough of PCI device 0000:01:00.0

```
-device vfio-pci,host=0000:01:00.0
```

(For further information on these command line argument, please refere to the KVM/QEMU documentation)

All these devices will appear on the virtual PCI bus in the virtual machine. However, in the context of libvirt it is necessary to explicitly assign the PCI slot for each device and therefore it is necessary to determine what slot are occupied and free.

First, start the virtual machine with libvirt **without the new devices** so that the default PCI slot assignment can be viewed (the example is for a domain called "kvm1" on a p4080ds):

```
$ echo '/usr/bin/qemu-system-ppc -m 512 -nographic -M ppce500 -kernel /home/root/uImage -initrd /
home/root/fsl-image-minimal-p4080ds.rootfs.ext2.gz  -append "root=/dev/ram rw
console=ttyS0,115200" -enable-kvm -smp 8 -mem-path /var/lib/hugetlbfs/pagesize-16MB -name kvm1 -
serial pty' > kvm1.args

$ virsh domxml-from-native qemu-argv kvm1.args  > kvm1.xml
$ virsh define kvm1.xml
$ virsh start kvm1
```

Next, view the devices on the PCI bus:

```
$ virsh qemu-monitor-command --hmp kvm1 'info pci'
  Bus  0, device   0, function 0:
    PCI bridge: PCI device 1957:0030
      BUS 0.
      secondary bus 0.
      subordinate bus 0.
      IO range [0x0000, 0x0fff]
      memory range [0x00000000, 0x000fffff]
      prefetchable memory range [0x00000000, 0x000fffff]
      BAR0: 32 bit memory at 0xc0000000 [0xc00fffff].
      id ""
  Bus  0, device   1, function 2:
    USB controller: PCI device 8086:7020
      IRQ 0.
      BAR4: I/O at 0xffffffffffffffff [0x001e].
      id "usb"
  Bus  0, device   3, function 0:
    Class 0255: PCI device 1af4:1002
      IRQ 0.
      BAR0: I/O at 0x1000 [0x101f].
      id "balloon0"
```

In this example it can be seen that the highest occupied slot is 0x3, which means that slots 0x4 and higher are available.

The explicit definition of the PCI slot numbers on the QEMU command line looks like this:

- for a virtual network interface (PCI slot/addr 0x4)

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/root/qemu-ifdown,vhost=on -
device virtio-net-pci,netdev=tap0,bus=pci.0,addr=0x4
```

- for the passthrough of USB device on bus #2, port #1 (PCI slot/addr 0x5)

```
-device usb-ehci,id=ehci,bus=pci.0,addr=0x5 -device usb-host,bus=ehci.0,hostbus=2,hostport=1
```

- for the passthrough of PCI device 0000:01:00.0 (PCI slot/addr 0x6)

```
-device vfio-pci,host=0000:01:00.0,bus=pci.0,addr=0x06
```

The virtual machine's PCI bus is "pci.0".

The complete command sequence to define the domain and start it looks like:

```
$ echo '/usr/bin/qemu-system-ppc -m 512 -nographic -M ppce500 -kernel /home/root/uImage -initrd /
home/root/fsl-image-minimal-p4080ds.rootfs.ext2.gz  -append "root=/dev/ram rw
console=ttyS0,115200" -enable-kvm -smp 8 -mem-path /var/lib/hugetlbfs/pagesize-16MB -name kvm1 -
serial pty -netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/root/qemu-
ifdown,vhost=on -device virtio-net-pci,netdev=tap0,bus=pci.0,addr=0x4 -device usb-
ehci,id=ehci,bus=pci.0,addr=0x5 -device usb-host,bus=ehci.0,hostbus=2,hostport=1 -device vfio-
pci,host=0000:01:00.0,bus=pci.0,addr=0x06' > kvm1.args

$ virsh domxml-from-native qemu-argv kvm1.args  > kvm1.xml
$ virsh define kvm1.xml
$ virsh start kvm1
```

Libvirt also supports adding devices into a domain by modifying the domain XML file directly.

Shutdown the guest domain and edit the guest XML definition with the virsh edit command

```
$ virsh edit kvm1
```

- for a virtual network interface (PCI slot/addr 0x4)

```
<interface type="ethernet">
  <script path="/home/root/qemu-ifup"/>
  <model type="virtio"/>
  <driver name="vhost"/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
</interface>
```

- for the passthrough of PCI device 0000:01:00.0 (PCI slot/addr 0x6)

```
<hostdev mode="subsystem" type="pci" managed="yes">
  <driver name="vfio"/>
  <source>
    <address domain="0x0000" bus="0x01" slot="0x06" function="0x0"/>
  </source>
</hostdev>
```

Start the domain again.

It is also possible to add and remove devices by defining the devicde in a standalone XML file. The **virsh attach-device/ detach-device** commands commands can be used when the domain is not active.

For example,

```
$ cat vhost_on.xml
<interface type='ethernet'>
  <script path='/home/root/qemu-ifup'/>
  <model type='virtio'/>
  <driver name='vhost'/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
</interface>
```

Attach device to the guest domain using **virsh attach-device.** The resulting XML file can be dumped to find the added device:

```
$ virsh attach-device --config kvm1 vhost_on.xml
$ virsh dumpxml kvm1
```

11.2.6 KVM for ARM Architecture Users Guide and Reference

11.2.6.1 Introduction to KVM and QEMU

11.2.6.1.1 Overview

This document is a guide and tutorial to building and using KVM (Kernel-based Virtual Machine) on NXP QorIQ SoCs.

Virtualization provides an environment that enables running multiple operating systems on a single computer system. Virtualization uses hardware and software technologies together to enable this by providing an abstraction layer between system hardware and the OS. The isolated environment in which OSes run is known as a *virtual machine* (or VM). The abstraction layer that manages all this is referred to as a *hypervisor or virtual machine manager*. The hypervisor layer operates at a privilege level higher than that of the operating systems, thus enabling it to enforce system security, ensure that virtual machines cannot interfere with each other, and transparently provide other services such as I/O sharing to the VM.

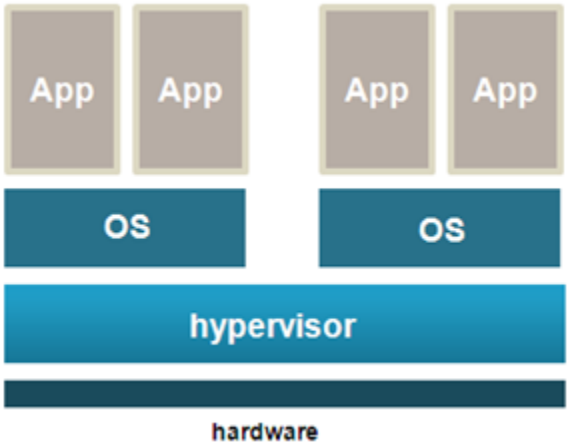


Figure 373.

KVM is a Linux kernel driver that together with QEMU, an open source machine emulator, provides an open source virtualization platfiorm based on Linux. KVM and QEMU together act as a virtual machine manager that can boot and run operating systems in virtual machines. See Figure below.

In this document the term *host* kernel refers to the underlying instance of Linux with the KVM driver that acts as the hypervisor. The term *guest* refers to the operating system, such as Linux, that runs in a virtual machine. A virtual machine will be referred to as a "VM".

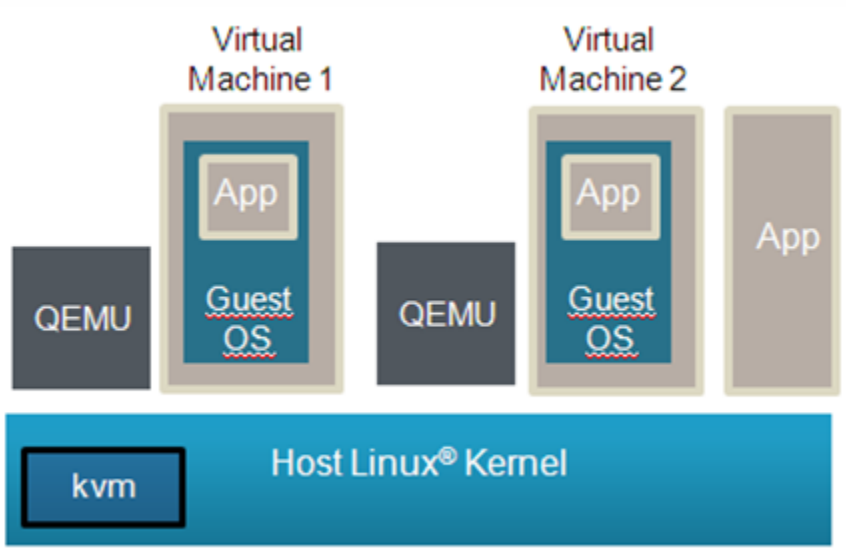


Figure 374.

NXP QorIQ SoCs based on ARM v7 and ARM v8 CPUs are supported.

11.2.6.1.2 Organization of this Document

This document is organized as follows:

- Introduction to KVM and QEMU on page 2372 provides an introduction to KVM/QEMU, including overview information and references.
- Building QEMU and KVM on page 2376 provides information on how to build QEMU and the Linux kernel with KVM.
- Using QEMU and KVM on page 2382 describes how to use KVM/QEMU, including how to invoke QEMU to start virtual machines and how to set up virtual I/O and passthrough I/O devices.
- Virtual machine reference on page 2387 provides a reference for virtual machines-- details about initial VM state, virtual CPUs, and virtual I/O devices. This information is relevant when porting an OS or device driver to a KVM-based virtual machine.
- Debugging virtual machines on page 2390 describes facilities available for debugging software running in a virtual machine.
- KVM/QEMU How-to's on page 2392 provides a set of examples for common tasks.

11.2.6.1.3 Virtual Machine Overview

A guest OS running in a KVM/QEMU virtual machine "sees" a hardware environment similar to running on a physical board. The guest sees CPUs, memory, and a number of I/O devices. Some aspects of this environment are virtualized (emulated in software by KVM/QEMU) but this virtualization is mostly transparent to the guest, and changes to the guest are typically not required to run in a virtual machine.

The number of virtual machines that can be run simultaneously is only limited by the amount of available resources (like any other application on Linux).

KVM/QEMU implements a generic virt machine which is described completely by the device tree. The virtual machine contains the following resources:

- one or more ARMv7/ARMv8 virtual CPUs
- memory

- virtual console based on an emulated PL011
- virtio over PCI (used for virtual devices such as block and network devices)
- ARM Virtual Generic Interrupt Controller
- ARM virtual timer and counter

11.2.6.1.4 Introduction to KVM and QEMU

QEMU (pronounced KYOO-em-yoo) is a software-based machine emulator that emulates a variety of CPUs and hardware systems. KVM is a Linux kernel device driver that provides virtual CPU services to QEMU. The two software components work together as a virtual machine manager.

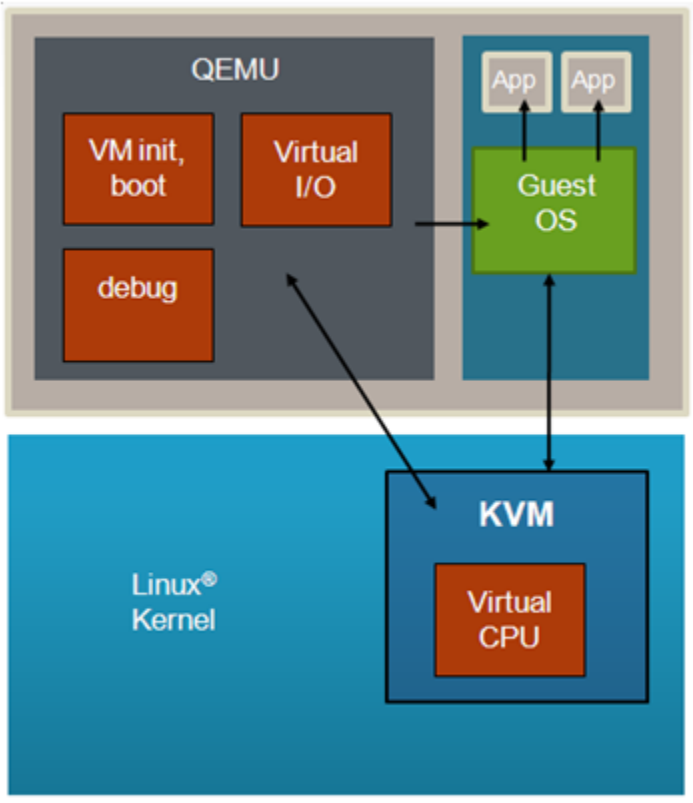


Figure 375.

QEMU is a Linux user-space application that runs on the host Linux instance and is used to start and manage a virtual machine. QEMU provides the following:

- A command line interface that provides extensive customization and configuration of a virtual machine when it is started-- e.g. type of VM, which images to load, and how virtual devices are configured
- Loading of all images needed by the guest-- e.g kernel images, root filesystem, guest device tree
- Setting the initial state of the VM and booting the guest
- Virtual I/O services, such as virtual network interfaces and virtual disks
- Debug services-which provide the capability to debug a guest OS using GDB (similar to a virtual JTAG)

KVM is a device driver in the Linux kernel whose key role in the VM architecture is to provide virtual CPU services. These services involve two aspects:

1. First, KVM provides an API set that QEMU uses to set and get the state of virtual CPUs and run them. For example, QEMU sets the initial values of the CPU's registers before starting the VM.
2. Second, after KVM starts a guest OS, certain operations (such as privileged instructions) performed by the OS cause an exception (or exit) into the host Linux kernel that must be handled and processed by KVM. This handling of traps is referred to as "emulation". These traps are transparent to the guest.

The KVM API is documented in the Linux kernel-- Documentation/virtual/kvm/api.txt.

KVM/QEMU supports virtual I/O which allows sharing of physical I/O devices by multiple VMs. Virtual network and block I/O are supported. See For More Information on page 2376 for references that provide additional information on virtio.

11.2.6.1.5 Device Tree Overview

A device tree is a data structure that describes hardware resources such as CPUs, memory, and I/O devices. An device tree aware OS is passed a device tree which it reads to determine what hardware resources are available.

The host Linux kernel is booted first, typically by u-boot (an open source bootloader). U-boot passes the kernel a **hardware** device tree that lists and describes all system hardware resources available to the host kernel (CPUs/cores, memory, interrupt controller and I/O).

Similarly, when a guest OS is booted in a KVM/QEMU virtual machine, QEMU passes it a **guest** device tree that describes all the hardware resources in the VM. See Figure below.

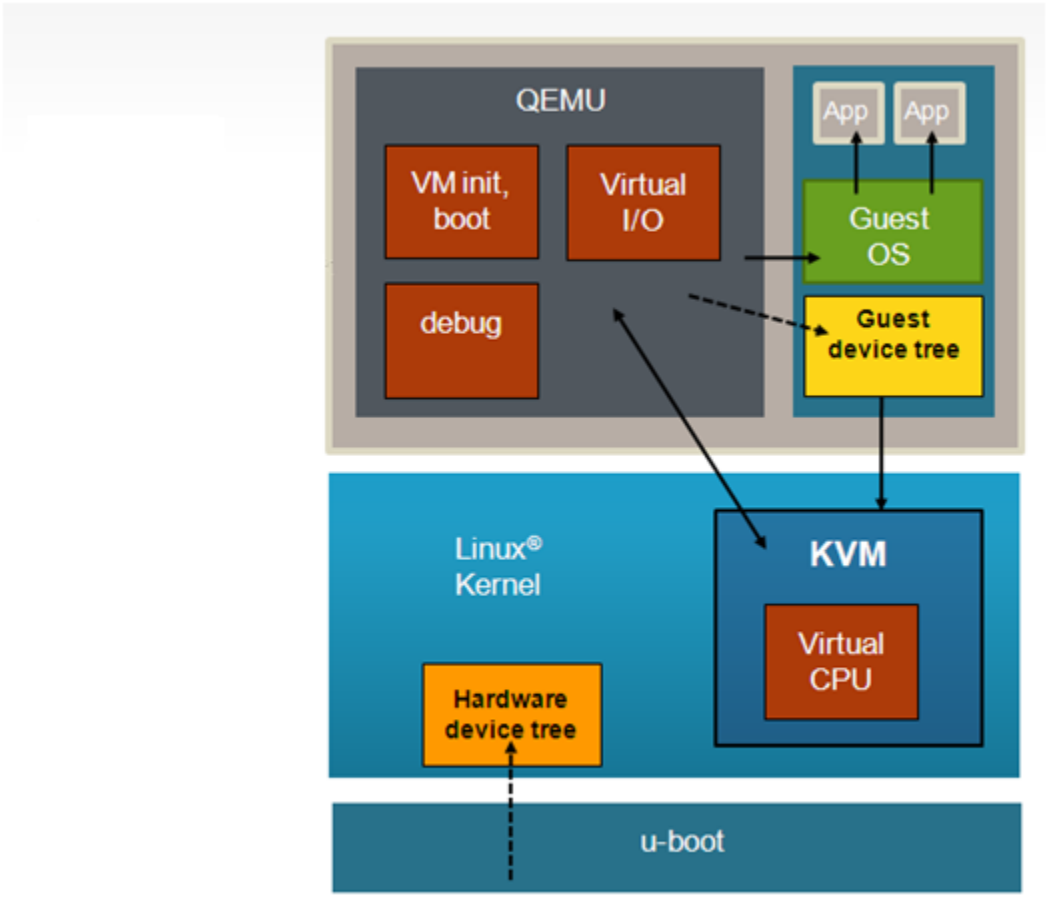


Figure 376.

The guest device tree is generated by QEMU and is used to define the resources a virtual machine will see. The guest device tree defines CPUs, memory, and I/O devices. QEMU places the guest device tree in the virtual machine's memory prior to starting the virtual machine.

11.2.6.1.6 References

[1] QEMU Emulator User Documentation: http://qemu.weilnetz.de/qemu-doc.html

[2] The Linux usage model for device tree data: https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt

[3] Specification for virtio devices: http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.pdf

11.2.6.1.7 For More Information

KVM

- KVM website: http://www.linux-kvm.org
- ARM VM specification: http://lwn.net/Articles/589122/
- Supporting KVM on ARM architecture: http://lwn.net/Articles/557132/

QEMU

- QEMU website: http://www.qemu.org/

Device Trees

- devicetree.org website: http://devicetree.org
- DTC, the device tree compiler is available at: http://git.jdl.com . DTC also includes a library called libfdt which can be used by software to parse device trees.

Virtio-- a framework for doing virtual I/O using KVM/QEMU

- http://www.ibm.com/developerworks/linux/library/l-virtio/
- http://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf
- http://www.linux-kvm.org/wiki/images/d/dd/KvmForum2007%24kvm_pv_drv.pdf
- http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.pdf

Virtual Networking with QEMU

- http://wiki.qemu.org/Documentation/Networking
- http://www.linux-kvm.org/page/Networking

11.2.6.2 Building QEMU and KVM

11.2.6.2.1 Overview

Linux with KVM enabled and QEMU can be built as part of the standard build process used to build the NXP SDK using Yocto.

They can also be built in a standalone manner outside of Yocto.

The build instructions in the sections that follow assume a working understanding of how to use Yocto to build the NXP SDK. Please refer to the Yocto documentation in the SDK.

11.2.6.2.2 Building Linux with KVM

11.2.6.2.2.1 Overview

KVM is a component in the Linux kernel. KVM is not enabled in the default kernel configuration in the NXP SDK and KVM features must be enabled using the kernel's menuconfig configuration utility prior to building the kernel.

In the sections that follow configuration options are described for both the host and guest Linux kernel. The host and guest kernels can be built separately, but it is possible to build a single Linux kernel image that can be used for both the host and the guest.

The kernel configuration options described below would be the same if building the kernel standalone (outside of Yocto).

The following sections provide high level build information:

- Running menuconfig with Yocto - describes how to configure the kernel under Yocto
- Quick Start - Recommended Configuration Options - in a single step shows all the recommended configuration options to enable to build a kernel with virtual I/O enabled with the same kernel image serving as both host and guest.

The following sections provide more detailed information on each KVM-related configuration option for host and guest:

- Host Kernel: Enabling KVM - describes the configuration options to enable KVM in the host kernel.
- Host Kernel: Enabling Virtual Networking - describes how to enable bridging and tun/tap in the host kernel which enables virtual networking.
- Guest Kernel: Enabling Network and Block Virtual I/O - describes how to enable virtual I/O in the guest kernel.
- Guest kernel: Enabling console - describes how to enable the console for the guest kernel

11.2.6.2.2.2 Running menuconfig with Yocto

The prerequisite and starting point for building the Linux kernel with KVM enabled is performing a standard kernel build with Yocto.

```
$ bitbake virtual/kernel
```

To change the kernel configuration options use the Linux standard menuconfig utility. To invoke menuconfig under Yocto do the following from the Yocto build environment:

```
$ bitbake -c menuconfig virtual/kernel
```

Note: Depending on what build steps may have been done previously, it may be necessary to invoke the command 'bitbake -c clean virtual/kernel' prior to running the menuconfig command.

The result will be an xterm that appears that displays the menuconfig screen:

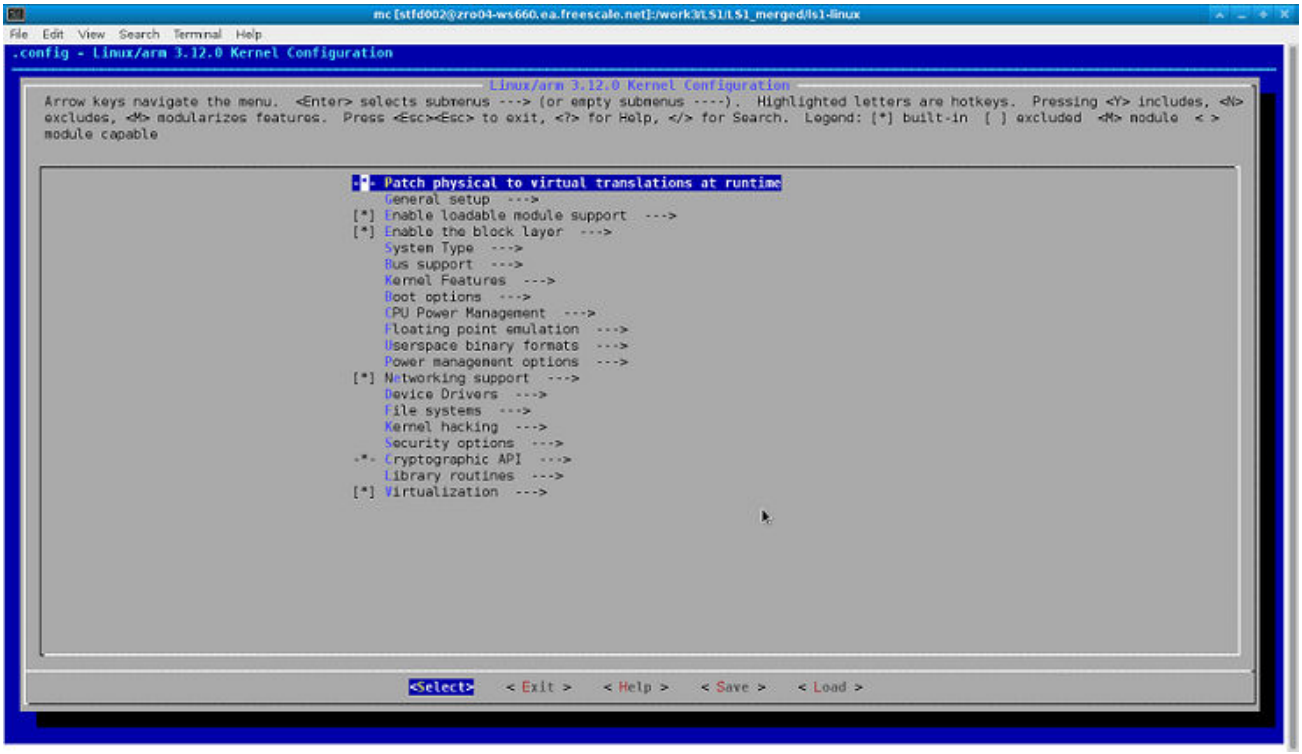


Figure 377.

11.2.6.2.2.3 Quick Start - Recommended Configuration Options

The steps below show all the recommended configuration options to enable to build a kernel with virtual I/O enabled with the same kernel image serving as both host and guest. The sections that follow explain these options in further detail.

Note: The configuration options are enabled by default in the kernel configuration. However, they are are listed here for reference.

1. From the main menuconfig window enable virtualization:

```
[*] Virtualization
```

2. In the virtualization menu enable the following options:

```
[*] Kernel-based Virtual Machine (KVM) support
```

3. Enable network bridging

```
Networking support  --->
    Networking options  --->
        <*> 802.1d Ethernet Bridging
```

4. Enable virtio PCI

```
Device Drivers  --->
    Virtio drivers  --->
      <*> PCI driver for virtio devices
```

5. Enable virtio for block devices

```
Device Drivers  --->
    [*] Block devices  --->
        <*>   Virtio block driver
```

6. Enable virtio for network devices

```
[*] Network core driver support
        <*>   Universal TUN/TAP device driver support
        <*>   Virtio network driver
```

7. Enable vhost for virtio network devices

```
[*] Virtualization
    <*>   Host kernel accelerator for virtio net
```

8. Enable Huge TLB file support

```
File Systems --->
    Pseudo filesystems --->
        [*] Huge TLB file system support
```

9. Enable guest serial support

```
Device Drivers --->
    Character devices --->
        Serial drivers --->
             <*> ARM AMBA PL011 serial port support
             [*]   Support for console on AMBA serial port
```

11.2.6.2.2.4 Host Kernel: Enabling KVM

This section describes the core, basic options needed to enable KVM in the host kernel. KVM is enabled in the host kernel under the virtualization menu of the main kernel menuconfig window.

```
[*] Virtualization
```

Core KVM support is enabled as follows:

```
[*] Kernel-based Virtual Machine (KVM) support
```

11.2.6.2.2.5 Host Kernel: Enabling Virtual Networking

Virtual network interfaces on page 2384 describes how virtual networking can be used to give each VMs a virtual network interface which share physical network interfaces in Linux.

One common approach to configuring virtual networking is for QEMU to use a tun/tap interface bridged to a physical network interface. To do this Ethernet bridging and the kernel's tun/tap features must be enabled in the host kernel:

```
Networking support  --->
          Networking options  --->
                    <*> 802.1d Ethernet Bridging
```

```
Device Drivers  --->
          [*] Network core driver support
                    <*>   Universal TUN/TAP device driver support
```

In order to enable vhost-net, the following config option should ne enabled:

```
[*] Virtualization
    <*>   Host kernel accelerator for virtio net
```

11.2.6.2.2.6 Guest Kernel: Enabling Network and Block Virtual I/O

Virtio is a framework for doing paravirtualized I/O using QEMU/KVM. In order to support communication between guest and hypervisor virtio uses a PCI transport protocol.

Below the kernel configuration options are shown to enable virtio-pci:

```
Device Drivers  --->
    Virtio drivers  --->
      <*> PCI driver for virtio devices
```

Below the kernel configuration options are shown to enable virtio drivers in the Linux kernel to support networking I/O and block (disk) I/O.

```
Device Drivers  --->
          [*] Network device support  --->
                    <*>   Virtio network driver
Device Drivers  --->
          [*] Block devices  --->
                    <*>   Virtio block driver
```

11.2.6.2.2.7 Guest kernel: Enabling console

QEMU emulates an AMBA/PL011 console.

Below the kernel configuration options are shown to enable console:

```
Device Drivers --->
    Character devices --->
        Serial drivers --->
             <*> ARM AMBA PL011 serial port support
             [*]   Support for console on AMBA serial port
```

11.2.6.2.3 Building QEMU

QEMU is a standard package in the QorIQ SDK and will be built for the following Yocto build configurations:

- fsl-image-virt
- fsl-image-full

To add QEMU to another Yocto build configuration, edit the conf/local.conf file and add the IMAGE_INSTALL_append variable:

```
IMAGE_INSTALL_append = " qemu"
```

After a Yocto build is complete, the target root filesystem will contain the following QEMU components that are required for using KVM/QEMU:

```
/usr/bin/qemu-system-arm                # QEMU executable for ARMv7 platforms
/usr/bin/qemu-system-aarch64            # QEMU executable for ARMv8 platforms
```

QEMU can be built as an individual component in the Yocto environment as well, it's package name is "qemu".

To clean and rebuild QEMU from the Yocto build environment:

```
$ bitbake -c clean qemu
$ bitbake qemu
```

11.2.6.2.4 Creating a host Linux root filesystem

11.2.6.2.4.1 Overview

Creating a Linux root filesystem is out of the scope of this document. Please reference the NXP QorIQ SDK for more information on how to create root filesystems with Yocto. This section describes the software components needed on a root filesystem to use KVM/QEMU.

The host root filesystem is the filesystem booted by the host kernel. The host rootfs is distinct from a guest root filesystem which may be needed by certain guest such as Linux.

A host root filesystem capable of running Linux as a guest needs the following components:

- Guest Linux kernel image (Image or zImage). *Note:*Only zImage is supported for ARMv7 platforms.
- QEMU executable
- Guest root filesystem
- Dynamic libraries needed by QEMU (libfdt, libz, glib2.0). These libraries are standard components in a Yocto-created rootfs.

Example host root filesystem layout with the required components to boot a Linux guest (excluding shared libraries):

```
/root/zImage                            # guest Linux kernel
/root/rootfs.ext2.gz                    # guest rootfs
/usr/bin/qemu-system-arm                # QEMU for ARMv7 platforms
/usr/bin/qemu-system-aarch64            # QEMU for ARMv8 platforms
```

11.2.6.2.4.2 Adding Images to a Root Filesystem with Yocto

If using Yocto, as described in Building QEMU on page 2380, the root filesystem produced by the build process will contain QEMU and the example guest device trees provided by the SDK.

A feature is also available with Yocto and the SDK to add arbitrary additional images to the root filesystem. This is done using the merge-files component in Yocto.

Any files and directories copied to the "merge" directory (see path below) will be copied to the root filesystem created by Yocto:

```
meta-freescale/recipes-extended/merge-files/merge-files/merge
```

After populating the merge directory with the desired files, clean and rebuild the rootfs. See example below for the fsl-image-core image type:

```
$ bitbake -c install -f merge-files
$ bitbake merge-files
$ bitbake fsl-image-core
```

See the how-to article Quick-start Steps to Build and Deploy KVM Using Yocto on page 2392 for a more detailed example.

11.2.6.3 Using QEMU and KVM

11.2.6.3.1 Overview of Using QEMU

QEMU is used to start virtual machines and is built and included in the rootfs created by Yocto. The QEMU application is named **qemu-system-arm** (for 32 bit platforms) or **qemu-system-aarch64** (for 64 bit platforms).

In addition to the QEMU executable itself, the following is a list of the minimum components that must be available on the target system to launch a virtual machine using QEMU:

- The host Linux kernel on the target must be built with virtualization support for KVM enabled as described in Building Linux with KVM on page 2376.
- A guest OS kernel image (e.g. zImage or Image for Linux)
- A guest root filesystem (If needed by the guest OS. For example, a Linux guest requires a rootfs.)
- Recommended: A working network interface (to interface to the guest's console and the QEMU monitor)

See the article Quick-start Steps to Run KVM Using Hugetlbfs on page 2394 for an example of how to boot a virtual machine with a rootfs created by Yocto.

The QEMU Emulator User Documentation [1] (see References on page 2376) contains complete documentation for all QEMU command line arguments. The Table below summarizes some of the flags and arguments for basic operation.

Table 591.

Argument	Descriptions
-enable-kvm	Specifies that the Linux KVM should be used for the virtual machine's CPUs
-nographic	Disables graphical output-console will be on emulated serial port.
-M *machine*	Specifies the type of virtual machine. One value is supported: • virt
-smp *cpu_count*	Specifies the number of CPUs for the virtual machine. The number of virtual CPUs allowed is the same as the value of the CONFIG_NR_CPUS config option in the host Linux kernel. To see this value issue the following command from Linux on the target board: `zcat /proc/config.gz | grep NR_CPUS`
-kernel *file*	Specifies the guest OS image. The supported image types are in *Image* format (the generic Linux kernel binary image file) and *zImage* (a compressed version of the Linux kernel image)

Table continues on the next page...

Table 591. (continued)

Argument	Descriptions
-initrd *file*	Specifies a root filesystem image
-append *cmdline*	Use *cmdline* as the guest OS kernel command line (passed in the bootargs property of the /chosen node in the guest device tree)
-serial *dev*	Redirects the virtual serial port to the host device *dev*. QEMU supports many possible host devices. Please refer to the QEMU User Documentation [1] (see References on page 2376) for complete details. Note: if using a tcp device with the **server** option QEMU will wait for a connection to the device before continuing unless the **nowait** option is used.
-m *megs*	Specifies the size of the VM's RAM in megabytes. This option is ignored if using direct mapped memory. See Virtual Machine Memory on page 2384 for further details on options for allocating memory.
-mem-path *path*	Specifies the path to a file from which to allocate memory for the virtual machine. This option should be used to allocate memory from hugetlbfs. See Virtual Machine Memory on page 2384 for further details on options for allocating memory.
-monitor *dev*	Redirects the QEMU monitor to the host device *dev*. QEMU supports many possible host devices. Please refer to the QEMU User Documentation [1] (see References on page 2376) for complete details. Note: if using a tcp device with the **server** option QEMU will wait for a connection to the device before continuing unless the **nowait** option is used.
-S	Do not start CPU at startup (you must type 'c' in the monitor). This can be useful if debugging.
-gdb *dev*	Wait for gdb connection on device dev
-drive *[args]*	Used to create a virtual disk in a virtual machine. See Virtual block devices on page 2385 for additional information.
-netdev *[args]* -device virtio-net-device *[args]*	The -netdev and -device virtio-net-device arguments specify the network backend and front end for createing virtual network devices in virtual machines. See Virtual network interfaces on page 2384 for additional information.
-cpu *model*	Select CPU model. Only one model is supported: • host

Below is an example command line a user would run from the host Linux to start virt virtual machine booting a Linux guest:

ARMv7:

```
qemu-system-arm -enable-kvm -m 512 -nographic -cpu host -machine type=virt -kernel /boot/zImage -
serial tcp::4446,server,telnet -initrd /boot/guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw
console=ttyAMA0 rootwait earlyprintk' -monitor stdio
```

ARMv8:

```
qemu-system-aarch64 -enable-kvm -m 512 -nographic -cpu host -machine type=virt -kernel /boot/Image
-serial tcp::4446,server,telnet -initrd /boot/guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw
console=ttyAMA0 rootwait earlyprintk' -monitor stdio
```

11.2.6.3.2 Virtual Machine Memory

QEMU allocates and loads images into a VM's memory prior to starting the VM. The amount of memory needed for a virtual machine will be dependent on the workload to be run in the VM. There are two ways to allocate memory:

1. Allocation via hugetlbfs

   Hugetlbfs is a Linux mechanism that allows applications to allocate memory backed large physically contiguous regions of memory. QEMU can take advantage of hugetlbfs for allocation of memory for virtual machines, which can provide a significant performance improvement over malloc allocated memory. Hugetlbfs allocated memory provides the flexibility of memory that can be allocated and freed with performance comparable to direct mapped memory.

   The 32 bit ARMv7 implementation in Linux supports 2MB sized huge pages. The 64 bit ARMv8 implementation in Linux supports 2MB and 1GB sized huge pages (for 4K granularity page size).

   The **-mem-path** argument to QEMU specifies the path to the hugetlbfs mount point where the huge pages should be allocated from.

   The **-m** argument to QEMU specifies the amount of memory to allocate to the virtual machine. There are no constraints on the size passed to this argument other than that the amount of memory must fit within the constraints of the system and be enough for the workload in the VM.

   See the how-to article Quick-start Steps to Run KVM Using Hugetlbfs on page 2394 for an example of how to use hugetlbfs.

2. Allocation via malloc

   The default for QEMU is to allocate guest memory by the standard malloc facility available to user space applications in Linux. The amount of memory is specified with the -m command line argument. Malloc'ed memory has the flexibility of being allocated and freed by QEMU as needed. However, malloc'ed memory is backed by 4KB physical pages that are not contiguous and emulation is required by KVM to present a contiguous guest physical memory region to the VM. This approach is discouraged since the emulation can result in a substantial performance penalty for certain workloads.

The guest device tree generated by QEMU will contain a memory node that specifies the total amount of memory.

NOTE

A virtual machine's memory is part of the address space of the QEMU process. This means that the amount of memory allocated to a VM is limited by the standard limits that exist for Linux processes. A 32-bit host kernel has a 2GiB virtual address space used for stack, text, and other data, and this limits the amount of memory that can be allocated to a VM.

11.2.6.3.3 Virtual network interfaces

QEMU provides a number of options for creating virtual network interfaces in virtual machines. Virtual network interfaces are specified using the QEMU command line and guest software sees them as memory mapped devices.

There are two aspects of virtual network interfaces with QEMU:

1. The network “front-end”, which is the network card as seen by the guest. This is specified with the **-device** QEMU argument. The argument to specify a virtio network front end would look like: **-device virtio-net-pci**
2. The network "backend", which connects the network card to some network. Network backend options include user mode networking, a host TAP interface, sockets, or virtual distributed Ethernet. The network backend is specified using the **-netdev** command line argument of QEMU. Note: It is possible to connect two virtual machines using virtual network interfaces. Normally QEMU userspace process emulates I/O accesses from the guest. However, there is an in-kernel implementation: *vhost-net* which puts the data plane emulation code into the kernel.

For example, to use a virtio NIC card with a TAP interface back-end the QEMU command line argument would look like:

```
-netdev tap,id=tap0,script=/root/qemu-ifup -device virtio-net-pci,netdev=tap0
```

The script “/root/qemu-ifup” is a script that QEMU invokes and passes the TAP interface name as an argument. For example, the script could add the TAP interface to an Ethernet bridge.

See the QEMU Users Manual [1] (see References on page 2376) for detailed information about command line options and the types of network interfaces and backends. For best performance, the virtio front-end is recommended.

For additional information about QEMU networking see the references in For More Information on page 2376.

For a detailed example, see the how-to article How to Use Virtual Network Interfaces Using Virtio on page 2396 .

11.2.6.3.4 Virtual block devices

There are a number of approaches to provide a virtual disk to a KVM/QEMU virtual machine. A guest disk image can be a single raw file on the host filesystem, a file in a virtual disk format such as qcow2 and vdi, or a block device on the host Linux system. The virtual disk is assigned on the QEMU command line. In the example below, the file **my_guest_disk** is a disk image and is assigned to the VM when QEMU is launched:

```
-drive file=my_guest_disk,cache=none,if=virtio
```

Refer to the QEMU Users manual [1] (see References on page 2376) for details on the types of virtual disk images that may be created and the related arguments to QEMU.

For a detailed example, see the how to article How to Use Virtual Disks Using Virtio on page 2398.

11.2.6.3.5 Using KVM/QEMU with libvirt

Libvirt is an open source toolkit that enables the management of Linux-based virtualization technologies such as KVM/QEMU virtual machines and Linux containers.

See the chapter ***Libvirt Users Guide*** (section under Linux User Space) for an overview of libvirt and its capabilties, features, and basic usage information.

The libvirt users guide describes the lifecycle of a virtual machine being managed by libvirt. To manage a KVM/QEMU virtual machine as a libvirt domain, it is first defined in an XML file and then a "define" operation (such as **virsh define**) is used to make libvirt aware of the domain. The domain can then be started, which causes QEMU to be invoked to boot the virtual machine. Libvirt tools can then be used to manage the running domain.

Libvirt URIs

To manage KVM/QEMU domains in libvirt, a QEMU URI is used:

- For a local node: *qemu:///system*
- For a remote node: *qemu[+transport]:///[hostname]/system*

See the libvirt users guide for more information on URIs.

Creating a KVM/QEMU Domain

The recommended procedure for creating a KVM/QEMU domain is to:

1. Identify the QEMU command line that fully specifies the desired capabilities and image to boot the virtual machine. Place the command line in a text file, including the full path to the QEMU executable and any images.
2. Convert the QEMU command line to libvirt XML with the *virsh domxml-from-native* command.
3. Use *virsh define* and *virsh start* to define and start the domain.

See Libvirt KVM/QEMU Example (ARM Architecture) on page 2403 for an example of creating and managing a KVM/QEMU domain.

Managing libvirt Domains

See the libvirt users guide for more information on URIs.

A running domain can then be managed by virsh (or other libvirt compatible development tools). Below are some examples of supported management operations:

- *virsh console* - connects to the console of the VM
- *virsh suspend* - suspends a running VM. After *suspend* the domain is in the "paused" state.
- *virsh resume* - resumes a suspended domain. After *resume* the domain is in the "running" state
- *virsh list* - lists active and running VMs
- *virsh reset* - resets the VM
- *qemu-monitor-command* - allows interactions with a VMs QEMU monitor

Libvirt XML

As described above the *virsh domxml-from-native* command is used to convert a QEMU command line into a libvirt XML file.

The full definition of the XML format and all XML tags is specified at: http://libvirt.org/formatdomain.html

11.2.6.3.6 VMs and the Linux Scheduler

Each virtual machine appears to the host Linux as a process with each virtual CPU in the VM implemented as a thread. A VM appears as an instance of QEMU when looking at Linux processes as can be seen in the example below:

```
$ ps -ef
                o
                o
root      1333     1  0 Oct01 ttyS0 00:00:00       -sh
root      1336     2  0 08:24 ?        00:00:00    [kworker/u4:2]
root      1372  1333 18 08:27 ttyS0    00:00:17    qemu-system-arm   -enable-kvm -m
root      1361  1304  0 08:28 ?        00:00:00    sshd: root@pts/0
root      1363  1361  0 08:28 pts/0    00:00:00    -sh
                o
                o
```

CPUs appear as threads. To see thread IDs use the `info cpus` command in the QEMU monitor. Example of a VM with 8 virtual CPUs:

```
(qemu) info cpus
* CPU #0: thread_id=1984
  CPU #1: (halted) thread_id=1985
  CPU #2: (halted) thread_id=1986
  CPU #3: (halted) thread_id=1987
  CPU #4: (halted) thread_id=1988
  CPU #5: (halted) thread_id=1989
```

```
  CPU #6: (halted) thread_id=1990
  CPU #7: (halted) thread_id=1991
```

To see the QEMU threads using the ps command:

```
root@ls_machine:~# ps -eL | grep qemu
 1981  1981 ttyS1    00:00:00 qemu-system-aar
 1981  1982 ttyS1    00:00:00 qemu-system-aar
 1981  1983 ttyS1    00:00:00 qemu-system-aar
 1981  1984 ttyS1    00:00:00 qemu-system-aar
 1981  1985 ttyS1    00:00:00 qemu-system-aar
 1981  1986 ttyS1    00:00:00 qemu-system-aar
 1981  1987 ttyS1    00:00:00 qemu-system-aar
 1981  1988 ttyS1    00:00:00 qemu-system-aar
 1981  1989 ttyS1    00:00:00 qemu-system-aar
 1981  1990 ttyS1    00:00:00 qemu-system-aar
 1981  1991 ttyS1    00:00:00 qemu-system-aar
```

Being a Linux thread means that standard Linux mechanisms can be used to control aspects of how the threads are scheduled relative to other threads/processes. These mechanisms include:

- process priority
- CPU affinity
- isolcpus
- cgroups

11.2.6.4 Virtual machine reference

11.2.6.4.1 VM Overview

In general the architecture of KVM/QEMU is such that few changes should be needed to guest software to run in a VM-- i.e. a full virtualization approach is used, which means that virtual CPUs and virtual I/O devices behave like the physical hardware they are emulating.

However, there are some differences between virtual machines and native hardware that should be considered when targeting an OS to a KVM virtual machine. These differences can be divided into 2 general categories that will be discussed in further detail in this section:

1. Initial state and boot
2. CPUs

11.2.6.4.2 Memory Map of Virtual I/O Devices

The virt virtual machine contains a small subset of the devices found on an SoC. The available devices will be represented in the device tree passed to the guest at boot. See the table below for a summary of the virtual I/O devices in the virt VM:

Table 592.

Virtual I/O Devices for virt machine	
virt VM **Address, size**	**Descriptions**
0,128MB	space for a flash device (this allows running bootrom code)
0x08000000, 0x20000	Virtual CPU peripherals (including GIC distributor and CPU peripheral space)
0x08000000, 0x10000	Virtual GIC distributor
0x08010000, 0x10000	Virtual GIC CPU interface
0x08020000, 0x10000	Virtul GICv2m controller
0x09000000, 0x10000	Virtual UART
0x09010000, 0x10000	Virtual RTC
0x0a000000..0x0a0001ff	Virtual MMIO
...	Virtual MMIO
0x0c000000, 0x02000000	Virtual platform bus
0x10000000, 0x30000000	virtual PCIE
0x40000000, 30G	guest RAM

11.2.6.4.3 Virtual machine state at initialization

11.2.6.4.3.1 Initial State and Boot

When booting the Host, kernel is entered into the HYP mode for ARMv7 respectively EL2 privillege level for ARMv8. After the boot the kernel uses a stub to install KVM and switches back to SVC, respectively EL1. The virtual machine has no virtualization extensions available, so the guest kernel will be entered in SVC mode (ARMv7) respectively EL1 (ARMv8).

In case of a real hardware the boot program will provide some services before giving control to the OS. The necessary steps needed to be done by the bootloader are described in the kernel documentation: *Documentation/arm/Booting/* (ARMv7), *Documentation/arm64/booting.txt* (ARMv8). In case of virtualization, KVM/QEMU makes the necessary actions to put hardware into the initial state (as seen by the guest) and also will take the role of the bootloader and makes the necessary settings. QEMU also installs a very simple bootloader which just set some registers and after that it jumps to the kernel.

It is recommended that a guest OS be minimally device tree aware. The libfdt library (available with the DTC tool) provides a full range of APIs to parse and manipulate device trees and will make the process of adding device tree awareness to an OS straightforward.

11.2.6.4.3.2 Initial State of Virtual CPUs

In a VM with multiple virtual CPUs, CPU #0 is the boot CPU and all other vcpus in the partition are considered secondary. The boot method for the secondary CPUs is PSCI.

The virtual CPU entry conditions comply with the entry conditions specified by *linux/Documentation/arm/Booting* (ARMv7) or *Documentation/ arm64/booting.txt* (ARMv8)

The virtual CPU state is summarized below:

ARMv7:

- R0 = 0

- R1 = machine type number
- R2 = physical address of device tree block (DTB) in system RAM
- MMU off
- data cache off
- CPSR: 0x000001d3 (SVC mode, asynchronous abort, IRQ and FIQ masked)

ARMv8:

- x0 = physical address of device tree blob (dtb) in system RAM.
- x1 = 0
- x2 = 0
- x3 = 0
- MMU off

11.2.6.4.4 Virtual CPUs

11.2.6.4.4.1 Virtual CPU Specification

Software running in a virtual machine sees a virtual CPU that emulates an ARMv7/ARMv8 core without virtualization extensions.

The virtual CPU type will match that of the host hardware platform.

11.2.6.4.4.2 Time in the Virtual CPU

ARM architecture has an optional extension, the generic timers, which provides:

- a counter (*physical counter*) that measures passing of time in real time
- a timer (*physical timer*) for each CPU. The timer is programmed to raise an interrupt to the CPU after a certain amount of time has passed.

The generic timers include virtualization support by introducing:

- a new counter, the *virtual counter*
- a new timer, the *virtual timer*.

This allows the virtual machine to have direct access to reading (virtual) counters and programming (virtual) timers without trapping.

KVM uses the physical timers in the host, the virtual machine access to the physical timers being disabled.

The virtual machine accesses the virtual timer and can, in this way, directly access the timer hardware without trapping to the hypervisor. However, the virtual timers do not raise virtual interrupts, but hardware interrupts which trap to the hypervisor. KVM injects a corresponding virtual interrupt into the VM when it detects that the virtual timer expired.

11.2.6.4.5 VGIC

The ARM Generic Interrupt Controller (GIC) provides hardware support for virtualization. The guest is able to mask, acknowledge and EOI interrupts without trapping to the hypervisor. However, there is a central part of the GIC called distributor which is responsible for interrupt prioritization and distribution to each CPU which does not provide virtualization extensions and for this part KVM provides an in-kernel emulation. Also, all the physical interrupts cannot be directly received by the guest. Instead, the KVM will program a virtual interrupt which will be raised in the guest. But, with the virtualization support in the GIC controller, when the guest is ACK-ing and EOI-ing the virtual interrupt, there is no need to trap into KVM.

11.2.6.5 Debugging virtual machines

11.2.6.5.1 QEMU Monitor

When starting QEMU, a monitor shell is available that can be used to control and see the state of VM. By default this monitor is started in the Linux shell where QEMU is invoked.

See example below of the output when starting QEMU. The user can interact with the monitor at the (qemu) prompt.

```
QEMU 2.4.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server
```

The monitor can also be exposed over a network port by using the `-monitor dev` command line option. See Overview of Using QEMU on page 2382 and the QEMU user's manual [1] (see References on page 2376).

Refer to the QEMU user's manual [1] for a complete listing of the monitor commands available. Below is a list of some useful commands supported in the NXP SDK implementation of QEMU:

- **help** - lists all the available commands with usage information
- **info cpus** - displays the state and thread ID of all virtual CPUs
- **info** *registers* - displays the contents of the default vcpu's registers
- **cpu cpu_number** - sets the default vcpu number
- **system_reset** - resets the VM
- **x**/*fmt addr* -- virtual memory dump starting at 'addr'
- **xp**/*fmt addr* -- physical memory dump starting at 'addr'

11.2.6.5.2 QEMU GDB Stub

QEMU supports debugging of a VM using gdb. QEMU contains a gdb stub that can be attached to from a host system and allows standard source level debugging capabilities to examine the state of the VM and do run control.

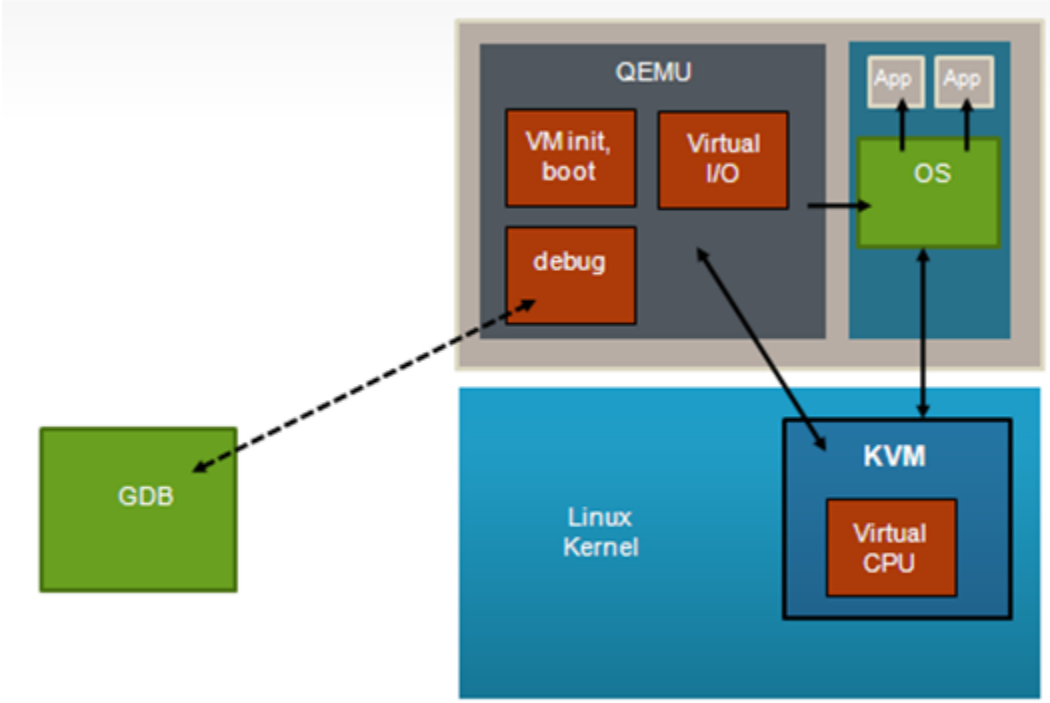


Figure 378.

To use the gdb stub, start QEMU with the -gdb *dev* option where dev specifies the type of connection to be used. See the QEMU user's manual [1] (see References on page 2376) for details.

One useful option when debugging is the -S argument to QEMU which causes QEMU to wait to start the first instruction of the guest until told to start using the monitor (**continue** command).

In the example below the tcp device type is used. A gdb stub will be active on port 4445 of the host Linux kernel when starting QEMU:

```
$ qemu-system-arm -enable-kvm -m 512 -mem-path /var/lib/hugetlbfs/pagesize-2MB -nographic -cpu
host -machine type=virt -kernel /boot/zImage  -serial tcp::4446,server,telnet -initrd /boot/fsl-
image-core-ls1021atwr.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -
monitor stdio -gdb tcp::4445

QEMU 2.4.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: telnet:0.0.0.0:4446,server
```

After the guest has been started normally, gdb can be used to connect to the VM (in this example the host kernel has an ip address of 192.168.3.30):

```
(gdb) target remote 192.168.3.30:4445
Remote debugging using 192.168.3.30:4445
0x80024f44 in ?? ()
```

Debugging with gdb can then proceed normally:

```
(gdb) p/x $r15
$2 = 0x80024f44
(gdb)
```

11.2.6.6 KVM/QEMU How-to's

11.2.6.6.1 Quick-start Steps to Build and Deploy KVM Using Yocto

The following steps show how to build host and guest root filesystems, Linux kernel, and QEMU in the Yocto environment.

There are two possibilities to deploy KVM using Yocto:

- Using the preconfigured **fsl-image-virt** Yocto image as the base for host root filesystem. This image type will generate a host root filesystem which contains: QEMU, guest root filesystem and guest kernel image.
- Using other Yocto images as the base for host rootfilesystem . If the user wants to use other Yocto images to enable KVM there will be additional steps needed in order to add all the needed pieces into the host root filesystem: QEMU, guest root filesystem, guest kernel image.

11.2.6.6.1.1 Deploy KVM using fsl-image-virt Image

The host roofs is based on the **fsl-image-virt** image type and the guest rootfs is based on **fsl-image-core**.

The steps outlined below assume that the Yocto build environment is configured so that the **bitbake** command can be run.

1. Build the base host root filesystem using the **fsl-image-virt** image type.

   ```
   $ bitbake fsl-image-virt
   ```

2. Build a kernel with KVM enabled. In this case the same kernel image will be used for both host and guest.

   Configure the Linux kernel to enable KVM-related features (if not enabled by default in the kernel config)

   ```
   $ bitbake -c menuconfig virtual/kernel
   ```

   Follow the steps described in section Quick Start - Recommended Configuration Options on page 2378 to enable KVM in the Linux kernel.

   Then rebuild the kernel based on the new configuration options:

   ```
   $ bitbake virtual/kernel
   ```

3. Re-build the fsl-image-virt image (if nothing was done at step 2, this step may be skipped)

   ```
   bitbake fsl-image-virt
   ```

4. Create the kernel.itb file (if necessary)

   U-boot may use both uImage or FIT image format to load the kernel image. For ARMv7 platforms an uImage is used and for these platforms this step is not needed. For ARMv8, a FIT image is used and the FIT image must be generated. The following steps need to be taken:

   - Update the *fsl-image-kernelitb* to include the rootfs generated by the *fsl-image-virt* (by default the recipee would use the rootfs generated by *fsl-image-core*). In order to change the rootfs type in the generated itb file there are two possible solutions:
     - Redefine the *ROOTFS_IMAGE* variable in *meta-freescale/recipes-fsl/images/fsl-image-kernelitb.bb* directly:

       ```
       -ROOTFS_IMAGE ?= "fsl-image-core"
       +ROOTFS_IMAGE ?= "fsl-image-virt"
       ```

     - Add the following line to *build_<machine_release>/conf/local.conf*:

       ```
       ROOTFS_IMAGE = "fsl-image-virt”
       ```

- Generate the kernel itb:

```
bitbake fsl-image-kernelitb
```

The resulting host rootfs will contain:

- Linux kernel image. Currently for ARMv7 paltforms the zImage file will be included and for ARMv8 platforms the Image file. The kernel image will be located in */boot*
- Guest root filesystem: based on **fsl-image-core**. The guest root file system will be located in */boot*
- QEMU

For steps to run QEMU see the article Quick-start Steps to Run KVM Using Hugetlbfs on page 2394.

11.2.6.6.1.2 Deploy KVM using fsl-image-core Image

The host roofs is based on the **fsl-image-core** image type and the guest rootfs is based on **fsl-image-minimal**.

The steps outlined below assume that the Yocto build environment is configured so that the **bitbake** command can be run.

1. Build the base host root filesystem using the **fsl-image-core** image type.

```
$ bitbake fsl-image-core
```

2. Build a host kernel with KVM-enabled (if not enabled by default in the kernel config)

   Configure the Linux kernel to enable KVM-related features.

```
$ bitbake -c menuconfig virtual/kernel
```

   Follow the steps described in section Quick Start - Recommended Configuration Options on page 2378 to enable KVM in the Linux kernel.

   Then rebuild the kernel based on the new configuration options:

```
$ bitbake virtual/kernel
```

3. Add QEMU to the packages built by **fsl-image-core**.

   Edit the `conf/local.conf` file and append the following line which adds the QEMU package:

```
IMAGE_INSTALL_append = " qemu"
```

4. Build a guest root filesystem and add it to the host rootfs.

   A. Build the guest roofs using the **fsl-image-minimal** image type. This creates a small rootfs sufficient for booting a Linux guest:

```
$ bitbake fsl-image-minimal
```

   This command results in the minimal rootfs being created in tmp/deploy/images. In this example the minimal rootfs built is named: fsl-image-minimal.rootfs.ext2.gz

   B. Use the merge-files feature of Yocto (see Adding Images to a Root Filesystem with Yocto on page 2381) to copy the guest rootfs to the host rootfs.

```
$ mkdir -p meta-freescale/recipes-extended/merge-files/merge-files/merge/home/root
```

```
$ cp tmp/deploy/images/machine/fsl-image-minimal.rootfs.ext2.gz meta-freescale/recipes-extended/
merge-files/merge-files/merge/home/root/guest.rootfs.ext2.gz

$ bitbake -c install -f merge-files

$ bitbake merge-files
```

5. Re-build the fsl-image-core image.

```
bitbake fsl-image-core
```

6. Create the kernel.itb file (if necessary)

   U-boot may use both uImage or FIT image format to load the kernel image. For ARMv7 platforms an uImage is used and for those platforms this step is not needed. For ARMv8, a FIT image is used and the FIT image must be generated.

```
bitbake fsl-image-kernelitb
```

   The resulting host rootfs will contain:

   - Linux kernel image
   - Guest root filesystem
   - QEMU, including the example guest device trees

For steps to run QEMU see the article Quick-start Steps to Run KVM Using Hugetlbfs on page 2394.

11.2.6.6.2 Quick-start Steps to Run KVM Using Hugetlbfs

The pre-requisite to this example is completing the steps in the article Quick-start Steps to Build and Deploy KVM Using Yocto on page 2392.

This example assumes that the host Linux kernel is booted, has a working network interface, and the following images are present in the host root filesystem:

- Guest kernel image (*/boot/zImage* or */boot/Image*)
- Guest root filesystem (*/boot/guest.rootfs.ext2.gz*)
- QEMU (*/usr/bin/qemu-system-arm* or */usr/bin/qemu-system-aarch64*)

There are a number of mechanisms for allocating huge pages and making them accessible via a mount point. Refer to the SDK documentation for details. This example assume allocating pages using the hugeadm command. Create a 512MB pool of 2MB huge pages, which can be used by QEMU for allocating VM memory:

```
$ hugeadm --pool-pages-min 2M:256

 hugeadm --pool-list
 Size  Minimum  Current  Maximum  Default
   2097152      256      256      256        *
```

Create a mount point to access the huge pages:

```
$ hugeadm --create-mounts

$  ls -1 /var/lib/hugetlbfs/
pagesize-2MB
```

Start QEMU specifying the 2MB huge page pool as the file from which to allocate memory. In this example 512MB of memory is allocated to the VM:

32 bit ARMv7:

```
qemu-system-arm -enable-kvm -m 512 -mem-path /var/lib/hugetlbfs/pagesize-2MB -nographic -cpu host -
machine type=virt -kernel /boot/zImage  -serial tcp::4446,server,telnet -initrd /boot/
guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio

QEMU waiting for connection on: telnet::0.0.0.04446,server
```

64bit ARMv8:

```
qemu-system-aarch64 -enable-kvm -m 512 -mem-path /var/lib/hugetlbfs/pagesize-2MB -nographic -cpu
host -machine type=virt -kernel /boot/Image  -serial tcp::4446,server,telnet -initrd /boot/
guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -monitor
stdio

QEMU waiting for connection on: telnet::0.0.0.04446,server
```

Explanation of the command line options:

- **-enable-kvm** : specifies that KVM should be used
- **-m 512** : the amount of memory for the VM
- **-mem-path /var/lib/hugetlbfs/pagesize-2MB** : allocates from hugetlbfs based memory
- **-nographic** : don't instantiate a graphics card, this is the only option supported for the SDK
- **-cpu host** : the type of the CPU. In this case it is the same as the host CPU
- **-machine type=virt** : the type of virtual machine
- **-kernel /boot/zImage** : name of guest Linux kernel
- **-initrd ./boot/guest.rootfs.ext2.gz** : name of guest roofs
- **-append "root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk"** : guest Linux bootargs
- **-serial tcp::4446,server,telne** : provide an emulated serial port (telnet server) on port 4444 on the host Linux system. Default behavior will be for QEMU to wait until the user connects to this port before booting the VM.
- **-monitor stdio** : start QEMU monitor

At this point QEMU is waiting for a telnet connection to the virtual machine's console (port 4446 of the target board) prior to starting the virtual machine.

Connect to QEMU via telnet to start the virtual machine booting. In this example the target board has IP address 192.168.3.30.

```
$ telnet 192.168.3.30 4446
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 4.1.8-rt8+gf42311c (gcc version 4.9.3 20150311 (prerelease) (Linaro
GCC 4.9-2015.03) ) #1 SMP Tue Apr 26 15:40:46 EEST 2016
[    0.000000] CPU: AArch64 Processor [411fd071] revision 1
[    0.000000] Detected PIPT I-cache on CPU0
[    0.000000] alternatives: enabling workaround for ARM erratum 832075
[    0.000000] alternatives: enabling workaround for ARM erratum 834220
[    0.000000] efi: Getting EFI parameters from FDT:

.........
Starting system log daemon...0
Starting kernel log daemon...0
Starting internet superserver: xinetd.

ls_machine login:
```

11.2.6.6.3 How to Use Virtual Network Interfaces Using Virtio

As discussed in Virtual network interfaces on page 2384, there are two aspects of virtual network interfaces-- 1) the "front end" (the device as seen by the guest OS) and 2) the "backend" (the means by the virtual device is connected to the network).

This example uses a "virtio" model NIC card and a tap network backend. The virtual network interface is bridged via a TAP interface to the physical network. The guest OS is Linux.

When starting QEMU we will add the following arguments to create the virtual network interface:

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=no,ifname="tap0" -device virtio-net-
pci,netdev=tap0
```

Perform the following steps:

1. Enable virtio networking in the host and guest Linux kernels (see Host Kernel: Enabling Virtual Networking on page 2379 and Guest Kernel: Enabling Network and Block Virtual I/O on page 2380).
2. On the host Linux create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine using the **brctl** command. In this example the physical interface being used is eth2:

   ```
   brctl addbr br0
   ifconfig br0 192.168.3.30 netmask 255.255.248.0
   ifconfig eth2 0.0.0.0
   brctl addif br0 eth2
   ```

3. Create a qemu-ifup script on the host Linux system. For the TAP backend type, when QEMU creates the virtual network interface it invokes a user-created script that allows customization of how the TAP interface is to be handled. The name of the TAP interface created by QEMU is passed as an argument. In this example we will bridge the the TAP inteface to the bridge created in step #2. See the example qemu-ifup script below:

   ```
   #!/bin/sh
   # TAP interface will be passed in $1
   bridge=br0
   guest_device=$1
   ifconfig $guest_device 0.0.0.0 up
   brctl addif $bridge $guest_device
   ```

4. When starting QEMU specify that the network device type is "virtio" and specify the path to the script created in step #3:

   ```
   qemu-system-aarch64 -smp 8 -enable-kvm -m 4096 -nographic -cpu host -machine type=virt -kernel /
   boot/Image  -serial tcp::4446,server,telnet -initrd /boot/fsl-image-core-xxxx.ext2.gz -append
   'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio -netdev
   tap,id=tap0,script=qemu-ifup,downscript=no,ifname="tap0" -device virtio-net-pci,netdev=tap0
   ```

5. In the guest OS the virtual network interface will appear and can be brought up and assigned an IP address in the normal way. In the example below (the commands are run from the guest command shell) the virtio interface is eth0.

   ```
   $ ip addr
   1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
       link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
       inet 127.0.0.1/8 scope host lo
          valid_lft forever preferred_lft forever
       inet6 ::1/128 scope host
          valid_lft forever preferred_lft forever
   2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
       link/ether 52:54:00:12:34:56 brd ff:ff:ff:ff:ff:ff
   ```

```
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default
    link/sit 0.0.0.0 brd 0.0.0.0

root@ls_machine:~# ethtool -i eth0
driver: virtio_net
version: 1.0.0
firmware-version:
expansion-rom-version:
bus-info: 0000:00:01.0
supports-statistics: no
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no

$ ifconfig eth0 192.168.3.31 netmask 255.255.248.0
```

11.2.6.6.4 How to use vhost-net with virtio

vhost-net is a character device that can be used to reduce the number of system calls involved in virtio networking. vhost-net moves network packets between the guest and the host system using the Linux kernel, bypassing QEMU.

In order to use vhost-net perform the following steps:

1. Enable virtio networking and vhost-net in the host and guest Linux kernels (see Host Kernel: Enabling Virtual Networking on page 2379 and Guest Kernel: Enabling Network and Block Virtual I/O on page 2380).
2. On the host Linux create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine using the **brctl** command. In this example the physical interface being used is eth2:

   ```
   brctl addbr br0
   ifconfig br0 192.168.3.30 netmask 255.255.248.0
   ifconfig eth2 0.0.0.0
   brctl addif br0 eth2
   ```

3. Create a qemu-ifup script on the host Linux system. For the TAP backend type, when QEMU creates the virtual network interface it invokes a user-created script that allows customization of how the TAP interface is to be handled. The name of the TAP interface created by QEMU is passed as an argument. In this example we will bridge the the TAP inteface to the bridge created in step #2. See the example qemu-ifup script below:

   ```
   #!/bin/sh
   # TAP interface will be passed in $1
   bridge=br0
   guest_device=$1
   ifconfig $guest_device 0.0.0.0 up
   brctl addif $bridge $guest_device
   ```

4. When starting QEMU specify that the network device type is "virtio" and that vhost-net (**vhost=on** parameter) is used:

   ```
   qemu-system-aarch64 -smp 8 -enable-kvm -m 8192 -mem-path /var/lib/hugetlbfs/pagesize-1GB -
   nographic -cpu host -machine type=virt -kernel /boot/Image  -serial tcp::4446,server,telnet -
   initrd /boot/fsl-image-core-xxxx.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait
   earlyprintk ramdisk_size=307200' -monitor stdio -netdev tap,id=tap0,script=qemu-
   ifup,downscript=no,ifname="tap0",vhost=on -device virtio-net-pci,netdev=tap0
   ```

5. In the guest the virtual interface will come up as described in How to Use Virtual Network Interfaces Using Virtio on page 2396. In the Host kernel the vhost thread can be seen consuming CPU:

```
 2078 root      20   0       0      0      0 R  93.7  0.0   0:07.09
vhost-2066
 2066 root      20   0 9192660 511632   7532 S  82.0  3.3   0:12.70 qemu-system-
aar
 2091 root      20   0  159636   1092    960 S  68.0  0.0   0:05.50 iperf
```

11.2.6.6.5 How to Use Virtual Disks Using Virtio

As discussed in Virtual block devices on page 2385, there are a number of formats available for virtual disk images.

The example below uses a raw file. The steps below go through the process of creating a virtual disk image, assigning it to a VM, partitioning the disk, creating a filesystem on it, and mounting it.

1. On the host Linux, create a binary image to represent the guest disk. For example to create a 16MB disk:

```
$ dd if=/dev/zero of=my_guest_disk bs=4K count=4K
```

2. Start QEMU, specifying the name of the virtual disk file for the -drive argument:

```
qemu-system-aarch64 -enable-kvm -m 512 -mem-path /var/lib/hugetlbfs/pagesize-2MB -nographic  -
cpu host -machine type=virt -kernel /boot/Image  -serial tcp::4446,server,telnet -initrd /boot/
guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0  rootwait earlyprintk' -monitor
stdio -drive if=none,file=my_guest_disk,cache=none,id=foo,format=raw -device virtio-blk-
pci,drive=foo
```

3. After the guest has booted the virtual disk is visible as a block device in /dev with the name vda, vdb, etc.

```
$ ls -l /dev/vda
brw-rw---- 1 root disk 254, 0 Jan  1  1970 /dev/vda
```

   A virtual block device can be treated like any other hard disk. It can be partitioned, formatted, and mounted.

4. Configure a partition on the disk with fdisk:

```
$ fdisk /dev/vda
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.
Command (m for help):
```

   Display the partition table:

```

Command (m for help): p

Disk /dev/vda: 16 MB, 16777216 bytes, 32768 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0xa22daf58

   Device Boot      Start         End      Blocks   Id  System
```

Create a new partition:

```
Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-32767, default 2048): 2048
Last sector, +sectors or +size{K,M,G} (2048-32767, default 32767):
Using default value 32767
Partition 1 of type Linux and of size 15 MiB is set
```

Display the new partition:

```
Disk /dev/vda: 16 MB, 16777216 bytes, 32768 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0xa22daf58

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1            2048       32767       15360   83  Linux
```

Write the partition table to disk and exit:

```
Command (m for help): w
The partition table has been altered!
Calling ioctl() to re-read partition table
 vda: vda1
```

5. Create a filesystem on the new partition:

```
$ mkfs.ext3 /dev/vda1
mke2fs 1.42.8 (20-Jun-2013)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
3840 inodes, 15360 blocks
768 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=15728640
2 block groups
8192 blocks per group, 8192 fragments per group
1920 inodes per group
Superblock backups stored on blocks:
    8193

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
```

6. Mount the filesystem:

```
$ mount /dev/vda1 /mnt/src
kjournald starting.  Commit interval 5 seconds
EXT3-fs (vda1): using internal journal
EXT3-fs (vda1): mounted filesystem with ordered data mode
```

11.2.6.6.6 How to use virtual disks using virtio-blk-dataplane

Virtio-blk-dataplane was developed for high performance disk I/O, especially for high IOPS devices. The QEMU performs the disk I/O in a dedicated thread that is optimized for I/O performance.

In this example a real disk is used, a block device on the Linux host.

1. Start QEMU:

```
qemu-system-aarch64 -enable-kvm -nographic -machine type=virt -cpu host -kernel /boot/Image -
append "root=/dev/ram rw console=ttyAMA0,115200 ramdisk_size=1000000" -serial tcp::
4444,server,telnet -initrd /boot/fsl-image-core-xxxx.ext2.gz -m 8192 -mem-path /var/lib/
hugetlbfs/pagesize-1GB -object iothread,id=iothread0 -drive
if=none,id=drive0,cache=none,format=raw,file=/dev/sda,aio=native -device virtio-blk-
pci,drive=drive0,scsi=off,iothread=iothread0
```

2. After the guest boots, the virtual disk is visible as a block device with the name vda, vdb, etc.

```
# fdisk -l
Disk /dev/vda: 238.5 GiB, 256060514304 bytes, 500118192 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x000031e0
Device Boot Start End Blocks Id System
/dev/vda1 * 2048 1050623 524288 83 Linux
/dev/vda2 1050624 9439231 4194304 82 Linux swap / Solaris
/dev/vda3 9439232 500117503 245339136 83 Linux
```

In this case the disk has 3 partitions. The partitions can be mounted and used.

11.2.6.6.7 Debugging: How to Examine Initial Virtual Machine State with QEMU

It can be helpful when debugging to examine the state of the virtual machine prior to executing the first instruction of the guest OS.

To do this, start QEMU with the -S option.

Example:

```
qemu-system-aarch64 -enable-kvm -m 512 -nographic -cpu host -machine type=virt -kernel /boot/
Image  -serial tcp::4446,server,telnet -initrd /boot/fsl-image-core-xxxx.ext2.gz -append
'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio -S
```

The console was started with the "-serial tcp::4446,server,telnet" option so QEMU waits for a connection prior to starting initialization. Use telnet to connect to port 4446 of the target.

At this point QEMU initializes the VM, but does not execute the entry point to the guest OS. The monitor prompt can now be used to examine initial state:

```
QEMU 2.4.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: telnet:0.0.0.0:4446,server
(qemu)
```

To see where boot images are loaded and placed by QEMU use the **info roms** command:

```
(qemu) info roms
addr=0000000040000000 size=0x000028 mem=ram name="bootloader"
addr=0000000040080000 size=0xb9a800 mem=ram name="/boot/Image"
addr=0000000048000000 size=0x1517eef mem=ram name="/boot/fsl-image-core-xxxx.ext2.gz"
addr=0000000049600000 size=0x010000 mem=ram name="dtb"
/rom@etc/acpi/tables size=0x200000 name="etc/acpi/tables"
/rom@etc/table-loader size=0x000880 name="etc/table-loader"
/rom@etc/acpi/rsdp size=0x000024 name="etc/acpi/rsdp"
```

A trivial bootloader is loaded at the start of guest memory at 0x40000000

The kernel image (zImage) is loaded at 0x40080000. The ramdisk is loaded at 0x48000000.

To examine the initial state of registers use the info registers command:

```
(qemu) info registers
PC=0000000040000000  SP=0000000000000000
X00=0000000000000000 X01=0000000000000000 X02=0000000000000000 X03=0000000000000000
X04=0000000000000000 X05=0000000000000000 X06=0000000000000000 X07=0000000000000000
X08=0000000000000000 X09=0000000000000000 X10=0000000000000000 X11=0000000000000000
X12=0000000000000000 X13=0000000000000000 X14=0000000000000000 X15=0000000000000000
X16=0000000000000000 X17=0000000000000000 X18=0000000000000000 X19=0000000000000000
X20=0000000000000000 X21=0000000000000000 X22=0000000000000000 X23=0000000000000000
X24=0000000000000000 X25=0000000000000000 X26=0000000000000000 X27=0000000000000000
X28=0000000000000000 X29=0000000000000000 X30=0000000000000000 PSTATE=400003c5 (flags -Z--)

q00=00000000000000000:0000000000000000 q01=00000000000000000:0000000000000000
q02=00000000000000000:0000000000000000 q03=00000000000000000:0000000000000000
q04=00000000000000000:0000000000000000 q05=00000000000000000:0000000000000000
q06=00000000000000000:0000000000000000 q07=00000000000000000:0000000000000000
q08=00000000000000000:0000000000000000 q09=00000000000000000:0000000000000000
q10=00000000000000000:0000000000000000 q11=00000000000000000:0000000000000000
q12=00000000000000000:0000000000000000 q13=00000000000000000:0000000000000000
q14=00000000000000000:0000000000000000 q15=00000000000000000:0000000000000000
q16=00000000000000000:0000000000000000 q17=00000000000000000:0000000000000000
q18=00000000000000000:0000000000000000 q19=00000000000000000:0000000000000000
q20=00000000000000000:0000000000000000 q21=00000000000000000:0000000000000000
q22=00000000000000000:0000000000000000 q23=00000000000000000:0000000000000000
q24=00000000000000000:0000000000000000 q25=00000000000000000:0000000000000000
q26=00000000000000000:0000000000000000 q27=00000000000000000:0000000000000000
q28=00000000000000000:0000000000000000 q29=00000000000000000:0000000000000000
q30=00000000000000000:0000000000000000 q31=00000000000000000:0000000000000000
FPCR: 00000000  FPSR: 00000000
```

The program counter is set to 0x40000000 which is the effective address of the entry point of the kernel.

11.2.6.6.8 Debugging: How to Profile Virtualization Overhead with KVM

Running software in a virtual machine can cause additional overhead that affects performance. The virtualization overhead is directly related to the number of times the hypervisor (KVM) is invoked to handle exception conditions that may occur in the virtual machine. These exception handling events are referred to as 'exits', because guest context is exited.

Examples of exits include things such the guest executing a privileged instruction, access a privileged CPU register, accessing a virtual I/O device, or a hardware interrupt such as a decrementer interrupt.

The type and number of exits that occur is workload dependent.

KVM implements a mechanism in which different events are logged. These events are actually tracepoint events, and perf nicely integrates with them. You have to compile the host kernel with the following options:

```
Kernel hacking  --->
     [*] Tracers  --->
         [*]   Trace process context switches and events
```

Counting Events

A count of a subset of KVM events that occur can be seen under debugfs. To see this first mount debugfs:

```
mount -t debugfs none /sys/kernel/debug
```

The statistics can be seen using perf tool:

```
# perf stat -e "kvm:*" -p 1395
^C
 Performance counter stats for process id '1395':

              5678 kvm:kvm_entry
              5678 kvm:kvm_exit
              3121 kvm:kvm_guest_fault
              2278 kvm:kvm_irq_line
                 0 kvm:kvm_mmio_emulate
                 0 kvm:kvm_emulate_cp15_imp
              2438 kvm:kvm_wfi
                 0 kvm:kvm_unmap_hva
                 2 kvm:kvm_unmap_hva_range
                 0 kvm:kvm_set_spte_hva
                 0 kvm:kvm_hvc
              3119 kvm:kvm_userspace_exit
                 0 kvm:kvm_set_irq
                 0 kvm:kvm_ack_irq
              4068 kvm:kvm_mmio
                 0 kvm:kvm_fpu
                 0 kvm:kvm_age_page

      59.316709040 seconds time elapsed
```

Tracing events

Detailed traced can be generated using ftrace:

```
[enable ftrace in kernel: events and system calls]
$echo 1 > /sys/kernel/debug/tracing/events/kvm/enable
$cat /sys/kernel/debug/tracing/trace_pipe

qemu-system-arm-1366  [000] ....   716.115891: kvm_guest_fault: ipa 0x9000000, hsr 0x93430046,
hxfar 0xa084c030, pc 0x80266a9c
```

```
qemu-system-arm-1366  [000] ....   716.115892: kvm_mmio: mmio write len 2 gpa 0x9000030 val 0xf01
qemu-system-arm-1366  [000] ....   716.115895: kvm_userspace_exit: reason KVM_EXIT_MMIO (6)
qemu-system-arm-1366  [000] d...   716.115907: kvm_entry: PC: 0x80266aa0
qemu-system-arm-1366  [000] d...   716.116234: kvm_exit: PC: 0x800cf508
qemu-system-arm-1366  [000] d...   716.118274: kvm_entry: PC: 0x800cf508
qemu-system-arm-1366  [000] d...   716.118704: kvm_exit: PC: 0x0000981c
qemu-system-arm-1366  [000] d...   716.120737: kvm_entry: PC: 0x0000981c
qemu-system-arm-1366  [000] d...   716.121159: kvm_exit: PC: 0x800bb104
qemu-system-arm-1366  [000] d...   716.123197: kvm_entry: PC: 0x800bb104
qemu-system-arm-1366  [000] d...   716.123620: kvm_exit: PC: 0x8009cae0
qemu-system-arm-1366  [000] d...   716.125696: kvm_entry: PC: 0x8009cae0
qemu-system-arm-1366  [000] d...   716.126091: kvm_exit: PC: 0x800c90f4
qemu-system-arm-1366  [000] d...   716.128130: kvm_entry: PC: 0x800c90f4
qemu-system-arm-1366  [000] d...   716.128561: kvm_exit: PC: 0x801f37f4
qemu-system-arm-1366  [000] d...   716.130594: kvm_entry: PC: 0x801f37f4
qemu-system-arm-1366  [000] d...   716.130623: kvm_exit: PC: 0x8020576c
qemu-system-arm-1366  [000] d...   716.130635: kvm_entry: PC: 0x8020576c
qemu-system-arm-1366  [000] d...   716.131018: kvm_exit: PC: 0x43014750
qemu-system-arm-1366  [000] d...   716.133053: kvm_entry: PC: 0x43014750
qemu-system-arm-1366  [000] d...   716.133478: kvm_exit: PC: 0x80205778
qemu-system-arm-1366  [000] d...   716.135555: kvm_entry: PC: 0x80205778
```

11.2.6.6.9 Libvirt KVM/QEMU Example (ARM Architecture)

The following example shows the lifecycle of a simple KVM/QEMU libvirt domain called **kvm**. In this example the default URI is qemu:///system, and because of this default an explicit URI is not used in the virsh commands

Libvirt has the possibility to convert qemu command line arguments in xml. However, the current version of libvirt does not have full support for all qemu arguments. It can be used to generate a basic xml file, but there is currently a problem that it generates a default USB node which should be removed.

1. We begin with a simple QEMU command line in a text file named **kvm.args**:

- 32 bit ARMv7:

```
echo "/usr/bin/qemu-system-arm -name kvm -smp 2 -enable-kvm -m 512  -nographic -cpu host -machine
type=virt -kernel /boot/zImage  -serial pty -initrd /boot/fsl-image-core-ls1021atwr.ext2.gz -
append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' " > kvm.args
```

- 64 bit ARMv8:

```
echo "/usr/bin/qemu-system-aarch64 -name kvm -smp 2 -enable-kvm -m 512 -nographic -cpu host -
machine type=virt -kernel /boot/Image  -serial pty -initrd /boot/guest.rootfs.ext2.gz -append
'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' " > kvm.args
```

NOTE

The serial console is a tty, not a telnet server. The -name option is required and specifies the name of the virtual machine.

2. Before defining the domain the QEMU command line must be converted to libvirt XML format:

```
virsh domxml-from-native qemu-argv kvm.args > kvm.xml
```

NOTE

For ARMv7 paltforms the generated xml file has to be manually changed to remove the USB node.

The content of the newly created domain XML file is shown below:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>12a3391e-9cd4-43dd-b8b7-27b1fb193378</uuid>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type machine='virt'>hvm</type>
    <kernel>/boot/zImage</kernel>
    <initrd>/boot/fsl-image-core-ls1021atwr.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-arm</emulator>
    <controller type='usb' index='0'/>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
  </devices>
 </domain>
```

A more complex example including devices can be found bellow.

The example contains two devices:

- a virtio network interface
- a virtio block device (disk)

NOTE

This example will use MMIO as transport for virtio. Currently libvirt has no support for PCI transport, but it can be used using passthrough QEMU command line arguments (see the next example)

Device example (using virtio with MMIO as transport):

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>12a3391e-9cd4-43dd-b8b7-27b1fb193378</uuid>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type machine='virt'>hvm</type>
    <kernel>/boot/zImage</kernel>
    <initrd>/boot/fsl-image-core-ls1021atwr.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
```

```
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-arm</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>

    <interface type='ethernet'>
      <mac address='52:54:00:b0:39:28'/>
      <script path='/home/root/qemu-ifup'/>
    </interface>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw' cache='none'/>
      <source file='/home/root/my_guest_disk'/>
      <target dev='vda' bus='virtio'/>
    </disk>

  </devices>
  <qemu:commandline>
    <qemu:arg value='-mem-path'/>
    <qemu:arg value='/var/lib/hugetlbfs/pagesize-2MB'/>
  </qemu:commandline>
</domain>
```

Device example (using virtio with PCI as transport):

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>faa89ddc-e156-46c0-9d0f-029c322f9bf7</uuid>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <vcpu placement='static'>4</vcpu>
  <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <kernel>/boot/Image</kernel>
    <initrd>/images/fsl-image-core-ls1043ardb.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-aarch64</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
```

```
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='none'/>
  </devices>
  <qemu:commandline>

    <qemu:arg value='-netdev'/>
    <qemu:arg value='tap,id=tap0,script=/home/root/qemu-ifup,downscript=no,ifname=tap0'/>
    <qemu:arg value='-device'/>
    <qemu:arg value='virtio-net-pci,netdev=tap0'/>

  </qemu:commandline>
</domain>
```

3. Now the domain can be defined:

```
# virsh define kvm.xml
Domain kvm defined from kvm.xml

# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm                            shut off
```

4. Next start the domain. This starts the VM and boots the guest Linux.

```
# virsh start kvm
Domain kvm started

# virsh list
 Id    Name                           State
----------------------------------------------------
 3     kvm                            running
```

5. The **virsh console** command can be used to connect to the console of the running Linux domain.

```
# virsh console kvm
Connected to domain kvm
Escape character is ^]

Poky (Yocto Project Reference Distro) 1.5 ls1021aqds /dev/ttyAMA0

ls1021aqds login: root
```

6. To stop the domain use the destroy command:

```
# virsh destroy kvm
Domain kvm destroyed

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm                            shut off
```

7. To remove the domain from libvirt, use the undefine command:

```
# virsh undefine kvm
Domain kvm has been undefined

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
```

11.3 Libvirt Users Guide

11.3.1 Introduction to libvirt

11.3.1.1 Overview

This document is a guide and tutorial to using libvirt on NXP SoCs.

Libvirt is an open source toolkit that enables the management of Linux-based virtualization technologies such as KVM/QEMU virtual machines and Linux containers.

The goal of the libvirt project (see http://libvirt.org) is to provide a stable, standard, hypervisor-agnostic interface for managing virtualization "domains" such as virtual machines and containers. Domains can be remote and libvirt provides full security for managing remote domains over a network. Libvirt is a layer intended to be used as a building block for higher level management tools and applications.

Libvirt provides:

- An interface to remotely manage the lifecycle of virtualization domains-- provisioning, start/stop, monitoring
- Support for a variety of hypervisors-- KVM/QEMU and Linux Containers are supported in the NXP SDK
- **libvirtd** -- a Linux daemon that runs on a target node/system and allows a libvirt management tool to manage virtualization domains on the node
- **virsh** -- a basic command shell for managing libvirt domains
- A standard XML format for defining domains

11.3.1.2 For Further Information

Libvirt is an open source project and a great deal of technical and usage information is available on the libvirt.org website:

- http://libvirt.org/index.html

Additional references:

- Architecture: http://libvirt.org/intro.html
- Deployment: http://libvirt.org/deployment.html
- XML Format: http://libvirt.org/format.html
- virsh command reference: http://linux.die.net/man/1/virsh
- The libvirt wiki has user generated content: http://wiki.libvirt.org/page/Main_Page

Mailing Lists

There are three libvirt mailing lists available which can be subscribed to. Archives of the lists are also available.

https://www.redhat.com/archives/libvir-list

https://www.redhat.com/archives/libvirt-users

https://www.redhat.com/archives/libvirt-announce

11.3.1.3 Libvirt in the NXP QorIQ SDK -- Supported Features

The libvirt packages provides a huge number of capabilities and features. This section describes the features tested in the NXP QorIQ SDK release.

The SDK supports QEMU/KVM and LXC and thus supports URIs for QEMU and LXC:

- qemu:///
- lxc:///

The following virsh commands are supported:

- Domain Management
  - *attach-device* - Attach a device from an XML file. To use --config option, then it will effect after the acitive domain restarted.
  - *attach-disk* - attach disk device
  - *attach-interface* - attach network interface
  - *autostart* - configure a domain to be automatically started at boot
  - *blkdeviotune* - set or query a block device I/O tuning parameters.
  - *console* - connect to the console of a domain
  - *cpu-stats* - show domain cpu statistics (need mount /cgroup/cpuacct).
  - *create* - creates a transient domain from an XML file and starts it
  - *define* - define a new persistent domain from an XML file
  - *desc* - show or modify the description and title of a domain
  - *destroy* - For persistent domains, stops the domain. For transient domains, the domain is destroyed. This command does not gracefully stop the domain.
  - *detach-device* - detach a device from an XML file. To use --config option, then it will effect after the acitive domain restarted.
  - *detach-disk* - detach disk device
  - *detach-interface* - detach network interface
  - *domid* - convert a domain name or UUID to domain id.
  - *domif-setlink* - set link state of a virtual interface.
  - *domiftune* - get/set parameters of a virtual interface.
  - *domname* - convert a domain id or UUID to domain name.
  - *domuuid* - convert a domain name or id to domain UUID.
  - *domxml-from-native* - convert a QEMU command line to libvirt XML
  - *domxml-to-native* - convert a libvirt XML file (for QEMU) to a native QEMU command line. Useful for debugging.
  - *dumpxml* - output the XML for the specified domain
  - *edit* - edit XML configuration for a domain.
  - *maxvcpus* - show connection vcpu maximum (for QEMU)
  - *memtune* - get or set memory parameters.
  - *qemu-monitor-command* - for QEMU/KVM domains allows sending commands to the QEMU monitor

- *reset* - reset a domain
- *restore* - restore a domain from a saved state in a file
- *resume* - resume a suspended domain. After *resume* the domain is in the "running" state.
- *save* - save a domain state to a file
- *schedinfo* - show/set scheduler parameters (need mount cgroup CPU controller).
- *setmaxmem* - change maximum memory limit.
- *setmem* - change memory allocation.
- *start* - start a domain
- *suspend* - suspend a running domain. After *suspend* the domain is the "paused" state.
- *ttyconsole* - show tty console.
- *undefine* - remove a domain (undo the effects of *define*)
- *vcpucount* - show domain vcpu counts.
- *vcpuinfo* - show detailed domain vcpu information (for QEMU).
- *vcpupin* - control or query domain vcpu affinity (for QEMU).
- *emulatorpin* - control or query domain emulator affinity.

- Domain Monitoring
  - *domblkerror* - show errors on block devices (for QEMU).
  - *domblkinfo* - show domain block device size information.
  - *domblklist* - list all domain blocks.
  - *domblkstat* - get device block stats for a domain.
  - *domcontrol* - show domain control interface state.
  - *domif-getlink* - get link state of a virtual interface.
  - *domiflist* - list all domain virtual interfaces.
  - *domifstat* - get network interface stats for a domain.
  - *dominfo* - show domain information.
  - *dommemstat* - get memory statistics for a domain.
  - *domstate* - show domain state.
  - *list* - show the status of all domains
- Host and Hypervisor
  - *capabilities* - show capabilities.
  - *hostname* - print the hypervisor hostname.
  - *nodecpumap* - show node cpu map.
  - *nodecpustats* - print cpu stats of the node.
  - *nodeinfo* - show node information.
  - *nodememstats* - print memory stats of the node.
  - *sysinfo* - print the hypervisor sysinfo.
  - *uri* - print the hypervisor canonical URI.
  - *version* - show version.

- Snapshot
  - *snapshot-create* - Create a snapshot from XML
  - *snapshot-create-as* - Create a snapshot from a set of args
  - *snapshot-current* - Get or set the current snapshot
  - *snapshot-delete* - Delete a domain snapshot
  - *snapshot-dumpxml* - Dump XML for a domain snapshot
  - *snapshot-edit* - edit XML for a snapshot
  - *snapshot-info* - snapshot information
  - *snapshot-list* - List snapshots for a domain
  - *snapshot-parent* - Get the name of the parent of a snapshot
  - *snapshot-revert* - Revert a domain to a snapshot
- Virsh itself
  - *cd* - change the current directory.
  - *connect* - (re)connect to hypervisor.
  - *echo* - echo arguments.
  - *exit* - quit this interactive terminal.
  - *help* - print help.
  - *pwd* - print the current directory.
  - *quit* - quit this interactive terminal.

Other virsh commands may operate correctly, but have not been specifically validated in the QorIQ SDK.

11.3.2 Build, Installation, and Configuration

11.3.2.1 Building Libvirt with Yocto

Libvirt is a Linux user space package that can easily be added to a root filesystem using the Yocto build system.

In the NXP SDK, Libvirt and all pre-requisite user space packages are included when building the "virt" and "full" image type:

```
bitbake fsl-image-virt
bitbake fsl-image-full
```

Libvirt can be easily added to any rootfs image by updating the IMAGE_INSTALL_append variable in the **conf/local.conf** file in the Yocto build environment. For example, append the following line to **local.conf**:

```
IMAGE_INSTALL_append = " libvirt libvirt-libvirtd libvirt-virsh"
```

After libvirt is included in a root filesystem the libvirtd daemon will be automatically started by the system init scripts.

11.3.2.2 Running libvirtd

Running libvirtd

The libvirtd daemon is installed as part of a libvirt packages installation. By default the target system init scripts should start libvirtd.

Running libvirtd on the target system is a pre-requisite to running any management tools such as virsh.

The libvirtd daemon can be manually started like this:

```
$ /etc/init.d/libvirtd start
```

In some circumstances the daemon may need to be restarted such as after mounting cgroups or hugetlbfs. Daemon restart can be done like this:

```
$ /etc/init.d/libvirtd restart
```

The libvirtd daemon can be configured in /etc/libvirt/libvirtd.conf. The file is self-documented and has detailed comments on the configuration options available.

The libvird Daemon and Logging

The libvirt daemon logs data to /var/log/libvirt/

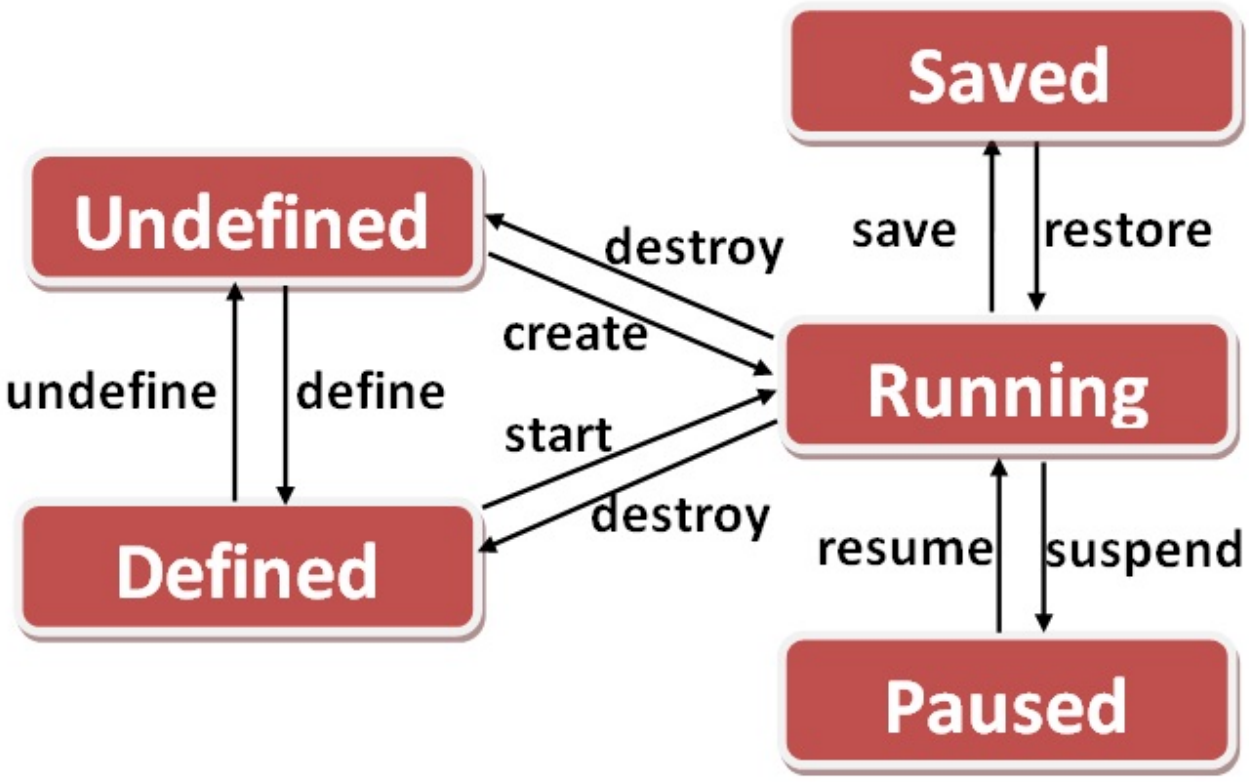
- General libvirtd log messages are in: */var/log/libvirt/libvirtd.log*
- QEMU/KVM domain logs are in: */var/log/libvirt/qemu/[domain-name].log*
- LXC domains logs are in: */var/log/libvirt/lxc/[domain-name].log*

The verbosity of logging can be controlled in */etc/libvirt/libvirtd.conf*.

11.3.2.3 Libvirt Domain Lifecycle

Two types of libvirt domains are supported in the QorIQ SDK-- KVM/QEMU virtual machines and Linux containers.

The following state diagram illustrates the lifecycle of a domain, the states that domains can be in and the **virsh** commands that move the domain between states.



Domain States

1. *Undefined*

There are two types of domains-- persistent and transient domains. All domains begin in the "undefined" state where they are defined in XML definition file, and libvirt is unaware of them.

2. *Defined*

Persistent domains begin with being "defined". This adds the domain to libvirt, but it is not running. This state can also be conceptually thought of as "stopped". The output of **virsh list --all** shows the domain as being "shut off".

3. *Running*

The "running" state is the normal state of an active domain after it has been started. The **start** command is used to move persistent domains into this state. Transient domains go from being undefined to "running" through the **create** command.

4. *Paused*

The domain execution has been suspended. The domain is unaware of being in this state.

5. *Saved*

The domain state has been saved and could be restored again.

See the Libvirt KVM/QEMU Example (Power Architecture) on page 2367 article for an example of a KVM/QEMU domain lifecycle.

See the Basic Example on page 2421 article for an example of a container domain lifecycle.

11.3.2.4 Libvirt URIs

Because libvirt supports managing multiple types of virtualization domains (possibly remote) it uses uniform resource identifiers (URIs) to describes the target "node" to manage and the type of domain being managed.

A URI is specified when tools such as **virsh** makes a connection to a target node running **libvirtd**.

Two types of URIs are supported in the QorIQ SDK-- QEMU/KVM and LXC.

QEMU/KVM URIs are in the form:

- For a local node: *qemu:///system*
- For a remote node: *qemu[+transport]://[hostname]/system*

For Linux containers:

- For a local node: *lxc:///*
- For a remote node: *lxc[+transport]://[hostname]/*

A default URI can be specified in the environment (LIBVIRT_DEFAULT_URI) or in the /etc/libvirt/libvirtd.conf config file.

For further information about URIs:

- http://libvirt.org/uri.html
- http://libvirt.org/remote.html#Remote_URI_reference

11.3.2.5 virsh

The virsh command is a command line tool provided with the libvirt package for managing libvirt domains. It can be used to create, start, pause, shutdown domains. The general command format is:

```
virsh [OPTION]... <command> <domain> [ARG]...
```

11.3.2.6 Libvirt xml

The libvirt XML format is defined at: http://libvirt.org/format.html.

11.3.3 Examples

11.3.3.1 KVM Examples

11.3.3.1.1 Libvirt KVM/QEMU Example (Power Architecture)

The following example shows the lifecycle of a simple KVM/QEMU libvirt domain called *kvm1*.

In this example the default URI is qemu:///system, and because of this default an explicit URI is not used in the virsh commands.

1. We begin with a simple QEMU command line in a text file named kvm1.args:

```
$ cat kvm1.args
/usr/bin/qemu-system-ppc -m 256 -nographic -M ppce500 -kernel /boot/uImage -initrd /home/root/
my.rootfs.ext2.gz -append "root=/dev/ram rw console=ttyS0,115200" -serial pty -enable-kvm -name kvm1
$
```

NOTE

The serial console is a tty, not a telnet server. The -name option is required and specifies the name of the virtual machine.

2. Before defining the domain the QEMU command line must be converted to libvirt XML format:

```
$ virsh domxml-from-native qemu-argv kvm1.args  > kvm1.xml
$
```

The content of the newly created domain XML file is shown below:

```
$ cat kvm1.xml
<domain type='kvm'>
  <name>kvm1</name>
  <uuid>f5b7cf86-41eb-eb78-4284-16501ff9f0e1</uuid>
  <memory unit='KiB'>262144</memory>
  <currentMemory unit='KiB'>262144</currentMemory>
  <vcpu placement='static'>1</vcpu>
  <os>
    <type arch='ppc' machine='ppce500'>hvm</type>
    <kernel>/boot/uImage</kernel>
    <initrd>/home/root/my.rootfs.ext2.gz</initrd>
    <cmdline>root=/dev/ram rw console=ttyS0,115200</cmdline>
  </os>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-ppc</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
```

```
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='virtio'/>
  </devices>
</domain>
```

3. Now the domain can be defined:

```
# virsh define kvm1.xml
Domain kvm1 defined from kvm1.xml

# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm1                           shut off
```

4. Next start the domain. This starts the VM and boots the guest Linux.

```
# virsh start kvm1
Domain kvm1 started

# virsh list
 Id    Name                           State
----------------------------------------------------
 3     kvm1                           running
```

5. The virsh console command can be used to connect to the console of the running Linux domain.

```
# virsh console kvm1
Connected to domain kvm1
Escape character is ^]

Poky 9.0 (Yocto Project 1.4 Reference Distro) 1.4 model : qemu ppce500 ttyS0

model : qemu ppce500 login:
```

Press CTRL +] to exit the console.

6. To stop the domain use the destroy command:

```
# virsh destroy kvm1
Domain kvm1 destroyed

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm1                           shut off
```

7. To remove the domain from libvirt, use the undefine command:

```
# virsh undefine kvm1
Domain kvm1 has been undefined

root@p4080ds:~# virsh list --all
 Id    Name                           State
```

```
-------------------------------------------------
```

11.3.3.1.2 Libvirt KVM/QEMU -- Adding Devices Example (Power Architecture)

This example shows how devices (virtual network, USB, and PCI) can be added to a libvirt domain. The steps are identical to the simple domain example shown in Libvirt KVM/QEMU Example (Power Architecture) on page 2367 , (except some additional preparation is needed for step #1):

1. Create a text file containing the QEMU command line
2. Use **virsh domxml-from-native** to create the libvirt XML
3. Use **virsh define** to define the QEMU domain
4. Use **virsh start** to start the domain

When defining the QEMU command line options in step #1 to add the network, USB, and PCI devices it is necessary to explicitly specify the PCI slot number for each device (all the devices in this example appear on the VM's virtual PCI bus). The remainder of this example explains how to do this.

The normal QEMU command line arguments to assign these devices to a virtual machine would look something like:

- for a virtual network interface

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/root/qemu-ifdown,vhost=on -
device virtio-net-pci,netdev=tap0
```

- for the passthrough of USB device on bus #2, port #1

```
-device usb-ehci,id=ehci -device usb-host,bus=ehci.0,hostbus=2,hostport=1
```

- for the passthrough of PCI device 0000:01:00.0

```
-device vfio-pci,host=0000:01:00.0
```

(For further information on these command line argument, please refere to the KVM/QEMU documentation)

All these devices will appear on the virtual PCI bus in the virtual machine. However, in the context of libvirt it is necessary to explicitly assign the PCI slot for each device and therefore it is necessary to determine what slot are occupied and free.

First, start the virtual machine with libvirt **without the new devices** so that the default PCI slot assignment can be viewed (the example is for a domain called "kvm1" on a p4080ds):

```
$ echo '/usr/bin/qemu-system-ppc -m 512 -nographic -M ppce500 -kernel /home/root/uImage -initrd /
home/root/fsl-image-minimal-p4080ds.rootfs.ext2.gz  -append "root=/dev/ram rw
console=ttyS0,115200" -enable-kvm -smp 8 -mem-path /var/lib/hugetlbfs/pagesize-16MB -name kvm1 -
serial pty' > kvm1.args

$ virsh domxml-from-native qemu-argv kvm1.args  > kvm1.xml
$ virsh define kvm1.xml
$ virsh start kvm1
```

Next, view the devices on the PCI bus:

```
$ virsh qemu-monitor-command --hmp kvm1 'info pci'
  Bus  0, device   0, function 0:
    PCI bridge: PCI device 1957:0030
      BUS 0.
      secondary bus 0.
```

```
      subordinate bus 0.
      IO range [0x0000, 0x0fff]
      memory range [0x00000000, 0x000fffff]
      prefetchable memory range [0x00000000, 0x000fffff]
      BAR0: 32 bit memory at 0xc0000000 [0xc00fffff].
      id ""
  Bus  0, device   1, function 2:
    USB controller: PCI device 8086:7020
      IRQ 0.
      BAR4: I/O at 0xffffffffffffffff [0x001e].
      id "usb"
  Bus  0, device   3, function 0:
    Class 0255: PCI device 1af4:1002
      IRQ 0.
      BAR0: I/O at 0x1000 [0x101f].
      id "balloon0"
```

In this example it can be seen that the highest occupied slot is 0x3, which means that slots 0x4 and higher are available.

The explicit definition of the PCI slot numbers on the QEMU command line looks like this:

- for a virtual network interface (PCI slot/addr 0x4)

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/root/qemu-ifdown,vhost=on -
device virtio-net-pci,netdev=tap0,bus=pci.0,addr=0x4
```

- for the passthrough of USB device on bus #2, port #1 (PCI slot/addr 0x5)

```
-device usb-ehci,id=ehci,bus=pci.0,addr=0x5 -device usb-host,bus=ehci.0,hostbus=2,hostport=1
```

- for the passthrough of PCI device 0000:01:00.0 (PCI slot/addr 0x6)

```
-device vfio-pci,host=0000:01:00.0,bus=pci.0,addr=0x06
```

The virtual machine's PCI bus is "pci.0".

The complete command sequence to define the domain and start it looks like:

```
$ echo '/usr/bin/qemu-system-ppc -m 512 -nographic -M ppce500 -kernel /home/root/uImage -initrd /
home/root/fsl-image-minimal-p4080ds.rootfs.ext2.gz  -append "root=/dev/ram rw
console=ttyS0,115200" -enable-kvm -smp 8 -mem-path /var/lib/hugetlbfs/pagesize-16MB -name kvm1 -
serial pty -netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/root/qemu-
ifdown,vhost=on -device virtio-net-pci,netdev=tap0,bus=pci.0,addr=0x4 -device usb-
ehci,id=ehci,bus=pci.0,addr=0x5 -device usb-host,bus=ehci.0,hostbus=2,hostport=1 -device vfio-
pci,host=0000:01:00.0,bus=pci.0,addr=0x06' > kvm1.args

$ virsh domxml-from-native qemu-argv kvm1.args  > kvm1.xml
$ virsh define kvm1.xml
$ virsh start kvm1
```

Libvirt also supports adding devices into a domain by modifying the domain XML file directly.

Shutdown the guest domain and edit the guest XML definition with the virsh edit command

```
$ virsh edit kvm1
```

- for a virtual network interface (PCI slot/addr 0x4)

```
<interface type="ethernet">
  <script path="/home/root/qemu-ifup"/>
```

```
  <model type="virtio"/>
  <driver name="vhost"/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
</interface>
```

- for the passthrough of PCI device 0000:01:00.0 (PCI slot/addr 0x6)

```
<hostdev mode="subsystem" type="pci" managed="yes">
  <driver name="vfio"/>
  <source>
    <address domain="0x0000" bus="0x01" slot="0x06" function="0x0"/>
  </source>
</hostdev>
```

Start the domain again.

It is also possible to add and remove devices by defining the devicde in a standalone XML file. The **virsh attach-device/ detach-device** commands commands can be used when the domain is not active.

For example,

```
$ cat vhost_on.xml
<interface type='ethernet'>
  <script path='/home/root/qemu-ifup'/>
  <model type='virtio'/>
  <driver name='vhost'/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
</interface>
```

Attach device to the guest domain using **virsh attach-device.** The resulting XML file can be dumped to find the added device:

```
$ virsh attach-device --config kvm1 vhost_on.xml
$ virsh dumpxml kvm1
```

11.3.3.1.3 Libvirt KVM/QEMU Example (ARM Architecture)

The following example shows the lifecycle of a simple KVM/QEMU libvirt domain called **kvm**. In this example the default URI is qemu:///system, and because of this default an explicit URI is not used in the virsh commands

Libvirt has the possibility to convert qemu command line arguments in xml. However, the current version of libvirt does not have full support for all qemu arguments. It can be used to generate a basic xml file, but there is currently a problem that it generates a default USB node which should be removed.

1. We begin with a simple QEMU command line in a text file named **kvm.args**:

- 32 bit ARMv7:

```
echo "/usr/bin/qemu-system-arm -name kvm -smp 2 -enable-kvm -m 512  -nographic -cpu host -machine
type=virt -kernel /boot/zImage  -serial pty -initrd /boot/fsl-image-core-ls1021atwr.ext2.gz -
append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' " > kvm.args
```

- 64 bit ARMv8:

```
echo "/usr/bin/qemu-system-aarch64 -name kvm -smp 2 -enable-kvm -m 512 -nographic -cpu host -
machine type=virt -kernel /boot/Image  -serial pty -initrd /boot/guest.rootfs.ext2.gz -append
'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' " > kvm.args
```

NOTE

The serial console is a tty, not a telnet server. The -name option is required and specifies the name of the virtual machine.

2. Before defining the domain the QEMU command line must be converted to libvirt XML format:

```
virsh domxml-from-native qemu-argv kvm.args > kvm.xml
```

NOTE

For ARMv7 paltforms the generated xml file has to be manually changed to remove the USB node.

The content of the newly created domain XML file is shown below:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>12a3391e-9cd4-43dd-b8b7-27b1fb193378</uuid>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type machine='virt'>hvm</type>
    <kernel>/boot/zImage</kernel>
    <initrd>/boot/fsl-image-core-ls1021atwr.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-arm</emulator>
    <controller type='usb' index='0'/>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
  </devices>
 </domain>
```

A more complex example including devices can be found bellow.

The example contains two devices:

- a virtio network interface
- a virtio block device (disk)

NOTE

This example will use MMIO as transport for virtio. Currently libvirt has no support for PCI transport, but it can be used using passthrough QEMU command line arguments (see the next example)

Device example (using virtio with MMIO as transport):

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>12a3391e-9cd4-43dd-b8b7-27b1fb193378</uuid>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type machine='virt'>hvm</type>
    <kernel>/boot/zImage</kernel>
    <initrd>/boot/fsl-image-core-ls1021atwr.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-arm</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>

    <interface type='ethernet'>
      <mac address='52:54:00:b0:39:28'/>
      <script path='/home/root/qemu-ifup'/>
    </interface>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw' cache='none'/>
      <source file='/home/root/my_guest_disk'/>
      <target dev='vda' bus='virtio'/>
    </disk>

  </devices>
  <qemu:commandline>
    <qemu:arg value='-mem-path'/>
    <qemu:arg value='/var/lib/hugetlbfs/pagesize-2MB'/>
  </qemu:commandline>
</domain>
```

Device example (using virtio with PCI as transport):

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>faa89ddc-e156-46c0-9d0f-029c322f9bf7</uuid>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <vcpu placement='static'>4</vcpu>
  <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <kernel>/boot/Image</kernel>
```

```
    <initrd>/images/fsl-image-core-ls1043ardb.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-aarch64</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='none'/>
  </devices>
  <qemu:commandline>

    <qemu:arg value='-netdev'/>
    <qemu:arg value='tap,id=tap0,script=/home/root/qemu-ifup,downscript=no,ifname=tap0'/>
    <qemu:arg value='-device'/>
    <qemu:arg value='virtio-net-pci,netdev=tap0'/>

  </qemu:commandline>
</domain>
```

3. Now the domain can be defined:

```
# virsh define kvm.xml
Domain kvm defined from kvm.xml

# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm                            shut off
```

4. Next start the domain. This starts the VM and boots the guest Linux.

```
# virsh start kvm
Domain kvm started

# virsh list
 Id    Name                           State
----------------------------------------------------
 3     kvm                            running
```

5. The **virsh console** command can be used to connect to the console of the running Linux domain.

```
# virsh console kvm
Connected to domain kvm
Escape character is ^]

Poky (Yocto Project Reference Distro) 1.5 ls1021aqds /dev/ttyAMA0
```

```
ls1021aqds login: root
```

6. To stop the domain use the destroy command:

```
# virsh destroy kvm
Domain kvm destroyed

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm                            shut off
```

7. To remove the domain from libvirt, use the undefine command:

```
# virsh undefine kvm
Domain kvm has been undefined

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
```

11.3.3.2 Libvirt_lxc Examples

11.3.3.2.1 Basic Example

The following example shows the lifecycle of a simple LXC libvirt domain called *container1*. The virsh tool is used for managing the lxc domain lifecycle.

1. Confirm the host Linux configuration. Begin by confirming that the host kernel is configured correctly and that rootfs setup such as mounting cgroups has been done. This can be done with the lxc-checkconfig command.

```
# lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

2. **Create a libvirt XML file defining the container**. The example below shows a very simple container defined in container1.xml that runs the command /bin/sh and has a console:

```
# cat container1.xml
<domain type='lxc'>
  <name>container1</name>
  <memory>500000</memory>
  <os>
    <type>exe</type>
    <init>/bin/sh</init>
  </os>
  <devices>
    <console type='pty'/>
  </devices>
</domain>
```

3. **Define the container.** The *virsh define* command processes the XML and makes creates the new libvirt domain.

```
# virsh -c lxc:/// define container1.xml
Domain container1 defined from container1.xml

# virsh -c lxc:/// list --all
 Id    Name                           State
----------------------------------------------------
 -     container1                     shut off
```

4. Start the container.

```
# virsh -c lxc:/// start container1
Domain container1 started

# virsh -c lxc:/// list
 Id    Name                           State
----------------------------------------------------
 3196  container1                     running
```

5. Connect to the console.

```
# virsh -c lxc:/// console container1
Connected to domain container1
Escape character is ^]
sh-4.2#
sh-4.2# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 18:25 pts/2    00:00:00 /bin/sh
root         3     1  0 18:36 pts/2    00:00:00 ps -ef

sh-4.2# ifconfig br0
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.171.73.123  netmask 255.255.254.0  broadcast 10.171.73.255
        inet6 fe80::a00:27ff:fe01:fe07  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:01:fe:07  txqueuelen 0  (Ethernet)
        RX packets 865838  bytes 104029354 (99.2 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 104446  bytes 43998714 (41.9 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
sh-4.2#
```

Press CTRL +] to exit the console.

The following aspects must be noted:

- the processes inside the container are running in a separate namespace, hence the different process hierarchy
- since no network configuration for the domain is explicitly specified, all networking interfaces are shared with the host (all the other interfaces are present too - br0 is mentioned as an example)
- since no filesystem configuration is specified for the domain, the filesystem is shared with the host-- all host mounts are present in the container as well.

6. To stop the container use the destroy command:

```
# virsh -c lxc:/// destroy container1
Domain container1 destroyed

# virsh -c lxc:/// list  --all
 Id    Name                           State
----------------------------------------------------
 -     container1                     shut off
```

7. To remove the domain from libvirt, use the undefine command.

```
# virsh -c lxc:/// undefine container1
Domain container1 has been undefined
```

11.3.3.2.2 Custom Container Filesystem

The libvirt documentation (http://libvirt.org/formatdomain.html#elementsFilesystems) details the flavors and usage of the **filesystem** tag in order to assign particular types of filesystem mounts to the domain.

Mounts

The <mount> tag specifies private root filesystem, available on host in a specific directory. It will be the rootfs of the container. The filesystem can be handcrafted, installed from media, debootstrapped, etc. Below is an example snippet of the XML that assigns a filesystem to a container:

```
<domain type='lxc'>
  [ ... ]
  <devices>
    [ ... ]
    <filesystem type='mount'>
      <source dir='/var/lib/lxc/foo/rootfs'/>
      <target dir='/'/>
    </filesystem>
  </devices>
</domain>
```

The result is that the domain is started with the root mounted at */var/lib/lxc/foo/rootfs*.

For ease of use, the example that follows will use the standard LXC command lxc-create to build a container Busybox rootfs. However, the default rootfs created by lxc-create will not work with libvirt tools as-is, and some additional terminal setup must be done. This will be detailed in the next example: Container Terminal Setup on page 2424.

Please note that, if you're planning to use this LXC built rootfs with libvirt containers, you will also have to bind-mount the host library directories. These vary depending on whether the libraries are 32bit, 64 bit or both. Basically, you will have to bind-mount all the available library dirs among **/lib, /usr/lib, /lib64, /urs/lib64**, using entries like the one below:

```
<domain type='lxc'>
  [ ... ]
  <devices>
    [ ... ]
    <filesystem type='mount'>
      <source dir='/lib'/>
      <target dir='/lib'/>
      <readonly />
    </filesystem>
  </devices>
</domain>
```

11.3.3.2.3 Container Terminal Setup

Each LXC domain needs at least one console device defined in the XML. By default, libvirt will link this console to the process and make it available with **virsh console** command.

Libvirt also offers support for multiple consoles in a container. This section provides an example describing how libvirt can be used to start four processes inside the container and assign each one of these a private terminal. This will be done by using:

- a Busybox container filesystem, built with lxc-create
- a modified inittab
- the container XML configuration

Libvirt will mount a tmpfs on /dev and a devpts on /dev/pts and it will create all the device nodes itself inside the domain. The domain definition will be altered so that it will start the busybox-init as the init process.

The init process reads the /etc/inittab file inside the rootfs and will determine additional processes to start, and the terminals to link them to. Here is an example inittab that specifies the four processes to start and the separate tty device assigned to each:

```
::sysinit:/etc/init.d/rcS
tty1::askfirst:/bin/sh
tty2::respawn:/bin/getty -L tty2 115200 vt100
tty3::respawn:/bin/getty -L tty3 115200 vt100
tty4::respawn:/bin/getty -L tty4 115200 vt100
```

The domain XML configuration that shows the /sbin/init as the initial program to run and describes the four tty devices looks like this:

```
<domain type='lxc'>
  [ ... ]
  <os>
    <type>exe</type>
    <init>/sbin/init</init>
  </os>
  [ ... ]
  <devices>
    [ ... ]
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
```

```
    <console type='pty'>
      <target type='serial' port='1'/>
    </console>
    <console type='pty'>
      <target type='serial' port='2'/>
    </console>
    <console type='pty'>
      <target type='serial' port='3'/>
    </console>
  </devices>
</domain>
```

Using the inittab above and XML configs, **virsh start** will start the following process hierarchy:

```
# ps axf
[ ... ]
18450 ?        Ss  0:00 /usr/libexec/libvirt_lxc --name foo --console 24 --console 25 --console 26
--console 27 --security=selinux --handshake 30 --background
18451 pts/0    Ss+ 0:00  \_ init
18454 ?        Ss  0:00      \_ /bin/syslogd
18459 ?        Ss  0:00      \_ /bin/sh
18460 pts/1    Ss+ 0:00      \_ /bin/getty -L tty2 115200 vt100
18461 pts/2    Ss+ 0:00      \_ /bin/getty -L tty3 115200 vt100
18462 pts/3    Ss+ 0:00      \_ /bin/getty -L tty4 115200 vt100
```

By running **virsh -c lxc:/// dumpxml foo**, we can see what alias libvirt has assigned to each console device:

```
<domain type='lxc' id='18450'>
  [ ... ]
  <devices>
    [ ... ]
    <console type='pty' tty='/dev/pts/3'>
      <source path='/dev/pts/3'/>
      <target type='serial' port='0'/>
      <alias name='console0'/>
    </console>
    <console type='pty'>
      <source path='/dev/pts/4'/>
      <target type='serial' port='1'/>
      <alias name='console1'/>
    </console>
    <console type='pty'>
      <source path='/dev/pts/5'/>
      <target type='serial' port='2'/>
      <alias name='console2'/>
    </console>
    <console type='pty'>
      <source path='/dev/pts/6'/>
      <target type='serial' port='3'/>
      <alias name='console3'/>
    </console>
  </devices>
</domain>
```

To connect to a particular console, one must run

```
virsh -c lxc:/// console --devname <console_alias> <domain_name>
```

```
[root@everest][~]# virsh -c lxc:/// console --devname console2 foo
Connected to domain foo
Escape character is ^]

everest.ea.freescale.net login: root
#
#
#
```

11.3.3.2.4 Networking Examples

Libvirt offers extensive support when it comes to networking. The purpose of this section is to provide some insight of how standard LXC networking scenarios can be achieved with Libvirt - to provide the same functionality. In its further versions, this document could include details regarding other, more advanced networking options as well.

11.3.3.2.4.1 Shared Networking

Shared Networking

By default libvirt containers share network interfaces and the network namespace with the host. To share the hosts network interfaces, simply do not define any network interfaces in the domain XML.

When at least one interface is defined, the container will be started in a new network namespace, with the defined interface and a loopback.

No Networking

In order to remove all access to the host's network interfaces, start the container in a new network namespace, without specifying any interface. To do this use the <privnet> XML tag as seen in the example below. This will a loopback interface in the new namespace, but the host network interfaces are not visible in the container.

```
<domain type='lxc'>
  [ ... ]
  <features>
    <privnet/>
  </features>
</domain>
```

11.3.3.2.4.2 Ethernet Bridging

This is the recommended configuration for general guest activity on hosts with static wired networking configurations. It relies on 802.1d Ethernet Bridging, and it provides a bridge from the VM directly into the LAN. This assumes there is a bridge device on the host with one or more of the host's physical NICs enslaved to it. The guest VM will have an associated tun device created with a name of vnetN, which can also be overridden with the **target** element in the XML config file. This tun device will also be enslaved to the bridge. The IP range / network configuration is whatever is used on the LAN. This provides the guest VM full incoming and outgoing net access just like a physical machine. The bridge normally is a Linux bridge - but this can be configured to be an open vSwitch as well, if it is supported on the host. This is done by adding some further parameters in the config file.

Host Configuration

The physical interface **fm1-gb1** is added to a bridge device - **br0**:

```
ifconfig fm1-gb1 0.0.0.0 up
brctl addbr br0
ifconfig br0 192.168.1.141 up
brctl addif br0
```

XML Description of the Interface

Only the relevant part is described here - defining the interface of the guest in XML:

```
<domain type='lxc'>
  [ ... ]
  <devices>
    [ ... ]
    <interface type="bridge">
      <source bridge="br0" />
    </interface>
  </devices>
</domain>
```

Guest Configuration and Testing

After booting the container the network interface can be managed normally:

```
~ # ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:83:78:F4
          inet6 addr: fe80::5054:ff:fe83:78f4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3104 (3.0 KiB)  TX bytes:636 (636.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

~ # ifconfig eth0 192.168.1.143
~ # ping -c 3 192.168.1.1 # gateway
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: seq=0 ttl=64 time=3.557 ms
64 bytes from 192.168.1.1: seq=1 ttl=64 time=0.220 ms
64 bytes from 192.168.1.1: seq=2 ttl=64 time=0.227 ms

--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.220/1.334/3.557 ms
~ # ping -c 3 192.168.1.141 # host
PING 192.168.1.141 (192.168.1.141): 56 data bytes
64 bytes from 192.168.1.141: seq=0 ttl=64 time=3.493 ms
64 bytes from 192.168.1.141: seq=1 ttl=64 time=0.050 ms
64 bytes from 192.168.1.141: seq=2 ttl=64 time=0.036 ms

--- 192.168.1.141 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.036/1.193/3.493 ms
~ #
```

11.3.3.2.4.3 MACVLAN

Macvlan is a relatively new Linux kernel technology used to ease the task of networking in virtual machines. See the following article for some useful overview information: http://seravo.fi/2012/virtualized-bridged-networking-with-macvtap.

Macvlan is device driver consisting of 2 components:

- the **macvlan** driver - which makes it possible to create **virtual network interfaces** that "cling on" a physical network interface. Each virtual interface has its own MAC address distinct from the physical interface's MAC address. Frames sent to or from the virtual interfaces are mapped to the physical interface, which is called **the lower interface**.
- the **tap** interface - a software-only interface, using to pass Ethernet frames. Instead of passing frames to and from a physical Ethernet card, the frames are read and written by a userspace program. The kernel makes the Tap interface available via the */dev/tapN* device file, where N is the index of the network interface.

Libvirt uses the macvtap device technology to attach virtual network interfaces to physical ones. This has no impact on the functionality of the physical interfaces on the host. The macvtap device has a different approach than the bridge. While the latter provides connectivity from the host to the virtual devices, the former isolates them. The bridge unifies the physical interface with the virtual ones, thus providing a common addressing space and connectivity between each 2 endpoints. The macvtap device will put the virtual interfaces in separate MAC-based VLAN's, isolated from the host.

The macvtap device can function in three different modes: **vepa**, **bridge** and **private** - libvirt supports all three of them. In order to configure networking using a macvtap device, the type attribute of the interface is set to "direct" which can be seen in the following XML example:

```
<domain type='lxc'>
  [ ... ]
    <devices>
    [ ... ]
    <interface type="direct">
      <source dev="p7p1" mode="vepa" />
    </interface>
  </devices>
</domain>
```

In the above example, **p7p1** is a physical network interface on the host. The **mode** tag specifies the mode of the macvtap device - and it can be one of the following:

- **vepa** (Virtual Ethernet Port Aggregator) - in this mode, which is the default, data between endpoints on the same lower device are sent via the lower device (physical Ethernet card), to the physical switch the lower device is connected to. This mode requires that the switch supports *Reflective Relay* mode, also known as *Hairpin* mode. Reflective Relay means that the switch can send back a frame on the same port it received it on. The reason this mode exists is to leverage the switching computation to an external switch (and thus freeing the host).
- **bridge** - in this mode, the endpoints can communicate directly without sending the data out via the lower device. There is no isolation between endpoints on the same lower device, but there is isolation between them and the lower device itself.
- **private** - the nodes on the same Macvtap device can never talk to each other. This is used when you want to isolate the virtual machines connected to the endpoints from each other, but not from the outside network.

11.3.3.2.4.4 Direct Assignment

This options enables a container to have private access to a host interface directly. It is similar to the **lxc-phys** networking configuration option. Technically, this option will move a host network interface from the host network namespace to the container's. Once the container is stopped (destroyed), the interface will be assigned back to the host network namespace. While the container is running, the interface will not be available from the host.

XML Description of the Interface

Assuming the host has interface **fm2-gb0**, this is the XML snippet that assigns it to the container:

```
<domain type='lxc'>
  [ ... ]
  <devices>
    [ ... ]
```

```
    <hostdev mode='capabilities' type='net'>
      <source>
        <interface>fm2-gb0</interface>
      </source>
    </hostdev>
  </devices>
</domain>
```

Guest Configuration and Testing

Once the container is started, the inteface must be configured with IP, netmask, etc. Then it can be used just like it would have been on the host.

```
~ # ifconfig fm2-gb0
fm2-gb0   Link encap:Ethernet  HWaddr 00:04:9f:00:02:05
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:18 errors:0 dropped:0 overruns:0 frame:0
          TX packets:56 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1094 (1.0 KiB)  TX bytes:4068 (3.9 KiB)
          Memory:fe5e0000-fe5e0fff

~ # ifconfig fm2-gb0 20.0.0.3 netmask 255.0.0.0
~ # ping -c 3 20.0.0.1
PING 20.0.0.1 (20.0.0.1) 56(84) bytes of data.
64 bytes from 20.0.0.1: icmp_req=1 ttl=64 time=0.377 ms
64 bytes from 20.0.0.1: icmp_req=2 ttl=64 time=0.186 ms
64 bytes from 20.0.0.1: icmp_req=3 ttl=64 time=0.198 ms

--- 20.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.186/0.253/0.377/0.089 ms
~ #
```

When returning to the host network namespace, the interface loses its configuration and it is down.

11.3.3.2.4.5 VLAN

Libvirt does not directly provide VLAN configuration support-- there is no equivalent to the capability in user space LXC where a container can be configured to have a specific VLAN assigned only to itself (see LXC: How to configure networking using a VLAN (lxc-vlan.conf) on page 2451.

Libvirt containers does not offer equivalent support, but VLANs can be used with libvirt containers using traditional VLAN tools such as vconfig, and the **bridge** or **macvtap** network sharing modes.

This leads to the following different configuration scenarios for using VLAN based network interfaces:

1. Configuring a VLAN on the host:

   ```
   vconfig add eth0 2
   ifconfig eth0.2 192.168.20.2
   ```

   then exposing the **eth0.2** sub-interface in the containers, using **bridging** or direct attachment through a **macvtap** device. These would provide the same functionality as if **eth0.2** were a normal physical interface on the host, but it will only provide connectivity inside the VLAN.

2. Configuring VLAN in container shared with macvtap - in this scenario we assume that the domain has been configured to attach directly to the eth0 interface, and we create the VLAN inside the container. Note that this scenario works only if eth0 does not have a sub-interface in the same VLAN defined on the host. As stated in the Macvtap section, ping will work only between virtual endpoints and the outside network.

3. Configuring VLAN in container shared with bridge - in this scenario we assume that the domain has been configured to attach to a host bridge. This bridge previously had a physical interface attached to it. When starting the domain, virtual endpoints will be created and attached to this bridge inside the domain. We create the VLAN sub-interfaces inside the created containers. Note that this scenario works only if there has not been configured a sub-interface on the physical interface with the same VLAN id.

11.4 Linux Containers (LXC) for NXP QorIQ User's Guide

11.4.1 Introduction to Linux Containers

11.4.1.1 NXP LXC Release Notes

This document describes current limitations in the release of LXC for NXP SoCs.

```
Copyright (C) 2015 NXP Semiconductors, Inc.

NXP LXC Release Notes
04/27/2016

Overview
--------
This document describes new features, current limitations, and
working demos in Linux Containers (LXC) for NXP QorIQ SDK 2.0.


Fixes
-----
  o Seccomp support on ARMv8 platforms.

  o Unprivileged containers support on ARMv8 platforms.


SDK Demo List
-----
  o Basic container usage flow and management commands
  o Container networking setups
       o Shared networking
       o Private NICs
       o Ethernet bridge
       o MACVLAN
       o VLAN
  o Adjusting container capabilities
  o Tuning container resource usage
  o Running application containers
  o Isolating USDPAA applications in LXC containers. This has been
    tested using the USDPAA reflector app in a Multiple Instance Scenario
    on a DPAA board. After parititioning the board resources in order to
    support multiple reflector instances, these have been further isolated
    in container environments.
  o Running an unprivileged container linked to a host bridge.
  o Running containers with Seccomp protection.
```

11.4.1.2 Overview

This document is a guide and tutorial to using Linux Containers on NXP e500-based, ARMv7 and ARMv8-based SoCs.

Linux Containers is a lightweight virtualization technology that allows the creation of environments in Linux called "**containers**" in which Linux applications can be run in isolation from the rest of the system and with fine grained control over resources allocated to the container (e.g. CPU, memory, network).

There are 2 implementations of containers in the QorIQ SDK:

- LXC. LXC is a user space package that provides a set of commands to create and manage containers and uses existing Linux kernel features to accomplish the desired isolation and control.
- Libvirt. The libvirt package is a virtualization toolkit that provides a set of management tools for managing virtual machines and Linux containers. See the ***Libvirt Users Guide*** chapter for general information regarding libvirt. The libvirt driver for containers is called "lxc", but the libvirt "lxc" driver is distinct from the user space LXC package.

Applications in a container run in a "sandbox" and can be restricted in what they can do and what visibility they have. In a container:

- An application "sees" only other processes that are in the container.
- An application has access only to network resources granted to the container.
- If configured as such, an application "sees" only a container-specific root filesystem. In addition to limiting access to data in the system's host rootfs, by limiting the */dev* entries that exist in the containers rootfs this limits the devices that the container can access.
- The file POSIX capabilities available to programs are controlled and configured by the system administrator.
- The container's processes run in what is known as a "control group" which the system administrator can use to monitor and control the container's resources.

Why are containers useful? Below are a few examples of container use cases:

- **Application partitioning** -- control CPU utilization between high priority and low priority applications, control what resources applications can access.
- **Virtual private server** -- boot multiple instances of user space, each which effectively looks like a private instance of a server. This approach is commonly used in website infrastructure.
- **Software upgrade** -- run Linux user space in a container, when it becomes necessary to upgrade applications in the system, create and test upgraded software in a new container. The old container can be stopped and the new container can be started as desired.
- **Terminal servers** -- user accesses the system with a thin client, with containers on the server providing applications. Each user gets a private, sandboxed workspace.

There are two general usage models for containers:

- **application containers**: Running a single application in a container. In this scenario, a single executable program is started in the container.
- **system containers**: Booting an instance of user space in a container. Booting multiple system containers allows multiple isolated instances of user space to run at the same time.

Containers are conceptually different than virtual machine technologies such as QEMU/KVM. Virtual machines emulate a hardware platform and are capable of booting an operating system kernel. A container is a mechanism to isolate Linux applications. In a system using containers there is only one Linux kernel running -- the host Linux kernel.

11.4.1.3 Comparing LXC and Libvirt

LXC and the lxc driver in libvirt provide similar capabilities and use the same kernel mechanisms to create containers. This section highlights some of the differences between the two tools.

LXC

- Container management is done with local LXC package commands. No remote support.

- Container creation done with **lxc-create**. LXC config file and template govern the creation of the template and the container's rootfs.

libvirt

- libvirt abstracts the container and thus a variety of tools can be used to manage containers.
- Remote management is supported.
- Container configuration defined in libvirt XML file.
- No tools to facilitate container creation.
- Same tools can be used to manage containers and KVM/QEMU virtual machines.

11.4.1.4 For Further Information

Linux containers is an approach to virtualization similar to OS virtualization solutions such as Linux VServer and OpenVZ that are widely used for virtual private servers. Documentation for these projects has helpful and relevant information:

- http://linux-vserver.org/Overview
- http://wiki.openvz.org/Main_Page

The LXC package is an open source project and much information is available online.

See the chapter ***Libvirt Users Guide*** for general information about libvirt.

Web

- libvirt LXC driver: http://libvirt.org/drvlxc.html
- Getting started with LXC using libvirt : https://www.berrange.com/posts/2011/09/27/getting-started-with-lxc-using-libvirt/
- LXC: Official web page for the LXC project: https://linuxcontainers.org/
- LXC: Overview article on LXC on IBM developerWorks (2009): http://www.ibm.com/developerworks/linux/library/l-lxc-containers/
- Article on POSIX file capabilities: http://www.friedhoff.org/posixfilecaps.html
- SUSE LXC tutorial: https://www.suse.com/documentation/sles11/singlehtml/lxc_quickstart/lxc_quickstart.html
- LXC Linux Containers, presentation: http://www.slideshare.net/samof76/lxc-17456998
- Stephane Graber's LXC 1.0 blog posts: https://www.stgraber.org/2013/12/20/lxc-1-0-blog-post-series/
- Linux Plumbers 2013 videos: https://www.youtube.com/channel/UCIxsmRWj3-795FMlrsikd3A/videos

Containers and Security

If using containers to sandbox untrusted applications, a thorough understanding is needed of the capabilities granted to a container and the security vulnerabilities they may imply. The following references are helpful for understanding container security:

- Ubuntu's security issues and mitigations with LXC, https://wiki.ubuntu.com/LxcSecurity
- Emeric Nasi, Exploiting capabilities, http://www.sevagas.com/IMG/pdf/exploiting_capabilities_the_dark_side.pdf
- Secure containers with SELinux and Smack, http://www.ibm.com/developerworks/linux/library/l-lxc-security/index.html
- Seccomp and sandboxing, http://lwn.net/Articles/332974/

Mailing Lists

For LXC, there are two mailing lists available which can be subscribed to. Archives of the lists are also available.

```
https://lists.linuxcontainers.org/listinfo/lxc-devel
```

```
https://lists.linuxcontainers.org/listinfo/lxc-users
```

11.4.2 LXC: Build, Installation, Configuration

11.4.2.1 Summary

To prepare a root filesystem with the needed components for using LXC-based or libvirt containers the following steps are required:

1. Build and include the LXC and/or libvirt packages in the rootfs. For LXC see: LXC: Building with Yocto on page 2433). For libvirt see the chapter ***Libvirt Users Guide*** .
2. Update the Linux kernel configuration and build to included features needed to support containers (see Building the Linux Kernel on page 2433).
3. Update the rootfs so the system is ready to support containers (see Host Root Filesystem Configuration for Linux Containers on page 2436).

11.4.2.2 LXC: Building with Yocto

LXC is a Linux user space package that can easily be added to a rootfs using the Yocto build system.

In the NXP SDK, LXC and all pre-requisite user space packages are included when building the "full" and "virt" image types:

```
bitbake fsl-image-full
bitbake fsl-image-virt
```

LXC can be easily added to any rootfs image by updating the IMAGE_INSTALL_append variable in the **conf/local.conf** file in the Yocto build environment. For example, append the following line to **local.conf**:

```
IMAGE_INSTALL_append = " lxc"
```

If you are building for ARM64 platforms, you need to perform an additional step. You need to update the rootfs target in the board image - by default it is fsl-image-core. If you plan to update this rootfs to, say, fsl-image-virt, you need to add the following line in to build_<machine_release>/conf/local.conf:

```
ROOTFS_IMAGE = "fsl-image-virt”
```

After enabling the required Linux options mentioned in the following chapter, generate the kernel itb:

```
bitbake fsl-image-kernelitb
```

11.4.2.3 Building the Linux Kernel

In order to use LXC the Linux kernel must be configured with options to enable cgroups, namespaces, POSIX file capabilities, and options to support networking in containers.

These options can be enabled automatically by building the Linux kernel with an additional config fragment. In order to do this, add the following line in the **conf/local.conf** file in the Yocto build environment:

```
DELTA_KERNEL_DEFCONFIG_append = " <sdk-devel>/sources/meta-freescale/recipes-kernel/linux/files/
```

```
containers.config"
```

Alternatively, you can enable the options manually. To configure and build the Linux kernel:

```
bitbake virtual/kernel -c cleansstate
bitbake virtual/kernel -c menuconfig
bitbake virtual/kernel
```

Make sure the following configuration options are enabled:

Kernel Configuration Options	Description
General setup ---> [*] Control Group support --> [*] Example debug cgroup subsystem [*] Freezer cgroup subsystem [*] Device controller for cgroups [*] Cpuset support [*] Simple CPU accounting cgroup subsystem [*] Resource counters [*] Memory Resource Controller for Control Groups [*] Memory Resource Controller Swap Extension [*] Memory Resource Controller Swap Extension enabled by default (NEW) [*] Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL) [*] HugeTLB Resource Controller for Control Groups [*] Enable perf_event per-cpu per-container group (cgroup) monitoring [*] Group CPU scheduler [*] Block IO controller	Control Group settings

Kernel Configuration Options	Description
General setup ---> [*] Namespaces support --->[*] UTS namespace [*] IPC namespace [*] User namespace [*] PID Namespaces [*] Network namespace	Namespaces settings

Kernel Configuration Options	Description
Device Drivers --->[*] Network device support --->[*] MAC-VLAN support (EXPERIMENTAL)	Network Device Drivers settings

Kernel Configuration Options	Description
<*> MAC-VLAN based tap driver (EXPERIMENTAL) <*> Virtual ethernet pair device	

Kernel Configuration Options	Description
Device Drivers ---> [*] Character devices ---> [*] Unix98 PTY support [*] Support multiple instances of devpts	Character Device Drivers settings

Kernel Configuration Options	Description
[*] Networking support ---> Networking options ---> <*> 802.1d Ethernet Bridging [*] IGMP/MLD snooping (NEW) <*> 802.1Q VLAN Support [*] GVRP (GARP VLAN Registration Protocol) support	Networking support settings

Kernel Configuration Options	Description
File systems ---> <*> Second extended fs support [*] Ext2 extended attributes [*] Ext2 POSIX Access Control Lists [*] Ext2 Security Labels <*> Ext3 journalling file system support [*] Ext3 extended attributes [*] Ext3 POSIX Access Control Lists [*] Ext3 Security Labels <*> The Extended 4 (ext4) filesystem [*] Ext4 POSIX Access Control Lists [*] Ext4 Security Labels	File System settings

Kernel Configuration Options	Description
[*] Enable the block layer ---> [*] Block layer bio throttling support IO Schedulers ---> <*> CFQ I/O scheduler> [*] CFQ Group Scheduling support	Block layer settings

Kernel Configuration Options	Description
`Kernel Features --->` `[*] Enable seccomp to safely compute untrusted bytecode`	Seccomp kernel support

11.4.2.4 Host Root Filesystem Configuration for Linux Containers

In order to use containers, mount the 'cgroup' pseudo-filesystem. When booting kernels compiled with cgroups support, there is a default directory for mounting them - /sys/fs/cgroup. If they are not already mounted, we will use this to mount our cgroup controllers.

```
mount -t cgroup cgroups /sys/fs/cgroup
```

11.4.3 More Details

11.4.3.1 LXC: Command Reference

This section contains links to available open source documentation for the commands in the LXC user space package.

For a description of the libvirt commands for managing containers see the chapter **Libvirt Users Guide**.

Table 593.

LXC man page	Description	Man Page Link
lxc	lxc overview	click here
lxc-attach	start a process inside a running container	click here
lxc-autostart	start/stop/kill auto-started containers	click here
lxc-cgroup	manage the control group associated with a container	click here
lxc-checkconfig	check the current kernel for lxc support	click here
lxc-clone	clone a new container from an existing one	click here
lxc-config	query LXC system configuration	click here
lxc.conf	a description of all configuration options available	LXC Configuration File Reference on page 2470
lxc-console	launch a console for the specified container	click here
lxc-create	creates a container	click here
lxc-destroy	destroy a container previously created with lxc-create	click here
lxc-execute	run the specified command inside a container	click here
lxc-freeze	freeze (suspend) all the container's processes	click here
lxc-info	query information about a container	click here
lxc-ls	list the containers existing on the system	click here

Table continues on the next page...

Table 593. (continued)

LXC man page	Description	Man Page Link
lxc-monitor	monitor the container state	click here
lxc-snapshot	snapshot an existing container	click here
lxc-start	starts a container previously created with lxc-create	click here
lxc-stop	stop a container	click here
lxc-unfreeze	resumes a containers processes suspended previously with lxc-freeze	click here
lxc-unshare	run a task in a new set of namespaces	click here
lxc-usernsexec	run task as root in a new user namespace	click here
lxc-wait	wait for a specific container state	click here

The following LXC commands are not supported:

- lxc-usernsexec

11.4.3.2 LXC: Configuration Files

NOTE

This section is applicable to LXC only, not to libvirt.

For LXC, configuration files are used to configure aspects of a container at the time it is created. The configuration file defines what resources are private to the container and what is shared. By default the following resources are private to a container:

- process IDs
- sysv ipc mechanisms
- mount points

This means for example, that by default the container will share network resources and the filesystem with the host system, but will have it's own private process IDs.

The container configuration file allows additional isolation to be specified through configuration in the following areas:

- network
- console
- mount points and the backing store for the root filesystem
- control groups (cgroups)
- POSIX capabilities

See the LXC Configuration File Reference on page 2470 for details on each configuration option.

When a container is created a new directory with the container's name is created in /var/lib/lxc. The configuration file for the container is stored in:

```
/var/lib/lxc/[container-name]/config
```

Below is an example of the contents of a minimal configuration file for a container named "foo", which has no networking:

```
$ cat /var/lib/lxc/foo/config
# Container with non-virtualized network
lxc.utsname = foo
```

```
lxc.tty = 1
lxc.pts = 1
lxc.rootfs = /var/lib/lxc/foo/rootfs
lxc.mount.entry=/lib /var/lib/lxc/foo/rootfs/lib none ro,bind 0 0
lxc.mount.entry=/usr/lib /var/lib/lxc/foo/rootfs/usr/lib none ro,bind 0 0
```

See the LXC: Getting Started (with a Busybox System Container) on page 2442 how-to article for an introduction to the container lifecycle and how configuration files are used when creating containers.

Several example configuration files are provided with LXC:

```
/usr/share/doc/lxc/examples/lxc-empty-netns.conf
/usr/share/doc/lxc/examples/lxc-complex.conf
/usr/share/doc/lxc/examples/lxc-no-netns.conf
/usr/share/doc/lxc/examples/lxc-vlan.conf
/usr/share/doc/lxc/examples/lxc-macvlan.conf
/usr/share/doc/lxc/examples/lxc-veth.conf
/usr/share/doc/lxc/examples/lxc-phys.conf
```

11.4.3.3 LXC: Templates

NOTE

This section is applicable to LXC only, not to libvirt.

For LXC, When a container is "created" a directory for the container (which has the same name as the container) is created under /var/lib/lxc. This is where the container's configuration file is stored and can be edited.

For system containers (containers created with **lxc-create**), the default is for the root filesystem structure of the container to be stored here as well.

Creating containers is simplified by the use of example "templates" provided with the LXC. Template examples are provided for a number of different Linux distributions. A template is a script invoked by **lxc-create** that creates the root filesystem structure and sets up the container's config file.

The following example templates are provided with LXC and can be referred to for the expected template structure:

```
/usr/share/lxc/templates/lxc-alpine
/usr/share/lxc/templates/lxc-altlinux
/usr/share/lxc/templates/lxc-archlinux
/usr/share/lxc/templates/lxc-busybox
/usr/share/lxc/templates/lxc-centos
/usr/share/lxc/templates/lxc-cirros
/usr/share/lxc/templates/lxc-debian
/usr/share/lxc/templates/lxc-download
/usr/share/lxc/templates/lxc-fedora
/usr/share/lxc/templates/lxc-gentoo
/usr/share/lxc/templates/lxc-openmandriva
/usr/share/lxc/templates/lxc-opensuse
/usr/share/lxc/templates/lxc-oracle
/usr/share/lxc/templates/lxc-plamo
/usr/share/lxc/templates/lxc-sshd
/usr/share/lxc/templates/lxc-ubuntu
/usr/share/lxc/templates/lxc-ubuntu-cloud
```

For the NXP Linux SDK for QorIQ the busybox template is recommended and has been tested with Yocto-created root filesystems.

The how-to examples provided in this user guide that create system containers use the busybox template.

11.4.3.4 Containers with Libvirt

This section provides an overview to using libvirt-based containers.

For an general introduction to libvirt, please see the chapter ***Libvirt Users Guide***. Also, see the container information available on the libvirt website: http://libvirt.org/drvlxc.html.

With libvirt, a container "domain" is specified in an XML file. The XML is used to "define" the container, which then allows the container to be managed with the standard libvirt domain lifecycle.

Libvirt XML

The XML for the simplest functional container would look like the example below:

```
<domain type='lxc'>
  <name>container1</name>
  <memory>500000</memory>
  <os>
    <type>exe</type>
    <init>/bin/sh</init>
  </os>
  <devices>
    <console type='pty'/>
  </devices>
</domain>
```

Refer to the XML reference information available on the libvirt website for detailed reference information: http://libvirt.org/formatdomain.html

The <domain> element must specify a type attribute of "lxc" for a container/lxc domain. There are 4 additional sub-nodes required:

- <name> - specifies the name of the container
- <memory> - specifies the maximum memory the container may use
- <os> - identifies the initial program to run. In the example this is /bin/sh. For an application based container this is the name of the application. If booting an instance of Linux user space this would typically by /sbin/init.
- <devices> - specifies any devices, in the above example there is just a console

To see a working example using this XML, see the how to article: Basic Example on page 2421.

Filesystem mounts (from http://libvirt.org/drvlxc.html)

In the absence of any explicit configuration, the container will inherit the host OS filesystem mounts. A number of mount points will be made read only, or re-mounted with new instances to provide container specific data. The following special mounts are setup by libvirt:

- /dev a new "tmpfs" pre-populated with authorized device nodes
- /dev/pts a new private "devpts" instance for console devices
- /sys the host "sysfs" instance remounted read-only
- /proc a new instance of the "proc" filesystem
- /proc/sys the host "/proc/sys" bind-mounted read-only
- /sys/fs/selinux the host "selinux" instance remounted read-only
- /sys/fs/cgroup/NNNN the host cgroups controllers bind-mounted to only expose the sub-tree associated with the container
- /proc/meminfo a FUSE backed file reflecting memory limits of the container

Additional filesystem mounts can be created using the <filesystem> node under the <devices> node. See the libvirt.org documentation referenced above for further details.

Device nodes from http://libvirt.org/drvlxc.html

The container init process will be started with CAP_MKNOD capability removed and blocked from re-acquiring it. As such it will not be able to create any device nodes in /dev or anywhere else in its filesystems. Libvirt itself will take care of pre-populating the /dev filesystem with any devices that the container is authorized to use. The current devices that will be made available to all containers are:

- /dev/zero
- /dev/null
- /dev/full
- /dev/random
- /dev/urandom
- /dev/stdin symlinked to /proc/self/fd/0
- /dev/stdout symlinked to /proc/self/fd/1
- /dev/stderr symlinked to /proc/self/fd/2
- /dev/fd symlinked to /proc/self/fd
- /dev/ptmx symlinked to /dev/pts/ptmx
- /dev/console symlinked to /dev/pts/0

11.4.3.5 Linux Control Groups (cgroups)

Linux control groups (or cgroups) is a feature of the Linux kernel that allows the allocation, prioritization,control, and monitoring of resources such as CPU time, memory, network bandwidth among groups of Linux processes.

Cgroups is one of the underlying Linux kernel features that LXC is built upon. LXC automatically creates a cgroup for each container when it is started. A pre-requisite for using LXC is mounting the cgroup virtual filesystem. Mounting the cgroup filesystem is presented in section Host Root Filesystem Configuration for Linux Containers on page 2436.

Cgroups encompass a number of different subsystems or "controllers" that are used for managing and controlling different resources. The following subsystems/controllers are supported:

- cpu - controls CPU allocation for tasks in a cgroup;
- cpuset - assigns individual CPUs and memory nodes to tasks in a cgroup;
- cpuacct - generates automatic reports on CPU resources used by the tasks in a cgroup;
- memory - isolates the memory behavior of a group of tasks from the rest of the system;
- devices - allows or denies access to devices by tasks in a crgroup;
- freezer - suspends or resumes tasks in a cgroup;
- net_cls - tags packets with a class identifier that allows the Linux traffic controller to identify packets originating from a particular cgroup;
- net_prio - provides a way to dynamically set the priority of network traffic per each network interface for applications within various cgroups;
- blkio - controls and monitors access to I/O on block devices by tasks in cgroups.

For an overview of cgoups, see the Linux kernel documentation overview here: Documentation/cgroups/cgroups.txt on page 2486.

You may also check out the Red Hat documentation on cgroups here: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch-Subsystems_and_Tunable_Parameters.html.

Cgroup subsystems can be configured within the configuration file used when creating a container. The configuration file accepts cgroup configuration in the following form:

```
lxc.cgroup.[subsystem name] = <value>
```

See the LXC Configuration File Reference on page 2470 for further details.

Cgroup subsystems can also be displayed or updated while a container is running using the **lxc-cgroup** command:

```
lxc-cgroup -n [container-name] [cgroup-subsystem] [value]
```

For some examples of how to use cgroups to control container configuration, see the article: LXC: How to use cgroups to manage and control a containers resources on page 2454.

11.4.3.6 Linux Namespaces

Linux namespaces is a feature in the Linux kernel that allows one to unshare and isolate a processes' resources like UTS, PID, IPC, file system mount and network from their parent. To achieve this the kernel places the resources in different namespaces.

When LXC spawns the container's main process it unshares all these resources except the network. The network is controlled from the configuration file and is shared by default.

A network namespace provides an isolated view of the networking stack (network device interfaces, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the /proc/net and /sys/class/net directory trees, sockets, etc.). A physical network device can live in exactly one network namespace. A virtual network device ("veth") pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace. When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace (not to the parent of the process).

Each namespace is documented in the Linux **clone** man page. See: clone (2)

11.4.3.7 POSIX Capabilities

Linux supports the concept of file "capabilities" which provides fine grained control over what executable programs are permitted to do. Instead of the "all or nothing" paradigm where a super-user or "root" has the power to perform all operations, capabilities provide a mechanism to grant a specific program specific capabilities.

LXC uses this feature of the kernel to implement containers. By default processes running in a container will have **all** capabilities, but this can be configured. Capabilities can be dropped in the container's configuration file. See LXC: Configuration Files on page 2437.

For example, to drop the CAP_SYS_MODULE, CAP_MKNOD, CAP_SETUID, and CAP_NET_RAW capabilities, the following configuration file options would be specified:

```
lxc.cap.drop = sys_module mknod setuid net_raw
```

Each capability is documented in the Linux **capabilities** man page. See: capabilities (7)

In order to fully isolate a container, the capabilities to be dropped must be carefully considered. The Linux Vserver project considers only the following capabilities as **safe** for virtual private servers:

```
CAP_CHOWN
CAP_DAC_OVERRIDE
CAP_DAC_READ_SEARCH
```

```
    CAP_FOWNER
    CAP_FSETID
    CAP_KILL
    CAP_SETGID
    CAP_SETUID
    CAP_NET_BIND_SERVICE
    CAP_SYS_CHROOT
    CAP_SYS_PTRACE
    CAP_SYS_BOOT
    CAP_SYS_TTY_CONFIG
    CAP_LEASE
```

(see: http://linux-vserver.org/Paper#Secure_Capabilities)

11.4.4 LXC: How To's

11.4.4.1 LXC: Getting Started (with a Busybox System Container)

The following article describes steps to run a simple container example. All the command below are issued from a host Linux command prompt.

1. Confirm that your kernel environment is configured correctly using **lxc-checkconfig**. All options should show as 'enabled.

```
# lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled

Note : Before booting a new kernel, you can check its configuration
usage : CONFIG=/path/to/config /usr/bin/lxc-checkconfig
```

If the cgroup namespace option shows as required:

```
Cgroup namespace: required
```

...most likely the /cgroup directory needs to be created and or mounted. See Host Root Filesystem Configuration for Linux Containers on page 2436.

NOTE: the User namespace is reported as missing. Although initial support for User Namespaces has been enabled in Linux 3.8, a lot of work still had to be done after this release and the USER_NS config flag could not be enabled. This has no impact on the functionality of containers, since User namespace support was not implemented - just the flag was there.

2. Create a container

   Create a system container using lxc-create and specify the busybox template and lxc-empty-netns.conf config file. lxc-empty-netns.conf is a simple config file with no networking:

```
# lxc-create -n foo -t busybox -f /usr/share/doc/lxc/examples/lxc-empty-netns.conf
setting root password to "root"
Password for 'root' changed
#
```

   By default, LXC will try to install the dropbear ssh utility, if it's available on the host system. The Busybox template also has support for installing OpenSSH (assuming it's installed on the host Linux) in the container. This needs to be passed explicitly using a command line parameter:

```
# lxc-create -n foo -t busybox -f /usr/share/doc/lxc/examples/lxc-empty-netns.conf -- -s openssh
setting root password to "root"
Password for 'root' changed
'OpenSSH' ssh utility installed
#
```

3. List containers that exist

```
# lxc-ls -l
drwxr-xr-x 3 root root 1024 May 30 15:37 foo
```

4. From a shell on the host Linux, start the container. When prompted, press 'Enter'.

```
# lxc-start -n foo -F

Please press Enter to activate this console.

/ #

/ #
```

   Note that the shell is now running within the container. Normal Linux commands can be executed.

   Important notice: while this mode starts the container and directly connects to one of its terminals, there is a minor caveat: the terminal will be stuck in this container console until the container is halted (either from here, by running **halt**, or from another terminal by running **lxc-stop**). In order to avoid this, there is also the possibility to start the container as a daemon and connect to it using **lxc-console** (this is the default mode). This provides better terminal capabilities and the user is not forced to stop the container from another terminal. On the other hand, there is no indication that after running **lxc-start** the container has actually started - no errors are reported. You must check if the container is running yourself, using **lxc-info** - see below.

```
# lxc-start -n foo
# lxc-console -n foo

Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

foo login: root
Password: (root)
~ #
~ #
```

```
~ #
~ # (Ctrl+a q)
#
```

This will be the preferred mode of starting and connecting to containers.

5. List processes in the container.

   From in the container shell use the ps command to list processes:

```
~ # ps
  PID USER       VSZ STAT COMMAND
    1 root      2384 S    init
    4 root      2384 S    /bin/syslogd
    6 root      2388 S    -sh
    7 root      2384 S    init
    8 root      2388 R    ps
```

   Note process IDs have a number-space unique to the container.

6. Show the status of the foo container (from a host shell):

```
# lxc-info -n foo
Name:           foo
State:          RUNNING
PID:            4544
CPU use:        0.01 seconds
Memory use:     472.00 KiB
KMem use:       0 bytes
```

7. Look at the files/directories in /var/lib/lxc related to the container

```
# ls -l /var/lib/lxc/foo
total 2
-rw-r--r--  1 root root  675 May 30 15:37 config
drwxr-xr-x 16 root root 1024 May 30 15:44 rootfs
```

   This shows the containers config file and rootfs backing store.

   Look at the contents of the config file:

```
# cat /var/lib/lxc/foo/config
# Template used to create this container: /usr/share/lxc/templates/lxc-busybox
# Parameters passed to the template:
# For additional config options, please look at lxc.conf(5)
lxc.utsname = omega
lxc.network.type = empty
lxc.network.flags = up
lxc.rootfs = /var/lib/lxc/foo/rootfs
lxc.haltsignal = SIGUSR1
lxc.utsname = foo
lxc.tty = 1
lxc.pts = 1
lxc.cap.drop = sys_module mac_admin mac_override sys_time

# When using LXC with apparmor, uncomment the next line to run unconfined:
#lxc.aa_profile = unconfined
lxc.mount.entry = /lib lib none ro,bind 0 0
lxc.mount.entry = /usr/lib usr/lib none ro,bind 0 0
```

```
lxc.mount.entry = /sys/kernel/security sys/kernel/security none ro,bind,optional 0 0
lxc.mount.auto = proc:mixed sys
```

8. Start a process inside the container using lxc-attach. This command will run the process inside the system container's isolated environment. The container has to be running already.

```
# lxc-attach -n foo -- /bin/sh
root@foo:/# ps
PID   USER     TIME   COMMAND
    1 root       0:00 init
    6 root       0:00 /bin/syslogd
    8 root       0:00 /bin/getty -L tty1 115200 vt100
    9 root       0:00 init
   10 root       0:00 /bin/sh
   11 root       0:00 ps
root@foo:/# ls -l /dev
total 0
crw-rw-rw-    1 root     5         136,   1 May 26 13:13 console
lrwxrwxrwx    1 root     root             13 May 26 13:12 fd -> /proc/self/fd
lrwxrwxrwx    1 root     root              7 May 26 13:13 kmsg -> console
srw-rw-rw-    1 root     root              0 May 26 13:13 log
crw-rw-rw-    1 root     root        1,    3 May 26 13:10 null
lrwxrwxrwx    1 root     root             13 May 26 13:12 ptmx -> /dev/pts/ptmx
drwxr-xr-x    2 root     root              0 May 26 13:13 pts
brw-------    1 root     root        1,    0 May 26 13:10 ram0
drwxrwxrwt    2 root     root             40 May 26 13:13 shm
lrwxrwxrwx    1 root     root             15 May 26 13:12 stderr -> /proc/self/fd/2
lrwxrwxrwx    1 root     root             15 May 26 13:12 stdin -> /proc/self/fd/0
lrwxrwxrwx    1 root     root             15 May 26 13:12 stdout -> /proc/self/fd/1
crw-rw-rw-    1 root     root        5,    0 May 26 13:10 tty
crw-rw-rw-    1 root     root        4,    0 May 26 13:10 tty0
crw--w----    1 root     root      136,    0 May 26 13:13 tty1
crw-rw-rw-    1 root     root        4,    0 May 26 13:10 tty5
crw-rw-rw-    1 root     root        1,    9 May 26 13:10 urandom
crw-rw-rw-    1 root     root        1,    5 May 26 13:10 zero
root@foo:/#
```

9. Stop the container (from a host shell)

```
# lxc-stop -n foo
#
# lxc-info -n foo
Name:           foo
State:          STOPPED
```

10. Destroy the container. This removes the containers config file and backing store.

```
# lxc-destroy -n foo
#
```

11.4.4.2 LXC: How to configure non-virtualized networking (lxc-no-netns.conf)

One approach to providing networking capability to a container is to simply allow the container to use existing host network interfaces. To accomplish this, a configuration file is created with no networking setup (i.e. the **lxc.network.type** property is not set) and the default will be to allow the container to access the host's networking interfaces.

With this approach no network namespace is created for the container.

An example config is provided:

```
/usr/share/doc/lxc/examples/lxc-no-netns.conf
```

The contents of lxc-no-netns.conf look like this:

```
# Container with non-virtualized network
lxc.network.type = none
lxc.utsname = delta
```

The example below shows starting an application container (running bash) with this config file and shows that the host network interface fm2-mac5 is inherited and accessible by the container:

```
# lxc-execute -n mytest -f /usr/share/doc/lxc/examples/lxc-no-netns.conf -- /bin/bash
bash-4.3# ifconfig
fm2-mac5  Link encap:Ethernet  HWaddr 00:04:9f:02:7a:3b
          inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::204:9fff:fe02:7a3b/64 Scope:Link
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:508 (508.0 B)
          Memory:fe5e8000-fe5e8fff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:272 (272.0 B)  TX bytes:272 (272.0 B)
bash-4.2#
```

11.4.4.3 LXC: How to assign a physical network interface to a container (lxc-phys.conf)

One approach to providing networking capability to a container is to directly assign an available, unused network interface to the container. The interface is not shared, it becomes the private resource of the container.

An example LXC configuration file is provided to configure this type of networking:

```
    /usr/share/doc/lxc/examples/lxc-phys.conf
```

The contents of the default lxc-phys.conf example are show below:

```
# Container with network virtualized using a physical network device with name
# 'eth0'
lxc.utsname = gamma
lxc.network.type = phys
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:ff
```

```
lxc.network.ipv4 = 10.2.3.6/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
```

Note: The network type is set to: **phys**. Make a copy of the example config file and update it with the name of the Ethernet interface to be assigned, an appropriate IP address, and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to set the interface fm2-gb0 and IP address 192.168.10.3 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-phys.conf
+++ lxc-phys.conf
@@ -3,7 +3,6 @@
 lxc.utsname = gamma
 lxc.network.type = phys
 lxc.network.flags = up
-lxc.network.link = eth0
-lxc.network.hwaddr = 4a:49:43:49:79:ff
-lxc.network.ipv4 = 10.2.3.6/24
-lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
+lxc.network.link = fm2-mac5
+lxc.network.hwaddr = 00:e0:0c:00:93:05
+lxc.network.ipv4 = 192.168.10.3/24
```

A simple way to test the new config file and the network interface is to run /bin/bash as a command with lxc-execute, which will provide a shell running in the container:

```
# lxc-execute -n mytest -f lxc-phys.conf -- /bin/bash
bash-4.2#
```

In the container, use the fm1-gb4 interface normally:

```
bash-4.3# ifconfig
fm2-mac5  Link encap:Ethernet  HWaddr 00:e0:0c:00:93:05
          inet addr:192.168.10.3  Bcast:192.168.10.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:508 (508.0 B)
          Memory:fe5e8000-fe5e8fff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

bash-4.2# ping -c 3 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_req=1 ttl=64 time=0.385 ms
64 bytes from 192.168.10.1: icmp_req=2 ttl=64 time=0.207 ms
64 bytes from 192.168.10.1: icmp_req=3 ttl=64 time=0.187 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.187/0.259/0.385/0.090 ms
```

11.4.4.4 LXC: How to configure networking with virtual Ethernet pairs (lxc-veth.conf)

One approach to providing a virtual network interface to a container is using the "Virtual ethernet pair device" feature of the Linux kernel in conjunction with a network bridge.

See the veth description in LXC Configuration File Reference on page 2470 for additional details on this approach to networking.

With this approach LXC creates a new network namespace for the container.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-veth.conf
```

The contents of the default lxc-veth.conf example are show below:

```
# Container with network virtualized using a pre-configured bridge named br0 and
# veth pair virtual network devices
lxc.utsname = beta
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.hwaddr = 4a:49:43:49:79:bf
lxc.network.ipv4 = 10.2.3.5/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

Note, the network type value is: **veth** and the link property value is **br0**.

First, create a network bridge which is attached to a physical network interface and assign the bridge an IP address. The bridge becomes one endpoint In the example below the bridge br0 is created, interface fm2-gb1 is added to it, and the bridge is assigned an IP address of 192.168.20.2.

```
# brctl addbr br0
# ifconfig br0 192.168.20.2 up
# ifconfig fm2-mac5 up
# brctl addif br0 fm2-mac5
```

Make a copy of the example config file and update it with an appropriate IP address and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to update the IP address to 192.168.20.3 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-veth.conf
+++ lxc-veth.conf
@@ -5,5 +5,5 @@
 lxc.network.flags = up
 lxc.network.link = br0
 lxc.network.hwaddr = 4a:49:43:49:79:bf
-lxc.network.ipv4 = 10.2.3.5/24
+lxc.network.ipv4 = 192.168.20.3/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

A simple way to test the new config file and the network interface is to run /bin/bash as a command with lxc-execute, which will provide a shell running in the container:

```
# lxc-execute -n mytest -f lxc-veth.conf -- /bin/bash
bash-4.2#
```

In the container, use the virtual network interface (eth0 in this example) normally:

```
bash-4.2# ifconfig
eth0      Link encap:Ethernet  HWaddr 4a:49:43:49:79:bf
          inet addr:192.168.20.3  Bcast:192.168.20.255  Mask:255.255.255.0
          inet6 addr: fe80::4849:43ff:fe49:79bf/64 Scope:Link
          inet6 addr: 2003:db8:1:0:214:1234:fe0b:3597/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:468 (468.0 B)  TX bytes:586 (586.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

bash-4.2# ping -c 3 192.168.20.1
PING 192.168.20.1 (192.168.20.1) 56(84) bytes of data.
64 bytes from 192.168.20.1: icmp_req=1 ttl=64 time=0.433 ms
64 bytes from 192.168.20.1: icmp_req=2 ttl=64 time=0.221 ms
64 bytes from 192.168.20.1: icmp_req=3 ttl=64 time=0.228 ms

--- 192.168.20.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.221/0.294/0.433/0.098 ms
```

11.4.4.5 LXC: How to configure networking with macvlan (lxc-macvlan.conf)

An LXC container can be provided with a virtual network interface using the "MAC-VLAN" feature of the Linux kernel (see kernel config option CONFIG_MACVLAN). MAC-VLAN allows virtual interfaces to be created that route packets to or from a MAC address to a physical network interface.

See the macvlan description in LXC Configuration File Reference on page 2470 for some additional details on this approach to networking.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-macvlan.conf
```

The contents of the provided lxc-phys.conf example configuration file are show below:

```
# Container with network virtualized using the macvlan device driver
lxc.utsname = alpha
lxc.network.type = macvlan
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Make a copy of the example config file and update it with the physical network interface to be used, an appropriate IP address, and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to specify the fm1-gb4 interface and update the IP address to 192.168.1.24 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-macvlan.conf
+++ lxc-macvlan.conf
@@ -2,7 +2,7 @@
 lxc.utsname = alpha
 lxc.network.type = macvlan
 lxc.network.flags = up
-lxc.network.link = eth0
+lxc.network.link = fm2-mac5
 lxc.network.hwaddr = 4a:49:43:49:79:bd
-lxc.network.ipv4 = 10.2.3.4/24
+lxc.network.ipv4 = 192.168.10.3/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Put the network interface in promiscuous mode:

```
# ifconfig fm2-mac5 promisc
# ifconfig fm2-mac5
fm2-gb0   Link encap:Ethernet  HWaddr 00:e0:0c:00:93:05
          inet addr:192.168.10.2  Bcast:192.168.10.255  Mask:255.255.255.0
          inet6 addr: fe80::2e0:cff:fe00:9305/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:5 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:344 (344.0 B)  TX bytes:1314 (1.2 KiB)
          Memory:fe5e0000-fe5e0fff
```

Test the MAC-VLAN interface by starting an application container running /bin/bash:

```
# lxc-execute -n mytest -f lxc-macvlan.conf -- /bin/bash
bash-4.2#
```

Note: the shell prompt above ("bash-4.2") is in the container.

Test the interface in the now running container:

```
bash-4.2# ifconfig
eth0      Link encap:Ethernet  HWaddr 4a:49:43:49:79:bd
          inet addr:192.168.10.3  Bcast:192.168.10.255  Mask:255.255.255.0
          inet6 addr: fe80::4849:43ff:fe49:79bd/64 Scope:Link
          inet6 addr: 2003:db8:1:0:214:1234:fe0b:3596/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:586 (586.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

```
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

bash-4.2# ping -c 3 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_req=1 ttl=64 time=0.380 ms
64 bytes from 192.168.10.1: icmp_req=2 ttl=64 time=0.204 ms
64 bytes from 192.168.10.1: icmp_req=3 ttl=64 time=0.201 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.201/0.261/0.380/0.085 ms
```

11.4.4.6 LXC: How to configure networking using a VLAN (lxc-vlan.conf)

A container can be provided with a virtual network interface using VLANs.

See the vlan description in LXC Configuration File Reference on page 2470 for some additional details on this approach to networking.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-vlan.conf
```

The contents of the provided lxc-vlan.conf example configuration file are show below:

```
 # Container with network virtualized using the vlan device driver
lxc.utsname = alpha
lxc.network.type = vlan
lxc.network.vlan.id = 1234
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Make a copy of the example config file and update it with the physical network interface to be used and the vlan ID, an appropriate IP address, and any other appropriate changes. For example, the change (in universal diff format) to specify the fm2-gb0 interface, a VLAN id of 2, and an IP address of 192.168.30.2 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-vlan.conf   2013-05-30 14:22:14.980406375 +0300
+++ lxc-vlan.conf       2013-06-03 13:26:38.477580000 +0300
@@ -1,9 +1,9 @@
 # Container with network virtualized using the vlan device driver
 lxc.utsname = alpha
 lxc.network.type = vlan
-lxc.network.vlan.id = 1234
+lxc.network.vlan.id = 2
 lxc.network.flags = up
-lxc.network.link = eth0
+lxc.network.link = fm2-mac5
 lxc.network.hwaddr = 4a:49:43:49:79:bd
-lxc.network.ipv4 = 10.2.3.4/24
+lxc.network.ipv4 = 192.168.30.2/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

In this setup, the host is connected to a test machine through physical interface fm2-gb0. On the test machine, the following commands have been issued (interface p7p1 on this machine has physical link to fm2-gb0):

```
[root@everest][~]# modprobe 8021q
[root@everest][~]# lsmod | grep 8021q
8021q                  23476  0
garp                   13763  1 8021q
[root@everest][~]# vconfig add p7p1 2
Added VLAN with VID == 2 to IF -:p7p1:-
[root@everest][~]# ifconfig p7p1.2 192.168.30.1 up
```

Test the VLAN interface by starting an application container running /bin/bash:

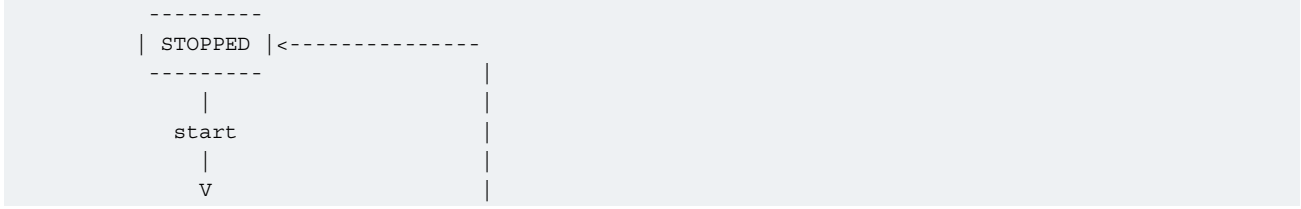
```
# lxc-execute -n mytest -f lxc-vlan.conf -- /bin/bash
bash-4.2#
```

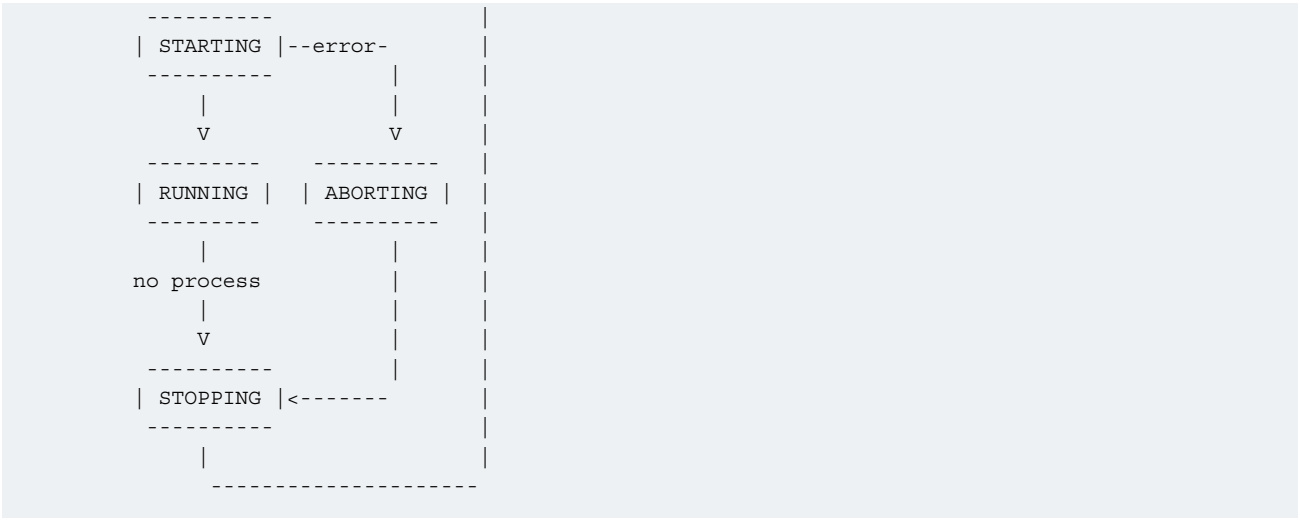
Test the interface in the now running container:

```
bash-4.2# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.30.2  netmask 255.255.255.0  broadcast 192.168.30.255
        inet6 fe80::21e:c9ff:fe49:bb93  prefixlen 64  scopeid 0x20<link>
        ether 00:1e:c9:49:bb:93  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6  bytes 468 (468.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 16436
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 4  bytes 200 (200.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 200 (200.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

bash-4.2# ping -c 3 192.168.30.1
PING 192.168.30.1 (192.168.30.1) 56(84) bytes of data.
64 bytes from 192.168.30.1: icmp_req=1 ttl=64 time=0.338 ms
64 bytes from 192.168.30.1: icmp_req=2 ttl=64 time=0.372 ms
64 bytes from 192.168.30.1: icmp_req=3 ttl=64 time=0.355 ms

--- 192.168.30.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.338/0.355/0.372/0.013 ms
```

11.4.4.7 LXC: How to monitor containers

Containers transition through a set of well defined states. After a container is created it is in the "stopped" state.





A number of commands are available in LXC to monitor the state of a container. The following examples provide an introduction and demonstrate the capabilities of these commands.

1. lxc-info

   The lxc-info command shows the current state of the container.

   In the example below, a container called "foo" has already been created but not started and the container is stopped:

```
# lxc-info -n foo
Name:           foo
State:          STOPPED
After the container is started lxc-info shows the container in the running state:
```

```
# lxc-start -n foo
# lxc-info -n foo
Name:           foo
State:          RUNNING
PID:            5075
CPU use:        0.01 seconds
Memory use:     508.00 KiB
KMem use:       0 bytes
```

2. lxc-monitor

   The **lxc-monitor** command can monitor the state of one or more containers, the command continues to run until it is killed.

   In this example **lxc-monitor** monitors the state of a container named "foo":

```
# lxc-monitor -n foo
```

   In a separate shell, start and then stop the container foo:

```
# lxc-start -n foo
# lxc-stop -n foo
```

   The running **lxc-monitor** command displays the state changes as they occur:

```
'foo' changed state to [STARTING]
'foo' changed state to [RUNNING]
```

```
'foo' changed state to [STOPPING]
'foo' changed state to [STOPPED]
```

3. lxc-wait

   The lxc-wait command will wait for a container state change and then exit. This can be useful for scripting and synchonizing the start or exit of a container.

   For example, to wait until the container named "foo" stops:

```
# lxc-wait -n foo -s STOPPED
```

11.4.4.8 LXC: How to modify the capabilities of a container to provide additional isolation

As described in POSIX Capabilities on page 2441 ,by default processes running in a container will have all capabilities. And the configuration for a container can further restrict these capabilities.

This example shows how to remove the ability for a container to issue the mknod command.

By default a container can issue the mknod command:

```
~ # mknod zero c 1 5
~ # ls -l zero
crw-r--r--    1 root     root        1,   5 Jun  3 17:08 zero
```

In this example we modify the config file of a container named "foo" (/var/lib/lxc/foo/config) and specify in the **lxc.cap.drop** property that the mknod capability (CAP_MKNOD) should be removed:

```
@@ -5,6 +5,7 @@
 lxc.utsname = foo
 lxc.tty = 1
 lxc.pts = 1
+lxc.cap.drop = mknod
 lxc.rootfs = /var/lib/lxc/foo/rootfs
 lxc.mount.entry=/lib /var/lib/lxc/foo/rootfs/lib none ro,bind 0 0
 lxc.mount.entry=/usr/lib /var/lib/lxc/foo/rootfs/usr/lib none ro,bind 0 0
```

Now restart the container and the mknod operation is no longer permitted:

```
~ # mknod zero c 1 5
mknod: zero: Operation not permitted
```

11.4.4.9 LXC: How to use cgroups to manage and control a containers resources

This example demonstrates how to use control croups to control which CPU's a container is scheduled on and the percentage of CPU time allocated to a container.

In this example we'll examine and change:

- the **cpuset** subsystem's **cpus** parameter which controls which physical CPUs the container's processes will run on
- the **cpu** subsystem's **shares** parameter which controls the percentage of the CPU to be allocated to the container

1. Start two application containers each running /bin/bash:

First container:

```
# lxc-execute -n foo1 -f lxc-no-netns.conf -- /bin/bash
bash-4.2#
```

Second container:

```
# lxc-execute -n foo2 -f lxc-no-netns.conf -- /bin/bash
bash-4.2#
```

2. In both containers start a process that will put a 100% load on the CPUs:

```
(while true; do true; done) &
```

3. The **cpuset.cpus** subsystem/value specifies which physical CPUs the container's processes run on. From a host shell, examine this with the **lxc-cgroup** command:

```
# lxc-cgroup -n foo1 cpuset.cpus
0-7
```

In this example the host system has 4 CPUs.

This can also be seen directly through the /cgroup filesystem:

```
# cat /cgroup/cpuset/lxc/foo1/cpuset.cpus
0-7
```

4. Change both containers to run only on CPU 2:

```
# lxc-cgroup -n foo1 cpuset.cpus 2
# lxc-cgroup -n foo2 cpuset.cpus 2
#
```

The top command now shows CPU 2 with 100% utilization. The bash commands running in each container, each have about 50% of the CPU:

```
top - 17:14:41 up 10 min,  4 users,  load average: 1.64, 0.61, 0.23
Tasks: 100 total,   3 running,  97 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.3%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2 :100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3996400k total,   189836k used,  3806564k free,     1652k buffers
Swap:        0k total,        0k used,        0k free,    26180k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2875 root      20   0  3624  416  164 R   50  0.0   1:28.12 bash
 2874 root      20   0  3624  424  168 R   50  0.0   1:31.06 bash
```

5. The cpu.shares subsystem/value specifies the percentage of the CPU allocated to the cgroup/container. By default each container has a shares value of 1024:

```
# lxc-cgroup -n foo1 cpu.shares
1024
```

```
# lxc-cgroup -n foo2 cpu.shares
1024
```

6. Change container "foo2" to have about 10% of the CPU:

```
# lxc-cgroup -n foo2 cpu.shares 100
# lxc-cgroup -n foo1 cpu.shares 900
```

Now the top command output reflects the new CPU allocation:

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2874 root      20   0  3624  424  168 R   90  0.0   2:53.63 bash
 2875 root      20   0  3624  416  164 R   10  0.0   2:11.36 bash
```

7. Stop the containers

```
# lxc-stop -n foo1 -k
# lxc-stop -n foo2 -k
#
```

11.4.4.10 LXC: How to run an application in a container with lxc-execute

The **lxc-execute** command allows a single application to be run in a container (as contrasted with a system container which boots an instance of Linux user space starting with System V style init).

In the example below an instance of a QEMU/KVM virtual machine is started in a container called foo.

Note, it is not required to explicitly create (and destroy) a container when running application containers with lxc-execute. The containers will automatically created and destroyed.

1. Start QEMU in the container with lxc-execute:

```
# lxc-execute -n foo -f lxc-no-netns.conf -- qemu-system-ppc -enable-kvm -smp 2 -m 256M -nographic
-M ppce500 -kernel uImage -initrd rootfs.ext2.gz -append "root=/dev/ram rw console=ttyS0,115200" -
serial tcp::4445,server,telnet
```

NOTE: For 64bit platforms, please replace qemu-system-ppc with qemu-system-ppc64.

Some notes:

- The QEMU command line follows the double dash ("--") specfied on the lxc-execute command line and distinguishes argument to lxc-execute from arguments to qemu-system-ppc.
- Using the specified configuration file, QEMU will run in the network namespace of the host system, meaning the TCP ports for serial and the monitor (ports 4445 and 4446) can be accessed from the host. However, lxc-execute will accept a configuration file as an argument allowing customization of the degree of isolation of the container.
- In this example there are 2 virtual cpus specified, which results in a total of 3 QEMU processes/threads. So we expect to see 3 QEMU processes in the container.

2. Examine the state of the container with lxc-ls and lxc-info:

```
# lxc-ls --active
foo

# lxc-info -n foo
Name:           foo
State:          RUNNING
```

```
PID:            3205
IP:             192.168.2.80
CPU use:        3.96 seconds
Memory use:     140.46 MiB
KMem use:       0 bytes
```

3. In the QEMU console look at the CPU status which shows the process IDs for the two virtual CPUs in in the virtual machine:

```
# (qemu) info cpus
* CPU #0: nip=0x00000000c001450c thread_id=4
  CPU #1: nip=0x00000000c001450c thread_id=5
(qemu)
```

Note that the process/thread IDs as viewed from within the container (thread IDs 4 and 5) are different than from the host, since they are in a different namespace.

5. Using the container's cgroup restrict the physical CPUs on which the virtual machine is allowed to run.

By default all 4 CPUs can be used by the container :

```
# by default all 4 CPUs can be used by the container
# cat /cgroup/lxc/foo/cpuset.cpus
0-3
```

Restrict the containers processes to CPUs 2 and 3:

```
# echo 2-3 > /cgroup/lxc/foo/cpuset.cpus
# cat /cgroup/lxc/foo/cpuset.cpus
2-3
```

11.4.4.11 LXC: How to run an unprivileged container

With the addition of the user namespace in the Linux kernel, a normal user on a Linux host can create and run container instances. This feature has been integrated in the LXC package, starting from version 1.0.

The steps below detail the necessary steps required in order to configure and manage an unprivileged container.

NOTE: Before running these steps, make sure that the host is properly configured for container use, by running **lxc-checkconfig** (cgroups, namespaces, etc.). If some of the options are missing, please refer to the guidelines in LXC: Build, Installation, Configuration on page 2433.

1. Create the **/etc/subuid** and **/etc/subgid** file on the Linux host. These will be used to store the unprivileged user's subordinate UIDs and GIDs. The unprivileged user has the ability to manage users on his own in his user namespace, and their IDs will be mapped to corresponding ranges on IDs on the host system. The subordinate IDs will correspond to the ranges defined in these files.

   ```
   for file in '/etc/subuid' '/etc/subgid'; do
       touch $file
       chown root:root $file
       chmod 644 $file
   done
   ```

2. Add a user in the system - **lxc-user**.

   ```
   useradd lxc-user -p $(echo test | openssl passwd -1 -stdin)
   ```

3. Check the contents of **/etc/subuid** and **/etc/subgid**. If they contain the following entries, the user has been automatically assigned a default set of subordinate IDs.

```
root@t4240qds:~# cat /etc/sub*
lxc-user:100000:65536
lxc-user:100000:65536
root@t4240qds:~#
```

   If the files are empty, you need to manually assign a set of subordinate IDs to the user.

```
usermod --add-subuids 100000-165536 lxc-user
usermod --add-subgids 100000-165536 lxc-user
```

4. The container will have a virtual interface linked to a bridge on the host. Use the following command to create the bridge.

```
brctl addbr br0 && ifconfig br0 10.0.0.1
```

5. You must create and edit the **/etc/lxc/lxc-usernet** file. This file specifies how many interfaces the lxc-user will be allowed to have linked in this bridge.

```
echo "lxc-user veth br0 10" > /etc/lxc/lxc-usernet
```

6. Create the **/home/lxc-user/.config/lxc** directory on the host. This will hold the default configuration for unprivileged containers belonging to the lxc-user.

```
mkdir -p /home/lxc-user/.config/lxc
```

7. **Create** the default container configuration file, **/home/lxc-user/.config/lxc/default.conf**, and paste the following lines.

```
lxc.network.type = veth
lxc.network.link = br0
lxc.network.flags = up
lxc.id_map = u 0 100000 65536
lxc.id_map = g 0 100000 65536
```

8. Change the ownership of the newly created files and folders to lxc-user.

```
chown -R lxc-user:lxc-user /home/lxc-user/.config
```

9. For each of the mounted cgroup controllers, created a directory in the top called **lxc-user**, and change its ownership to lxc-user. Be sure to enable the cgroup.clone_children and memory.use_hierarchy flags.

```
echo 1 > /sys/fs/cgroup/memory/memory.use_hierarchy

for c in `ls /sys/fs/cgroup/`; do
    echo 1 > /sys/fs/cgroup/$c/cgroup.clone_children
    mkdir /sys/fs/cgroup/$c/lxc-user
    chown -R lxc-user:lxc-user /sys/fs/cgroup/$c/lxc-user
done
```

10. **Login as the new user in a new console.**

```
t4240qds login: lxc-user
Password:
t4240qds:~$
```

11. Copy the shell PID in the lxc-user cgroups.

```
for c in `ls /sys/fs/cgroup/`; do
    echo $$ > /sys/fs/cgroup/$c/lxc-user/tasks
done
```

12. From the same shell as before, create a Busybox container. You can pass it a custom config file using the **-f** cmdline parameter. Otherwise, it will pick the default config from **/home/lxc-user/.config/default.conf**.

```
t4240qds:~$ lxc-create -n foo -t busybox
setting root password to "root"
Password for 'root' changed
t4240qds:~$
```

13. Start the container.

```
t4240qds:~$ lxc-start -n foo -F

Please press Enter to activate this console.
/ #
/ #
/ # whoami
root
/ #
```

Now you can interact with the container as you would with one created by root. **Make sure that all container commands are run as lxc-user.**

11.4.4.12 LXC: How to run containers with Seccomp protection

A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. As system calls change and mature, bugs are found and eradicated. A certain subset of userland applications benefit by having a reduced set of available system calls. The resulting set reduces the total kernel surface exposed to the application. System call filtering is meant for use with those applications.

Seccomp (short for *secure compute*) is a system call filtering mechanism present in the kernel. Initially it has been thought to be a sandboxing mechanism that would allow userspace processes to issue a very limited set of system calls - read(), write(), exit() and sigreturn(). This has been further known to be **seccomp mode 1**, and while it is strong on the security side, it doesn't leave much room for flexibility.

The next addition to seccomp was to allow filtering (or **seccomp mode 2**) based on the kernel BPF (Berkeley Packet Filter) infrastructure. This allows the system administrator to define complex and granular policies, per system call and its arguments. This is an extension to the BPF mechanism, that allows filtering to apply to system call numbers and their arguments, besides its original purpose (socket packets). The defined filter results in a seccomp policy which is attached to the userspace process in the form of a BPF program. Each time the process issues a system call, it is checked against this policy in order to determine how it will be handled:

- **SECCOMP_RET_KILL** - the task exits immediately without executing the system call.
- **SECCOMP_RET_TRAP** - the kernel sends a SIGSYS to the triggering task without executing the system call.
- **SECCOMP_RET_ERRNO** - a custom errno is returned to userspace without executing the system call.
- **SECCOMP_RET_TRACE** - causes the kernel to attempt to notify a ptrace-based tracer prior to executing the system call. The tracer can skip the system call or change it to a valid syscall number.
- **SECCOMP_RET_ALLOW** - results in the system call being executed.

In order to make the secure computing facility more userspace-friendly, the libseccomp library has been developed, which is meant to make it easier for applications to take advantage of the packet-filter-based seccomp mode. Prior to this, userspace applications had to define the BPF filter themselves. libseccomp restructures this approach into a simple and straightforward

API which userspace applications can use. The latest version of libseccomp adds support for Python bindings as well, and is designed to work on multiple architectures (ARM, MIPS). PowerPC support has also been merged on a separate branch, and is expected to be included in future releases.

Using seccomp with LXC containers

Please refer to Building LXC for information on how to build LXC with seccomp support in the SDK.

Note: Currently LXC seccomp support is not available for ARM64 architectures.

Seccomp filtering integrates well with processes sandboxed as containers, as they can be assigned to untrusted users and exposed with a limited set of allowed system calls. This is a portable and granular low-level security mechanism which can be used to increase container security. The seccomp policy file needs to be applied only to the init process in the container, and will be inherited by all its children.

The seccomp policy for the container is specified using the container configuration file, in the form of a single line containing:

```
lxc.seccomp = /var/lib/lxc/lxc_seccomp.conf
```

An example lxc_seccomp policy file can look as follows:

```
2
blacklist
[ppc64]
mknod errno 120
sched_setscheduler trap
fchmodat kill
[ppc]
mknod
```

The elements in the policy file represent the following:

1. Version number (1/2) - a single integer containing a single number, 1 or 2. Version 1 only allows to define a set of system calls which are allowed (whitelisted) in the container, specified by syscall number. This version is limited in configurability and portability, since it's only used to specify allowed syscall numbers, which may differ from arch to arch. Version 2 allows the policies to be either a whitelist (default deny, except mentioned syscalls) or a blacklist (default allow, except mentioned syscalls), and the syscalls can be expressed by name.
2. Policy type (whitelist/blacklist) - with an option of a default policy action (errno #, trap, kill, allow). The policy type is per seccomp context, and can be either whitelist or blacklist, not both.
3. Architecture tag [optional] - mentions that the following set of system calls will only be applied to a specific architecture. There can be multiple architecture tags and associated syscalls. These tags allow the same seccomp policy file to be used on multiple platforms, treating each one differently with respect to the set of system calls.
4. System calls - which can be expressed by number (in version 1) or name (in version 2). Optionally, an action can be expressed after the system call (errno #, trap, kill, allow), specifying the desired seccomp behavior. If this is omitted, the dafault rule action of the policy will be applied (allow for whitelist policies, kill for blacklist policies).

When running a container with the previous policy file on a PowerPC 64-bit platform, the mknod, sched_setscheduler (chrt) and fchmodat (chmod) system calls will be denied, with mentioned behaviors: mknod will return errno 120 without executing, chrt will trap and chmod will result in the process executing it being killed. On PowerPC platforms, only mknod will be denied, resulting in the process being killed. All other system calls will be allowed.

Notes:

- Containers can still be started without loading a seccomp policy file, simply by omitting the lxc.seccomp line in the config file. No seccomp policy is loaded by default.

- If a container process has a seccomp policy loaded, this can be seen in /proc/PID/status, on the seccomp line. This line will contain "Seccomp: 2" when using seccomp filter (mode 2). "Seccomp: 0" means there is no seccomp policy in effect.
- Seccomp policies of a process are automatically inherited by its children.
- Currently LXC supports only system call based filtering, with no support for system call arguments.
- The performance degradation of the processes running with a seccomp policy applied is directly proportional with the policy file size: normally, the system calls are listed as rules in the BPF filter program, and they all need to be parsed and matched at each system call. The longer the list, the more time this will take.
- The LXC package comes shipped with a set of example policy files which can be found at /share/doc/lxc/examples/seccomp-*. There's also a policy file, common.seccomp, which denies common security syscall threats in the container, such as kernel module manipulation, kexec and open_by_handle_at (the vector for the Shocker exploit).

11.4.5 Libvirt How To's

11.4.5.1 Basic Example

The following example shows the lifecycle of a simple LXC libvirt domain called *container1*. The virsh tool is used for managing the lxc domain lifecycle.

1. Confirm the host Linux configuration. Begin by confirming that the host kernel is configured correctly and that rootfs setup such as mounting cgroups has been done. This can be done with the lxc-checkconfig command.

```
# lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

2. **Create a libvirt XML file defining the container**. The example below shows a very simple container defined in container1.xml that runs the command /bin/sh and has a console:

```
# cat container1.xml
<domain type='lxc'>
  <name>container1</name>
  <memory>500000</memory>
  <os>
    <type>exe</type>
```

```
    <init>/bin/sh</init>
  </os>
  <devices>
    <console type='pty'/>
  </devices>
</domain>
```

3. **Define the container.** The *virsh define* command processes the XML and makes creates the new libvirt domain.

```
# virsh -c lxc:/// define container1.xml
Domain container1 defined from container1.xml

# virsh -c lxc:/// list --all
 Id    Name                           State
----------------------------------------------------
 -     container1                     shut off
```

4. Start the container.

```
# virsh -c lxc:/// start container1
Domain container1 started

# virsh -c lxc:/// list
 Id    Name                           State
----------------------------------------------------
 3196  container1                     running
```

5. Connect to the console.

```
# virsh -c lxc:/// console container1
Connected to domain container1
Escape character is ^]
sh-4.2#
sh-4.2# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 18:25 pts/2    00:00:00 /bin/sh
root         3     1  0 18:36 pts/2    00:00:00 ps -ef

sh-4.2# ifconfig br0
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.171.73.123  netmask 255.255.254.0  broadcast 10.171.73.255
        inet6 fe80::a00:27ff:fe01:fe07  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:01:fe:07  txqueuelen 0  (Ethernet)
        RX packets 865838  bytes 104029354 (99.2 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 104446  bytes 43998714 (41.9 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

sh-4.2#
```

Press CTRL +] to exit the console.

The following aspects must be noted:

- the processes inside the container are running in a separate namespace, hence the different process hierarchy
- since no network configuration for the domain is explicitly specified, all networking interfaces are shared with the host (all the other interfaces are present too - br0 is mentioned as an example)

- since no filesystem configuration is specified for the domain, the filesystem is shared with the host-- all host mounts are present in the container as well.

6. To stop the container use the destroy command:

```
# virsh -c lxc:/// destroy container1
Domain container1 destroyed

# virsh -c lxc:/// list  --all
 Id    Name                           State
----------------------------------------------------
 -     container1                     shut off
```

7. To remove the domain from libvirt, use the undefine command.

```
# virsh -c lxc:/// undefine container1
Domain container1 has been undefined
```

11.4.6 USDPAA in LXC

This section describes running the USDPAA driver software in LXC containers. It presents the configuration steps for this scenario, along with observed issues and raised questions. Running USDPAA in containers is intended to be a proof of concept and a supported feature for the QorIQ SDK starting from v1.4. USDPAA is only available for DPAA v1.x platforms.

The P4080DS RefMan was also used to determine the proper SerDes Lane Configuration for the test board.

11.4.6.1 General Setup

Prerequisites

The test environment consisted of the following components in the **FSR Boardfarm**:

- a P4080DS board - **dumbo**
- a Linux test machine - **zro04qoriq02** with 2 test ports

The **dumbo** test board presents the following port configuration:

```
[b43198@zro04qoriqsrv ~]$ lars info dumbo
Name          : dumbo
Template      : P4080DS
Category      : board
Status        : reserved
Current user  : Purcareata Bogdan-B43198
Usable        : True

Station    : <Station(176)>
Console    : telnet://10.171.77.74:51201
Power      : adam5000://10.171.77.131/18
Reset      : adam5000://10.171.77.131/18
JTAG       : 10161212

Installed devices:
dumbo.duart_p1  : PORT
dumbo.rgmii_p1  : SERVICE
dumbo.sgmii_s3p1: MCCPORT
dumbo.sgmii_s3p2: MCCPORT
dumbo.sgmii_s3p3: SWITCH
```

```
dumbo.usb20_p1   : PORT
dumbo.xaui_s4p1 : PORT
```

The board exposes 2 SGMII ports available for testing, managed by Fman: *dumbo.sgmii_s3p1* and *dumbo.sgmii_s3p2*. These ports have been connected to the test machine as follows:

```
[b43198@zro04qoriqsrv][~]$ lars list links
zro04qoriq02.p1      <--->     dumbo.sgmii_s3p1
zro04qoriq02.p2      <--->     dumbo.sgmii_s3p2
```

The following software components have been compiled from the SDK:

- fman microcode - fsl_fman_ucode.bin
- rcw - R_PPSXN_0x10/rcw_13g_1500mhz_rev2.bin
- u-boot
- kernel - uImage--3.8-r15.1-p4080ds-20130527105557.bin
- device tree blob - uImage_sdk_1.4_23.05_usdpaa_multiple_1pool.dtb
- rootfs - fsl-image-core-p4080ds-20130527121546.rootfs.ext2.gz.u-boot

The device tree has been slightly modified in order to support multiple running instances of the USDPAA application. This modification will be detailed in subsection Running multiple USDPAA instances on page 2466.

Booting the Board

```
setenv kernelimage; setenv dtbimage; setenv rootfsimage;setenv bootargs console=ttyS0,115200;setenv
tftpram 'tftp 1000000 $kernelimage;tftp c00000 $dtbimage;tftp 2000000 $rootfsimage'; setenv bootram
'bootm 1000000 2000000 c00000'

setenv kernelimage b43198/uImage--3.8-r15.1-p4080ds-20130527105557.bin; setenv dtbimage b43198/
uImage_sdk_1.4_23.05_usdpaa_multiple_1pool.dtb; setenv rootfsimage b43198/fsl-image-core-
p4080ds-20130527121546.rootfs.ext2.gz.u-boot

setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M bportals=s0-1 qportals=s0-1

run tftpram; run bootram
```

Running a Single Reflector Process

According to the documentation, it takes 2 steps:

- running the Frame Manager Configuration tool (**fmc**) - NOTE: documentation states that this tool can be used only once
- running the reflector process

Both applications require 2 files:

- FMC Policy Definition File - PCD
- FMC Configuration Source File

These files are passed as arguments and must be the same for the two applications. The policy definition file can be the same regardless of SerDes configuration. The configuration source file is SerDes config dependent. Since the **dumbo** board exposes 2 SGMII ports, it has been booted with SerDes config 0x10 and the config file **usdpaa_config_p4_serdes_0x10.xml** has been modified as follows:

```
<cfgdata>
 <config>
  <engine name="fm1">
   <port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
   <port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
```

```
  </engine>
 </config>
</cfgdata>
```

This translates into calling **fmc** and **reflector** for interfaces **fm2-gb0** and **fm2-gb1**.

```
root@p4080ds:~# fmc -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -a
root@p4080ds:~# reflector -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -b
120:0:0 0..7
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@5, Tx Channel = 61, FMAN = 1, Port ID = 0
Found /fsl,dpaa/ethernet@6, Tx Channel = 62, FMAN = 1, Port ID = 1
Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0
WARN:Can't find Qman clock frequency
Configuring for 2 network interfaces
Allocated DMA region size 0x1000000
reflector starting
Released 120 bufs to BPID 8
Released 120 bufs to BPID 9
Thread uid:0 alive (on cpu 0)
Thread uid:1 alive (on cpu 1)
Thread uid:2 alive (on cpu 2)
Thread uid:3 alive (on cpu 3)
Thread uid:4 alive (on cpu 4)
Thread uid:5 alive (on cpu 5)
Thread uid:6 alive (on cpu 6)
Thread uid:7 alive (on cpu 7)
reflector>
```

Testing

Testing has been done using the steps indicated in he documentation. The **zro04qoriq04** test workstation has been configured as follows:

- IP addresses:

```
ifconfig eth3 192.168.10.2 up
ifconfig eth4 192.168.20.2 up
```

- ARP entries:

```
arp -s 192.168.10.1 00:E0:0C:00:93:05
arp -s 192.168.20.1 00:E0:0C:00:93:06
```

Test commands:

```
[b43198@zro04qoriq04 ~]$ ping -c 3 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_seq=1 ttl=64 time=0.227 ms
64 bytes from 192.168.10.1: icmp_seq=2 ttl=64 time=0.216 ms
64 bytes from 192.168.10.1: icmp_seq=3 ttl=64 time=0.338 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2034ms
rtt min/avg/max/mdev = 0.216/0.260/0.338/0.056 ms

[b43198@zro04qoriq04 ~]$ ping -c 3 192.168.20.1
PING 192.168.20.1 (192.168.20.1) 56(84) bytes of data.
64 bytes from 192.168.20.1: icmp_seq=1 ttl=64 time=0.298 ms
64 bytes from 192.168.20.1: icmp_seq=2 ttl=64 time=0.204 ms
```

```
64 bytes from 192.168.20.1: icmp_seq=3 ttl=64 time=0.229 ms

--- 192.168.20.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2032ms
rtt min/avg/max/mdev = 0.204/0.243/0.298/0.043 ms
```

11.4.6.2 Running multiple USDPAA instances

According to reference "Running multiple distinct PPAM application processes (or "instances" as they are sometimes described) requires that each process have its own dedicated set of FMAN interfaces, buffer pools, and cores".

The following resources need to be partitioned:

- interfaces - by using an additional **-i** parameter to the reflector application. Since Since there are only 2 FMan interfaces available on the board - **fm2-gb0** and **fm2-gb1** - the maximum number of USDPAA instances that can be run is 2.
- cores - by using an additional **cpu-range** parameter to the reflector application, in the form of '**index**' or '**first .. last**'. The reflector will start threads affine to the mentioned cores.
- buffer pools - this is more delicate and will be discussed in detail in the next subsection.

Partitioning Buffer Pools

Buffer pools are associated to Ethernet interfaces and the mappings are done in the device tree blob.

There should be distinct buffer pools for distinct USDPAA processes. When sharing buffer pools among different running processes, they fail with SEGFAULT.

Documentation states that "*Each PPAC-based application process will initialize the (usually 3) buffer pools used by each FMan interface that belongs to it. (If a buffer pool is used by more than one interface, it will only be initialised once.) By default the number of buffers to allocate for the triplet of pools used by an FMan interface is 0 for the first two pools and 1728 for the third. The default allocation triplet can be overridden via the -b option.*"

One can define 1, 2 or 3 buffer pools per FMan interface. When doing custom seeding via the -b parameter - which only accepts a triplet x:y:z - the first value (x) corresponds to the first bufer, the second (y) corresponds to the second buffer and the third value (z) to the third buffer. If an interface only has 1 associated buffer pool, passing *-b 120:0:0* is valid, as it will seed only the 1st buffer pool - the interface's only one; if it has 2 buffer pools - *-b 120:120:0* is also valid. You may also pass other values than 0 in the triplets - since they don't have any buffer pool correspondence, they will be discarded. One must seed the buffer pool with respect to the DMA memory size allocated for USDPAA and passed via the **usdpaa_mem** boot argument.

The following is a snippet from the device tree containing the buffer pools for interfaces **fm2-gb0** (*ethernet@5*) and **fm2-gb1** (*ethernet@6*).

```
[ ... ]

2167     ethernet@5 {
2168       compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
2169       fsl,fman-mac = <0x59>;
2170       fsl,bman-buffer-pools = <0x53>;
2171       fsl,qman-frame-queues-rx = <0x5c 0x1 0x5d 0x1>;
2172       fsl,qman-frame-queues-tx = <0x7c 0x1 0x7d 0x1>;
2173     };
2174
2175     ethernet@6 {
2176       compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
2177       fsl,fman-mac = <0x5a>;
2178       fsl,bman-buffer-pools = <0x54>;
2179       fsl,qman-frame-queues-rx = <0x5e 0x1 0x5f 0x1>;
2180       fsl,qman-frame-queues-tx = <0x7e 0x1 0x7f 0x1>;
2181     };
```

```
[ ... ]

2214   buffer-pool@7 {
2215     compatible = "fsl,p4080-bpool", "fsl,bpool";
2216     fsl,bpid = <0x7>;
2217     fsl,bpool-ethernet-cfg = <0x0 0x0 0x0 0xc0 0x0 0xdeadbeef>;
2218     fsl,bpool-thresholds = <0x400 0xc00 0x0 0x0>;
2219     linux,phandle = <0x52>;
2220     phandle = <0x52>;
2221   };
2222
2223   buffer-pool@8 {
2224     compatible = "fsl,p4080-bpool", "fsl,bpool";
2225     fsl,bpid = <0x8>;
2226     fsl,bpool-ethernet-cfg = <0x0 0x0 0x0 0x240 0x0 0xabbaf00d>;
2227     fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
2228     linux,phandle = <0x53>;
2229     phandle = <0x53>;
2230   };
2231
2232   buffer-pool@9 {
2233     compatible = "fsl,p4080-bpool", "fsl,bpool";
2234     fsl,bpid = <0x9>;
2235     fsl,bpool-ethernet-cfg = <0x0 0x0 0x0 0x6c0 0x0 0xfeedabba>;
2236     fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
2237     linux,phandle = <0x54>;
2238     phandle = <0x54>;
2239   };

[ ... ]
```

Running 2 Reflectors

To run 2 instances of the reflector application, one must:

- run the fmc tool - only once
- run 2 reflector processes

The fmc tool is passed an .xml config file containing both FMan interfaces present on the board. In order to provide each reflector instance with a different interface, you can either:

- pass a .xml config file describing only that particular interface;
- pass a .xml config file with both interfaces, but also pass an additional **-i** parameter, specifying the designated one.

The differential config files may look as follows:

- **usdpaa_config_p4_serdes_0x10_0.xml** - only fm2-gb0 is present:

```
<cfgdata>
 <config>
  <engine name="fm1">
   <port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
  </engine>
 </config>
</cfgdata>
```

- **usdpaa_config_p4_serdes_0x10_1.xml** - only fm2-gb1 is present:

```
<cfgdata>
 <config>
  <engine name="fm1">
```

```
    <port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
   </engine>
  </config>
 </cfgdata>
```

In order to run 2 reflectors, you can issue the following commands - one in each terminal:

```
reflector -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -b 120:0:0 -i fm2-gb0
0..1
reflector -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -b 120:0:0 -i fm2-gb1
4..5
```

... or these:

```
reflector -c usdpaa_config_p4_serdes_0x10_0.xml -p usdpaa_policy_hash_ipv4.xml -b 120:0:0 0..1
reflector -c usdpaa_config_p4_serdes_0x10_1.xml -p usdpaa_policy_hash_ipv4.xml -b 120:0:0 4..5
```

11.4.6.3 Running Reflector in Containers

Inspecting the Reflector App

The first step in investigating reflector behavior was to **strace** it. Since strace exposes application syscalls, and containers are running with the same kernel as the host, this should be an appropriate approach.

Looking at the results, the reflector opens the following files:

- /lib/* - library files;
- /proc/device-tree/* - the entire board device tree (.dtb);
- the passed .xml config files;
- /dev/mem - access portal memory mappings;
- /dev/fsl-usdpaa - manage the kernel DPAA resource allocator;
- /dev/fsl-usdpaa-irq - interrupt controller device (replacement for /dev/uio* entries);
- /etc/terminfo/* - terminal settings.

In order to run the test, all these files must be present in the container filesystem. Each container filesystem is configured to contain the /dev/mem and /dev/fsl-usdpaa devices - these are shared. Since SDK 1.4, the /dev/bman-uio and /dev/qman-uio entries have been replaced with a common **/dev/fsl-usdpaa-irq** device used by the processes to register to receiving the interrupts. This device is shared among the containers as well. These devices will then be created with mknod commands inside each container.

Running the test

NOTE: commands are shown for one container only; commands for the other container are issued symmetrically.

1. mount the croups filesystem on host - requirement of LXC (see Host Root Filesystem Configuration for Linux Containers on page 2436).
2. run the fmc tool on host - the fmc tool is used to initialize the FMan kernel driver and should be run only once after each successful boot, regardless of how many USDPAA application instances are started/stopped afterwards. It has no particular importance whether this is run on the host or inside a container, since its purpose is to configure kernel space variables - and these are shared between the host and the containers. Considering this aspect, and the fact that fmc should be run only once, it is best for it to be run on the host, before starting any USDPAA aplication:

   ```
   # fmc -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -a
   ```

3. edit the LXC container config file - **lxc-usdpaa-1.conf** - doing the following:
   - set container name

- enable device rights
- set dedicated cores

```
lxc.utsname = usdpaa1

# You may add some other parameters here if you like

# First deny all device access , then allow specific ones
lxc.cgroup.devices.deny = a

# Enable console and terminal devices
lxc.cgroup.devices.allow = c 5:* rwm # console
lxc.cgroup.devices.allow = c 4:* rwm # tty0
lxc.cgroup.devices.allow = c 136:* rwm # tty1
# Enable other devices access
lxc.cgroup.devices.allow = b 1:0 rwm # ram0
lxc.cgroup.devices.allow = c 1:3 rwm # null
lxc.cgroup.devices.allow = c 1:9 rwm # urandom
# Enable usdpaa
lxc.cgroup.devices.allow = c 10:61 rwm # fsl-usdpaa-irq
lxc.cgroup.devices.allow = c 10:62 rwm # fsl-usdpaa
lxc.cgroup.devices.allow = c 1:1 rwm # mem

# Dedicated cores
lxc.cgroup.cpuset.cpus = 0,1,2,3
```

4. create and start the container:

```
root@p4080ds:~# lxc-create -n foo1 -t busybox -f lxc-usdpaa-1.conf
setting root password to "root"
Password for 'root' changed
'busybox' template installed
'foo1' created
root@p4080ds:~# lxc-start -n foo1 -d
root@p4080ds:~# lxc-console -n foo1

Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

foo1 login: root
Password: root
~ #
```

5. copy binaries and xml files from host:

```
# reflector binary
root@p4080ds:~# mkdir -p /var/lib/lxc/foo1/rootfs/usr/bin
root@p4080ds:~# cp /usr/bin/reflector /var/lib/lxc/foo1/rootfs/usr/bin/
# usdpaa local xmls
root@p4080ds:~# cp usdpaa_* /var/lib/lxc/foo1/rootfs/root/
# terminfo settings
root@p4080ds:~# mkdir -p /var/lib/lxc/foo1/rootfs/etc/terminfo
root@p4080ds:~# cp -R /etc/terminfo/* /var/lib/lxc/foo1/rootfs/etc/terminfo/
```

6. setup the container - create devices in /dev:

```
~ # mknod /dev/fsl-usdpaa-irq c 10 61
~ # mknod /dev/fsl-usdpaa c 10 62
~ # mknod /dev/mem c 1 1
```

7. run reflector in container:

```
~ # reflector -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -b 120:0:0 -i
fm2-gb0 0..3
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@5, Tx Channel = 61, FMAN = 1, Port ID = 0
Found /fsl,dpaa/ethernet@6, Tx Channel = 62, FMAN = 1, Port ID = 1
Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0
WARN:Can't find Qman clock frequency
Configuring for 1 network interface
Allocated DMA region size 0x1000000
reflector starting
Released 120 bufs to BPID 8
Thread uid:0 alive (on cpu 0)
Thread uid:1 alive (on cpu 1)
Thread uid:2 alive (on cpu 2)
Thread uid:3 alive (on cpu 3)
reflector>
```

8. test from Linux PC:

```
[b43198@zro04qoriq04 ~]$ ping -c 3 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_seq=1 ttl=64 time=0.177 ms
64 bytes from 192.168.10.1: icmp_seq=2 ttl=64 time=0.217 ms
64 bytes from 192.168.10.1: icmp_seq=3 ttl=64 time=0.321 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2037ms
rtt min/avg/max/mdev = 0.177/0.238/0.321/0.061 ms
```

By following the same steps for the second container - with slight modications regarding the container name and assigned resources - another reflector instance can be successfully started and run in parallel with the first one, for the second FMan interface **fm2-gb1**.

11.4.7 Appendix

11.4.7.1 LXC Configuration File Reference

The text below comes from the lxc.conf.5 man page:

```
LXC.CONF(5)                                                        LXC.CONF(5)
NAME

       lxc.conf - Configuration files for LXC.
DESCRIPTION

       LXC configuration is split in two parts. Container configuration and
       system configuration.

   CONTAINER CONFIGURATION
       The container configuration is held in the config stored in the
       container's directory.

       A basic configuration is generated at container creation time with
       the default's recommended for the chosen template as well as extra
       default keys coming from the default.conf file.
```

```
       That default.conf file is either located at
       /usr/local/etc/lxc/default.conf or for unprivileged containers at
       ~/.config/lxc/default.conf.

       Details about the syntax of this file can be found in:
       lxc.container.conf(5)

   SYSTEM CONFIGURATION
       The system configuration is located at /usr/local/etc/lxc/lxc.conf or
       ~/.config/lxc/lxc.conf for unprivileged containers.

       This configuration file is used to set values such as default lookup
       paths and storage backend settings for LXC.

       Details about the syntax of this file can be found in:
       lxc.system.conf(5)
SEE ALSO

       lxc(1), lxc.container.conf(5), lxc.system.conf(5), lxc-usernet(5)
AUTHOR

       Stéphane Graber <stgraber@ubuntu.com>
COLOPHON

       This page is part of the lxc (Linux containers) project.  Information
       about the project can be found at http://linuxcontainers.org/.  If
       you have a bug report for this manual page, send it to
       lxc-devel@lists.linuxcontainers.org.  This page was obtained from the
       project's upstream Git repository (git://github.com/lxc/lxc) on
       2014-05-21.  If you discover any rendering problems in this HTML
       version of the page, or you believe there is a better or more up-to-
       date source for the page, or you have corrections or improvements to
       the information in this COLOPHON (which is not part of the original
       manual page), send a mail to man-pages@man7.org

                         Wed May 21 10:30:15 CEST 2014            LXC.CONF(5)
```

The text below comes from the lxc.container.conf.5 man page:

```
LXC.CONTAINER.CONF(5)                                  LXC.CONTAINER.CONF(5)
NAME

       lxc.container.conf - LXC container configuration file

DESCRIPTION

       The linux containers (lxc) are always created before being used. This
       creation defines a set of system resources to be virtualized /
       isolated when a process is using the container. By default, the pids,
       sysv ipc and mount points are virtualized and isolated. The other
       system resources are shared across containers, until they are
       explicitly defined in the configuration file. For example, if there
       is no network configuration, the network will be shared between the
       creator of the container and the container itself, but if the network
       is specified, a new network stack is created for the container and
       the container can no longer use the network of its ancestor.

       The configuration file defines the different system resources to be
       assigned for the container. At present, the utsname, the network, the
       mount points, the root file system, the user namespace, and the
```

```
       control groups are supported.

       Each option in the configuration file has the form key = value
       fitting in one line. The '#' character means the line is a comment.

   CONFIGURATION
       In order to ease administration of multiple related containers, it is
       possible to have a container configuration file cause another file to
       be loaded. For instance, network configuration can be defined in one
       common file which is included by multiple containers. Then, if the
       containers are moved to another host, only one file may need to be
       updated.

       lxc.include
              Specify the file to be included. The included file must be in
              the same valid lxc configuration file format.

   ARCHITECTURE
       Allows one to set the architecture for the container. For example,
       set a 32bits architecture for a container running 32bits binaries on
       a 64bits host. This fixes the container scripts which rely on the
       architecture to do some work like downloading the packages.

       lxc.arch
              Specify the architecture for the container.

              Valid options are x86, i686, x86_64, amd64

   HOSTNAME
       The utsname section defines the hostname to be set for the container.
       That means the container can set its own hostname without changing
       the one from the system. That makes the hostname private for the
       container.

       lxc.utsname
              specify the hostname for the container

   HALT SIGNAL
       Allows one to specify signal name or number, sent by lxc-stop to the
       container's init process to cleanly shutdown the container. Different
       init systems could use different signals to perform clean shutdown
       sequence. This option allows the signal to be specified in kill(1)
       fashion, e.g. SIGPWR, SIGRTMIN+14, SIGRTMAX-10 or plain number. The
       default signal is SIGPWR.

       lxc.haltsignal
              specify the signal used to halt the container

   STOP SIGNAL
       Allows one to specify signal name or number, sent by lxc-stop to
       forcibly shutdown the container. This option allows signal to be
       specified in kill(1) fashion, e.g. SIGKILL, SIGRTMIN+14, SIGRTMAX-10
       or plain number.  The default signal is SIGKILL.

       lxc.stopsignal
              specify the signal used to stop the container

   NETWORK
       The network section defines how the network is virtualized in the
       container. The network virtualization acts at layer two. In order to
```

```
       use the network virtualization, parameters must be specified to
       define the network interfaces of the container. Several virtual
       interfaces can be assigned and used in a container even if the system
       has only one physical network interface.

       lxc.network.type
              specify what kind of network virtualization to be used for the
              container. Each time a lxc.network.type field is found a new
              round of network configuration begins. In this way, several
              network virtualization types can be specified for the same
              container, as well as assigning several network interfaces for
              one container. The different virtualization types can be:

              none: will cause the container to share the host's network
              namespace. This means the host network devices are usable in
              the container. It also means that if both the container and
              host have upstart as init, 'halt' in a container (for
              instance) will shut down the host.

              empty: will create only the loopback interface.

              veth: a peer network device is created with one side assigned
              to the container and the other side is attached to a bridge
              specified by the lxc.network.link. If the bridge is not
              specified, then the veth pair device will be created but not
              attached to any bridge. Otherwise, the bridge has to be setup
              before on the system, lxc won't handle any configuration
              outside of the container. By default lxc choose a name for the
              network device belonging to the outside of the container, this
              name is handled by lxc, but if you wish to handle this name
              yourself, you can tell lxc to set a specific name with the
              lxc.network.veth.pair option.

              vlan: a vlan interface is linked with the interface specified
              by the lxc.network.link and assigned to the container. The
              vlan identifier is specified with the option
              lxc.network.vlan.id.

              macvlan: a macvlan interface is linked with the interface
              specified by the lxc.network.link and assigned to the
              container.  lxc.network.macvlan.mode specifies the mode the
              macvlan will use to communicate between different macvlan on
              the same upper device. The accepted modes are private, the
              device never communicates with any other device on the same
              upper_dev (default), vepa, the new Virtual Ethernet Port
              Aggregator (VEPA) mode, it assumes that the adjacent bridge
              returns all frames where both source and destination are local
              to the macvlan port, i.e. the bridge is set up as a reflective
              relay. Broadcast frames coming in from the upper_dev get
              flooded to all macvlan interfaces in VEPA mode, local frames
              are not delivered locally, or bridge, it provides the behavior
              of a simple bridge between different macvlan interfaces on the
              same port. Frames from one interface to another one get
              delivered directly and are not sent out externally. Broadcast
              frames get flooded to all other bridge ports and to the
              external interface, but when they come back from a reflective
              relay, we don't deliver them again. Since we know all the MAC
              addresses, the macvlan bridge mode does not require learning
              or STP like the bridge module does.
```

```
              phys: an already existing interface specified by the
              lxc.network.link is assigned to the container.

       lxc.network.flags
              specify an action to do for the network.

              up: activates the interface.

       lxc.network.link
              specify the interface to be used for real network traffic.

       lxc.network.mtu
              specify the maximum transfer unit for this interface.

       lxc.network.name
              the interface name is dynamically allocated, but if another
              name is needed because the configuration files being used by
              the container use a generic name, eg. eth0, this option will
              rename the interface in the container.

       lxc.network.hwaddr
              the interface mac address is dynamically allocated by default
              to the virtual interface, but in some cases, this is needed to
              resolve a mac address conflict or to always have the same
              link-local ipv6 address.  Any "x" in address will be replaced
              by random value, this allows setting hwaddr templates.

       lxc.network.ipv4
              specify the ipv4 address to assign to the virtualized
              interface. Several lines specify several ipv4 addresses.  The
              address is in format x.y.z.t/m, eg. 192.168.1.123/24. The
              broadcast address should be specified on the same line, right
              after the ipv4 address.

       lxc.network.ipv4.gateway
              specify the ipv4 address to use as the gateway inside the
              container. The address is in format x.y.z.t, eg.
              192.168.1.123.  Can also have the special value auto, which
              means to take the primary address from the bridge interface
              (as specified by the lxc.network.link option) and use that as
              the gateway. auto is only available when using the veth and
              macvlan network types.

       lxc.network.ipv6
              specify the ipv6 address to assign to the virtualized
              interface. Several lines specify several ipv6 addresses.  The
              address is in format x::y/m, eg.
              2003:db8:1:0:214:1234:fe0b:3596/64

       lxc.network.ipv6.gateway
              specify the ipv6 address to use as the gateway inside the
              container. The address is in format x::y, eg. 2003:db8:1:0::1
              Can also have the special value auto, which means to take the
              primary address from the bridge interface (as specified by the
              lxc.network.link option) and use that as the gateway. auto is
              only available when using the veth and macvlan network types.

       lxc.network.script.up
              add a configuration option to specify a script to be executed
              after creating and configuring the network used from the host
```

```
              side. The following arguments are passed to the script:
              container name and config section name (net) Additional
              arguments depend on the config section employing a script
              hook; the following are used by the network system: execution
              context (up), network type (empty/veth/macvlan/phys),
              Depending on the network type, other arguments may be passed:
              veth/macvlan/phys. And finally (host-sided) device name.

              Standard output from the script is logged at debug level.
              Standard error is not logged, but can be captured by the hook
              redirecting its standard error to standard output.

      lxc.network.script.down
              add a configuration option to specify a script to be executed
              before destroying the network used from the host side. The
              following arguments are passed to the script: container name
              and config section name (net) Additional arguments depend on
              the config section employing a script hook; the following are
              used by the network system: execution context (down), network
              type (empty/veth/macvlan/phys), Depending on the network type,
              other arguments may be passed: veth/macvlan/phys. And finally
              (host-sided) device name.

              Standard output from the script is logged at debug level.
              Standard error is not logged, but can be captured by the hook
              redirecting its standard error to standard output.

  NEW PSEUDO TTY INSTANCE (DEVPTS)
      For stricter isolation the container can have its own private
      instance of the pseudo tty.

      lxc.pts
              If set, the container will have a new pseudo tty instance,
              making this private to it. The value specifies the maximum
              number of pseudo ttys allowed for a pts instance (this
              limitation is not implemented yet).

  CONTAINER SYSTEM CONSOLE
      If the container is configured with a root filesystem and the inittab
      file is setup to use the console, you may want to specify where the
      output of this console goes.

      lxc.console
              Specify a path to a file where the console output will be
              written. The keyword 'none' will simply disable the console.
              This is dangerous once if have a rootfs with a console device
              file where the application can write, the messages will fall
              in the host.

  CONSOLE THROUGH THE TTYS
      This option is useful if the container is configured with a root
      filesystem and the inittab file is setup to launch a getty on the
      ttys. The option specifies the number of ttys to be available for the
      container. The number of gettys in the inittab file of the container
      should not be greater than the number of ttys specified in this
      option, otherwise the excess getty sessions will die and respawn
      indefinitely giving annoying messages on the console or in
      /var/log/messages.

      lxc.tty
```

```
            Specify the number of tty to make available to the container.

CONSOLE DEVICES LOCATION
    LXC consoles are provided through Unix98 PTYs created on the host and
    bind-mounted over the expected devices in the container.  By default,
    they are bind-mounted over /dev/console and /dev/ttyN. This can
    prevent package upgrades in the guest. Therefore you can specify a
    directory location (under /dev under which LXC will create the files
    and bind-mount over them. These will then be symbolically linked to
    /dev/console and /dev/ttyN.  A package upgrade can then succeed as it
    is able to remove and replace the symbolic links.

    lxc.devttydir
           Specify a directory under /dev under which to create the
           container console devices.

/DEV DIRECTORY
    By default, lxc creates a few symbolic links (fd,stdin,stdout,stderr)
    in the container's /dev directory but does not automatically create
    device node entries. This allows the container's /dev to be set up as
    needed in the container rootfs. If lxc.autodev is set to 1, then
    after mounting the container's rootfs LXC will mount a fresh tmpfs
    under /dev (limited to 100k) and fill in a minimal set of initial
    devices.  This is generally required when starting a container
    containing a "systemd" based "init" but may be optional at other
    times. Additional devices in the containers /dev directory may be
    created through the use of the lxc.hook.autodev hook.

    lxc.autodev
           Set this to 1 to have LXC mount and populate a minimal /dev
           when starting the container.

ENABLE KMSG SYMLINK
    Enable creating /dev/kmsg as symlink to /dev/console. This defaults
    to 1.

    lxc.kmsg
           Set this to 0 to disable /dev/kmsg symlinking.

MOUNT POINTS
    The mount points section specifies the different places to be
    mounted. These mount points will be private to the container and
    won't be visible by the processes running outside of the container.
    This is useful to mount /etc, /var or /home for examples.

    lxc.mount
           specify a file location in the fstab format, containing the
           mount information. The mount target location can and in most
           cases should be a relative path, which will become relative to
           the mounted container root. For instance,

           proc proc proc nodev,noexec,nosuid 0 0

           Will mount a proc filesystem under the container's /proc,
           regardless of where the root filesystem comes from. This is
           resilient to block device backed filesystems as well as
           container cloning.

           Note that when mounting a filesystem from an image file or
           block device the third field (fs_vfstype) cannot be auto as
```

```
               with mount(8) but must be explicitly specified.

       lxc.mount.entry
               specify a mount point corresponding to a line in the fstab
               format.

       lxc.mount.auto
               specify which standard kernel file systems should be
               automatically mounted. This may dramatically simplify the
               configuration. The file systems are:

               · proc:mixed (or proc): mount /proc as read-write, but remount
                 /proc/sys and /proc/sysrq-trigger read-only for security /
                 container isolation purposes.

               · proc:rw: mount /proc as read-write

               · sys:ro (or sys): mount /sys as read-only for security /
                 container isolation purposes.

               · sys:rw: mount /sys as read-write

               · cgroup:mixed: mount a tmpfs to /sys/fs/cgroup, create
                 directories for all hierarchies to which the container is
                 added, create subdirectories there with the name of the
                 cgroup, and bind-mount the container's own cgroup into that
                 directory.  The container will be able to write to its own
                 cgroup directory, but not the parents, since they will be
                 remounted read-only

               · cgroup:ro: similar to cgroup:mixed, but everything will be
                 mounted read-only.

               · cgroup:rw: similar to cgroup:mixed, but everything will be
                 mounted read-write. Note that the paths leading up to the
                 container's own cgroup will be writable, but will not be a
                 cgroup filesystem but just part of the tmpfs of
                 /sys/fs/cgroup

               · cgroup (without specifier): defaults to cgroup:rw if the
                 container retains the CAP_SYS_ADMIN capability, cgroup:mixed
                 otherwise.

               · cgroup-full:mixed: mount a tmpfs to /sys/fs/cgroup, create
                 directories for all hierarchies to which the container is
                 added, bind-mount the hierarchies from the host to the
                 container and make everything read-only except the
                 container's own cgroup. Note that compared to cgroup, where
                 all paths leading up to the container's own cgroup are just
                 simple directories in the underlying tmpfs, here
                 /sys/fs/cgroup/$hierarchy will contain the host's full
                 cgroup hierarchy, albeit read-only outside the container's
                 own cgroup.  This may leak quite a bit of information into
                 the container.

               · cgroup-full:ro: similar to cgroup-full:mixed, but everything
                 will be mounted read-only.

               · cgroup-full:rw: similar to cgroup-full:mixed, but everything
                 will be mounted read-write. Note that in this case, the
```

```
                container may escape its own cgroup. (Note also that if the
                container has CAP_SYS_ADMIN support and can mount the cgroup
                filesystem itself, it may do so anyway.)

              · cgroup-full (without specifier): defaults to cgroup-full:rw
                if the container retains the CAP_SYS_ADMIN capability,
                cgroup-full:mixed otherwise.

       Note that if automatic mounting of the cgroup filesystem is enabled,
       the tmpfs under /sys/fs/cgroup will always be mounted read-write (but
       for the :mixed and :ro cases, the individual hierarchies,
       /sys/fs/cgroup/$hierarchy, will be read-only). This is in order to
       work around a quirk in Ubuntu's mountall(8) command that will cause
       containers to wait for user input at boot if /sys/fs/cgroup is
       mounted read-only and the container can't remount it read-write due
       to a lack of CAP_SYS_ADMIN.

       Examples:

                  lxc.mount.auto = proc sys cgroup
                  lxc.mount.auto = proc:rw sys:rw cgroup-full:rw

   ROOT FILE SYSTEM
       The root file system of the container can be different than that of
       the host system.

       lxc.rootfs
              specify the root file system for the container. It can be an
              image file, a directory or a block device. If not specified,
              the container shares its root file system with the host.

              For directory or simple block-device backed containers, a
              pathname can be used. If the rootfs is backed by a nbd device,
              then nbd:file:1 specifies that file should be attached to a
              nbd device, and partition 1 should be mounted as the rootfs.
              nbd:file specifies that the nbd device itself should be
              mounted. overlayfs:/lower:/upper specifies that the rootfs
              should be an overlay with /upper being mounted read-write over
              a read-only mount of /lower.  aufs:/lower:/upper does the same
              using aufs in place of overlayfs. loop:/file tells lxc to
              attach /file to a loop device and mount the loop device.

       lxc.rootfs.mount
              where to recursively bind lxc.rootfs before pivoting. This is
              to ensure success of the pivot_root(8) syscall. Any directory
              suffices, the default should generally work.

       lxc.rootfs.options
              extra mount options to use when mounting the rootfs.

       lxc.pivotdir
              where to pivot the original root file system under lxc.rootfs,
              specified relatively to that. The default is mnt.  It is
              created if necessary, and also removed after unmounting
              everything from it during container setup.

   CONTROL GROUP
       The control group section contains the configuration for the
       different subsystem. lxc does not check the correctness of the
       subsystem name. This has the disadvantage of not detecting
```

```
       configuration errors until the container is started, but has the
       advantage of permitting any future subsystem.

       lxc.cgroup.[subsystem name]
              specify the control group value to be set. The subsystem name
              is the literal name of the control group subsystem. The
              permitted names and the syntax of their values is not dictated
              by LXC, instead it depends on the features of the Linux kernel
              running at the time the container is started, eg.
              lxc.cgroup.cpuset.cpus

   CAPABILITIES
       The capabilities can be dropped in the container if this one is run
       as root.

       lxc.cap.drop
              Specify the capability to be dropped in the container. A
              single line defining several capabilities with a space
              separation is allowed. The format is the lower case of the
              capability definition without the "CAP_" prefix, eg.
              CAP_SYS_MODULE should be specified as sys_module. See
              capabilities(7),

       lxc.cap.keep
              Specify the capability to be kept in the container. All other
              capabilities will be dropped.

   APPARMOR PROFILE
       If lxc was compiled and installed with apparmor support, and the host
       system has apparmor enabled, then the apparmor profile under which
       the container should be run can be specified in the container
       configuration. The default is lxc-container-default.

       lxc.aa_profile
              Specify the apparmor profile under which the container should
              be run. To specify that the container should be unconfined,
              use

              lxc.aa_profile = unconfined

   SELINUX CONTEXT
       If lxc was compiled and installed with SELinux support, and the host
       system has SELinux enabled, then the SELinux context under which the
       container should be run can be specified in the container
       configuration. The default is unconfined_t, which means that lxc will
       not attempt to change contexts.

       lxc.se_context
              Specify the SELinux context under which the container should
              be run or unconfined_t. For example

              lxc.se_context = unconfined_u:unconfined_r:lxc_t:s0-s0:c0.c1023

   SECCOMP CONFIGURATION
       A container can be started with a reduced set of available system
       calls by loading a seccomp profile at startup. The seccomp
       configuration file must begin with a version number on the first
       line, a policy type on the second line, followed by the
       configuration.
```

```
       Versions 1 and 2 are currently supported. In version 1, the policy is
       a simple whitelist. The second line therefore must read "whitelist",
       with the rest of the file containing one (numeric) sycall number per
       line. Each syscall number is whitelisted, while every unlisted number
       is blacklisted for use in the container

       In version 2, the policy may be blacklist or whitelist, supports per-
       rule and per-policy default actions, and supports per-architecture
       system call resolution from textual names.

       An example blacklist policy, in which all system calls are allowed
       except for mknod, which will simply do nothing and return 0
       (success), looks like:

       2
       blacklist
       mknod errno 0

       lxc.seccomp
              Specify a file containing the seccomp configuration to load
              before the container starts.

   UID MAPPINGS
       A container can be started in a private user namespace with user and
       group id mappings. For instance, you can map userid 0 in the
       container to userid 200000 on the host. The root user in the
       container will be privileged in the container, but unprivileged on
       the host. Normally a system container will want a range of ids, so
       you would map, for instance, user and group ids 0 through 20,000 in
       the container to the ids 200,000 through 220,000.

       lxc.id_map
              Four values must be provided. First a character, either 'u',
              or 'g', to specify whether user or group ids are being mapped.
              Next is the first userid as seen in the user namespace of the
              container. Next is the userid as seen on the host. Finally, a
              range indicating the number of consecutive ids to map.

   CONTAINER HOOKS
       Container hooks are programs or scripts which can be executed at
       various times in a container's lifetime.

       When a container hook is executed, information is passed both as
       command line arguments and through environment variables.  The
       arguments are:

       · Container name.

       · Section (always 'lxc').

       · The hook type (i.e. 'clone' or 'pre-mount').

       · Additional arguments In the case of the clone hook, any extra
         arguments passed to lxc-clone will appear as further arguments to
         the hook.

       The following environment variables are set:

       · LXC_NAME: is the container's name.
```

```
       · LXC_ROOTFS_MOUNT: the path to the mounted root filesystem.

       · LXC_CONFIG_FILE: the path to the container configuration file.

       · LXC_SRC_NAME: in the case of the clone hook, this is the original
         container's name.

       · LXC_ROOTFS_PATH: this is the lxc.rootfs entry for the container.
         Note this is likely not where the mounted rootfs is to be found,
         use LXC_ROOTFS_MOUNT for that.

      Standard output from the hooks is logged at debug level.  Standard
      error is not logged, but can be captured by the hook redirecting its
      standard error to standard output.

      lxc.hook.pre-start
             A hook to be run in the host's namespace before the container
             ttys, consoles, or mounts are up.

      lxc.hook.pre-mount
             A hook to be run in the container's fs namespace but before
             the rootfs has been set up. This allows for manipulation of
             the rootfs, i.e. to mount an encrypted filesystem. Mounts done
             in this hook will not be reflected on the host (apart from
             mounts propagation), so they will be automatically cleaned up
             when the container shuts down.

      lxc.hook.mount
             A hook to be run in the container's namespace after mounting
             has been done, but before the pivot_root.

      lxc.hook.autodev
             A hook to be run in the container's namespace after mounting
             has been done and after any mount hooks have run, but before
             the pivot_root, if lxc.autodev == 1.  The purpose of this hook
             is to assist in populating the /dev directory of the container
             when using the autodev option for systemd based containers.
             The container's /dev directory is relative to the
             ${LXC_ROOTFS_MOUNT} environment variable available when the
             hook is run.

      lxc.hook.start
             A hook to be run in the container's namespace immediately
             before executing the container's init. This requires the
             program to be available in the container.

      lxc.hook.post-stop
             A hook to be run in the host's namespace after the container
             has been shut down.

      lxc.hook.clone
             A hook to be run when the container is cloned to a new one.
             See lxc-clone(1) for more information.

  CONTAINER HOOKS ENVIRONMENT VARIABLES
      A number of environment variables are made available to the startup
      hooks to provide configuration information and assist in the
      functioning of the hooks. Not all variables are valid in all
      contexts. In particular, all paths are relative to the host system
      and, as such, not valid during the lxc.hook.start hook.
```

```
    LXC_NAME
           The LXC name of the container. Useful for logging messages in
           common log environments. [-n]

    LXC_CONFIG_FILE
           Host relative path to the container configuration file. This
           gives the container to reference the original, top level,
           configuration file for the container in order to locate any
           additional configuration information not otherwise made
           available. [-f]

    LXC_CONSOLE
           The path to the console output of the container if not NULL.
           [-c] [lxc.console]

    LXC_CONSOLE_LOGPATH
           The path to the console log output of the container if not
           NULL.  [-L]

    LXC_ROOTFS_MOUNT
           The mount location to which the container is initially bound.
           This will be the host relative path to the container rootfs
           for the container instance being started and is where changes
           should be made for that instance.  [lxc.rootfs.mount]

    LXC_ROOTFS_PATH
           The host relative path to the container root which has been
           mounted to the rootfs.mount location.  [lxc.rootfs]

LOGGING
    Logging can be configured on a per-container basis. By default,
    depending upon how the lxc package was compiled, container startup is
    logged only at the ERROR level, and logged to a file named after the
    container (with '.log' appended) either under the container path, or
    under /usr/local/var/log/lxc.

    Both the default log level and the log file can be specified in the
    container configuration file, overriding the default behavior. Note
    that the configuration file entries can in turn be overridden by the
    command line options to lxc-start.

    lxc.loglevel
           The level at which to log. The log level is an integer in the
           range of 0..8 inclusive, where a lower number means more
           verbose debugging. In particular 0 = trace, 1 = debug, 2 =
           info, 3 = notice, 4 = warn, 5 = error, 6 = critical, 7 =
           alert, and 8 = fatal. If unspecified, the level defaults to 5
           (error), so that only errors and above are logged.

           Note that when a script (such as either a hook script or a
           network interface up or down script) is called, the script's
           standard output is logged at level 1, debug.

    lxc.logfile
           The file to which logging info should be written.

AUTOSTART
    The autostart options support marking which containers should be
    auto-started and in what order. These options may be used by LXC
```

```
       tools directly or by external tooling provided by the distributions.

       lxc.start.auto
              Whether the container should be auto-started.  Valid values
              are 0 (off) and 1 (on).

       lxc.start.delay
              How long to wait (in seconds) after the container is started
              before starting the next one.

       lxc.start.order
              An integer used to sort the containers when auto-starting a
              series of containers at once.

       lxc.group
              A multi-value key (can be used multiple times) to put the
              container in a container group. Those groups can then be used
              (amongst other things) to start a series of related
              containers.

EXAMPLES

       In addition to the few examples given below, you will find some other
       examples of configuration file in /usr/local/share/doc/lxc/examples

   NETWORK
       This configuration sets up a container to use a veth pair device with
       one side plugged to a bridge br0 (which has been configured before on
       the system by the administrator). The virtual network device visible
       in the container is renamed to eth0.

            lxc.utsname = myhostname
            lxc.network.type = veth
            lxc.network.flags = up
            lxc.network.link = br0
            lxc.network.name = eth0
            lxc.network.hwaddr = 4a:49:43:49:79:bf
            lxc.network.ipv4 = 10.2.3.5/24 10.2.3.255
            lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597

   UID/GID MAPPING
       This configuration will map both user and group ids in the range
       0-9999 in the container to the ids 100000-109999 on the host.

            lxc.id_map = u 0 100000 10000
            lxc.id_map = g 0 100000 10000

   CONTROL GROUP
       This configuration will setup several control groups for the
       application, cpuset.cpus restricts usage of the defined cpu,
       cpus.share prioritize the control group, devices.allow makes usable
       the specified devices.

            lxc.cgroup.cpuset.cpus = 0,1
            lxc.cgroup.cpu.shares = 1234
            lxc.cgroup.devices.deny = a
            lxc.cgroup.devices.allow = c 1:3 rw
            lxc.cgroup.devices.allow = b 8:0 rw

   COMPLEX CONFIGURATION
```

```
       This example show a complex configuration making a complex network
       stack, using the control groups, setting a new hostname, mounting
       some locations and a changing root file system.

            lxc.utsname = complex
            lxc.network.type = veth
            lxc.network.flags = up
            lxc.network.link = br0
            lxc.network.hwaddr = 4a:49:43:49:79:bf
            lxc.network.ipv4 = 10.2.3.5/24 10.2.3.255
            lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
            lxc.network.ipv6 = 2003:db8:1:0:214:5432:feab:3588
            lxc.network.type = macvlan
            lxc.network.flags = up
            lxc.network.link = eth0
            lxc.network.hwaddr = 4a:49:43:49:79:bd
            lxc.network.ipv4 = 10.2.3.4/24
            lxc.network.ipv4 = 192.168.10.125/24
            lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
            lxc.network.type = phys
            lxc.network.flags = up
            lxc.network.link = dummy0
            lxc.network.hwaddr = 4a:49:43:49:79:ff
            lxc.network.ipv4 = 10.2.3.6/24
            lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
            lxc.cgroup.cpuset.cpus = 0,1
            lxc.cgroup.cpu.shares = 1234
            lxc.cgroup.devices.deny = a
            lxc.cgroup.devices.allow = c 1:3 rw
            lxc.cgroup.devices.allow = b 8:0 rw
            lxc.mount = /etc/fstab.complex
            lxc.mount.entry = /lib /root/myrootfs/lib none ro,bind 0 0
            lxc.rootfs = /mnt/rootfs.complex
            lxc.cap.drop = sys_module mknod setuid net_raw
            lxc.cap.drop = mac_override
SEE ALSO

       chroot(1), pivot_root(8), fstab(5), capabilities(7)

SEE ALSO

       lxc(7), lxc-create(1), lxc-destroy(1), lxc-start(1), lxc-stop(1),
       lxc-execute(1), lxc-console(1), lxc-monitor(1), lxc-wait(1),
       lxc-cgroup(1), lxc-ls(1), lxc-info(1), lxc-freeze(1),
       lxc-unfreeze(1), lxc-attach(1), lxc.conf(5)

AUTHOR

       Daniel Lezcano <daniel.lezcano@free.fr>

COLOPHON

       This page is part of the lxc (Linux containers) project.  Information
       about the project can be found at http://linuxcontainers.org/.  If
       you have a bug report for this manual page, send it to
       lxc-devel@lists.linuxcontainers.org.  This page was obtained from the
       project's upstream Git repository (git://github.com/lxc/lxc) on
       2014-05-21.  If you discover any rendering problems in this HTML
       version of the page, or you believe there is a better or more up-to-
       date source for the page, or you have corrections or improvements to
```

```
       the information in this COLOPHON (which is not part of the original
       manual page), send a mail to man-pages@man7.org

                         Wed May 21 10:30:15 CEST 2014  LXC.CONTAINER.CONF(5)
```

The text below comes from the lxc.system.conf.5 man page:

```
LXC.SYSTEM.CONF(5)                                          LXC.SYSTEM.CONF(5)
NAME

       lxc.system.conf - LXC system configuration file

DESCRIPTION

       The system configuration is located at /usr/local/etc/lxc/lxc.conf or
       ~/.config/lxc/lxc.conf for unprivileged containers.

       This configuration file is used to set values such as default lookup
       paths and storage backend settings for LXC.

   CONFIGURATION PATHS
       lxc.lxcpath
              The location in which all containers are stored.

       lxc.default_config
              The path to the default container configuration.

   CONTROL GROUPS
       lxc.cgroup.use
              Comma separated list of cgroup controllers to setup.

       lxc.cgroup.pattern
              Format string used to generate the cgroup path (e.g. lxc/%n).

   LVM
       lxc.bdev.lvm.vg
              Default LVM volume group name.

       lxc.bdev.lvm.thin_pool
              Default LVM thin pool name.

   ZFS
       lxc.bdev.zfs.root
              Default ZFS root name.
        top

       lxc(1), lxc.container.conf(5), lxc.system.conf(5)

SEE ALSO

       lxc(7), lxc-create(1), lxc-destroy(1), lxc-start(1), lxc-stop(1),
       lxc-execute(1), lxc-console(1), lxc-monitor(1), lxc-wait(1),
       lxc-cgroup(1), lxc-ls(1), lxc-info(1), lxc-freeze(1),
       lxc-unfreeze(1), lxc-attach(1), lxc.conf(5)

AUTHOR

       Stéphane Graber <stgraber@ubuntu.com>

COLOPHON
```

```
       This page is part of the lxc (Linux containers) project.  Information
       about the project can be found at http://linuxcontainers.org/.  If
       you have a bug report for this manual page, send it to
       lxc-devel@lists.linuxcontainers.org.  This page was obtained from the
       project's upstream Git repository (git://github.com/lxc/lxc) on
       2014-05-21.  If you discover any rendering problems in this HTML
       version of the page, or you believe there is a better or more up-to-
       date source for the page, or you have corrections or improvements to
       the information in this COLOPHON (which is not part of the original
       manual page), send a mail to man-pages@man7.org

                         Wed May 21 10:30:15 CEST 2014     LXC.SYSTEM.CONF(5)
```

11.4.7.2 Documentation/cgroups/cgroups.txt

The following documentation is from the Linux kernel (Documentation/cgroups/cgroups.txt)

```
                CGROUPS
                -------

Written by Paul Menage <menage@google.com> based on
Documentation/cgroups/cpusets.txt

Original copyright statements from cpusets.txt:
Portions Copyright (C) 2004 BULL SA.
Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.
Modified by Paul Jackson <pj@sgi.com>
Modified by Christoph Lameter <clameter@sgi.com>

CONTENTS:
=========

1. Control Groups
  1.1 What are cgroups ?
  1.2 Why are cgroups needed ?
  1.3 How are cgroups implemented ?
  1.4 What does notify_on_release do ?
  1.5 What does clone_children do ?
  1.6 How do I use cgroups ?
2. Usage Examples and Syntax
  2.1 Basic Usage
  2.2 Attaching processes
  2.3 Mounting hierarchies by name
3. Kernel API
  3.1 Overview
  3.2 Synchronization
  3.3 Subsystem API
4. Extended attributes usage
5. Questions

1. Control Groups
=================

1.1 What are cgroups ?
----------------------

Control Groups provide a mechanism for aggregating/partitioning sets of
tasks, and all their future children, into hierarchical groups with
```

```
specialized behaviour.

Definitions:

A *cgroup* associates a set of tasks with a set of parameters for one
or more subsystems.

A *subsystem* is a module that makes use of the task grouping
facilities provided by cgroups to treat groups of tasks in
particular ways. A subsystem is typically a "resource controller" that
schedules a resource or applies per-cgroup limits, but it may be
anything that wants to act on a group of processes, e.g. a
virtualization subsystem.

A *hierarchy* is a set of cgroups arranged in a tree, such that
every task in the system is in exactly one of the cgroups in the
hierarchy, and a set of subsystems; each subsystem has system-specific
state attached to each cgroup in the hierarchy.  Each hierarchy has
an instance of the cgroup virtual filesystem associated with it.

At any one time there may be multiple active hierarchies of task
cgroups. Each hierarchy is a partition of all tasks in the system.

User-level code may create and destroy cgroups by name in an
instance of the cgroup virtual file system, specify and query to
which cgroup a task is assigned, and list the task PIDs assigned to
a cgroup. Those creations and assignments only affect the hierarchy
associated with that instance of the cgroup file system.

On their own, the only use for cgroups is for simple job
tracking. The intention is that other subsystems hook into the generic
cgroup support to provide new attributes for cgroups, such as
accounting/limiting the resources which processes in a cgroup can
access. For example, cpusets (see Documentation/cgroups/cpusets.txt) allow
you to associate a set of CPUs and a set of memory nodes with the
tasks in each cgroup.

1.2 Why are cgroups needed ?
----------------------------

There are multiple efforts to provide process aggregations in the
Linux kernel, mainly for resource-tracking purposes. Such efforts
include cpusets, CKRM/ResGroups, UserBeanCounters, and virtual server
namespaces. These all require the basic notion of a
grouping/partitioning of processes, with newly forked processes ending
up in the same group (cgroup) as their parent process.

The kernel cgroup patch provides the minimum essential kernel
mechanisms required to efficiently implement such groups. It has
minimal impact on the system fast paths, and provides hooks for
specific subsystems such as cpusets to provide additional behaviour as
desired.

Multiple hierarchy support is provided to allow for situations where
the division of tasks into cgroups is distinctly different for
different subsystems - having parallel hierarchies allows each
hierarchy to be a natural division of tasks, without having to handle
complex combinations of tasks that would be present if several
unrelated subsystems needed to be forced into the same tree of
cgroups.
```

```
At one extreme, each resource controller or subsystem could be in a
separate hierarchy; at the other extreme, all subsystems
would be attached to the same hierarchy.

As an example of a scenario (originally proposed by vatsa@in.ibm.com)
that can benefit from multiple hierarchies, consider a large
university server with various users - students, professors, system
tasks etc. The resource planning for this server could be along the
following lines:

       CPU :          "Top cpuset"
                       /       \
                CPUSet1         CPUSet2
                   |               |
                (Professors)    (Students)

                In addition (system tasks) are attached to topcpuset (so
                that they can run anywhere) with a limit of 20%

       Memory : Professors (50%), Students (30%), system (20%)

       Disk : Professors (50%), Students (30%), system (20%)

       Network : WWW browsing (20%), Network File System (60%), others (20%)
                                / \
                Professors (15%)  students (5%)

Browsers like Firefox/Lynx go into the WWW network class, while (k)nfsd goes
into the NFS network class.

At the same time Firefox/Lynx will share an appropriate CPU/Memory class
depending on who launched it (prof/student).

With the ability to classify tasks differently for different resources
(by putting those resource subsystems in different hierarchies),
the admin can easily set up a script which receives exec notifications
and depending on who is launching the browser he can

    # echo browser_pid > /sys/fs/cgroup/<restype>/<userclass>/tasks

With only a single hierarchy, he now would potentially have to create
a separate cgroup for every browser launched and associate it with
appropriate network and other resource class.  This may lead to
proliferation of such cgroups.

Also let's say that the administrator would like to give enhanced network
access temporarily to a student's browser (since it is night and the user
wants to do online gaming :))  OR give one of the student's simulation
apps enhanced CPU power.

With ability to write PIDs directly to resource classes, it's just a
matter of:

       # echo pid > /sys/fs/cgroup/network/<new_class>/tasks
       (after some time)
       # echo pid > /sys/fs/cgroup/network/<orig_class>/tasks

Without this ability, the administrator would have to split the cgroup into
multiple separate ones and then associate the new cgroups with the
```

```
new resource classes.



1.3 How are cgroups implemented ?
---------------------------------

Control Groups extends the kernel as follows:

 - Each task in the system has a reference-counted pointer to a
   css_set.

 - A css_set contains a set of reference-counted pointers to
   cgroup_subsys_state objects, one for each cgroup subsystem
   registered in the system. There is no direct link from a task to
   the cgroup of which it's a member in each hierarchy, but this
   can be determined by following pointers through the
   cgroup_subsys_state objects. This is because accessing the
   subsystem state is something that's expected to happen frequently
   and in performance-critical code, whereas operations that require a
   task's actual cgroup assignments (in particular, moving between
   cgroups) are less common. A linked list runs through the cg_list
   field of each task_struct using the css_set, anchored at
   css_set->tasks.

 - A cgroup hierarchy filesystem can be mounted for browsing and
   manipulation from user space.

 - You can list all the tasks (by PID) attached to any cgroup.

The implementation of cgroups requires a few, simple hooks
into the rest of the kernel, none in performance-critical paths:

 - in init/main.c, to initialize the root cgroups and initial
   css_set at system boot.

 - in fork and exit, to attach and detach a task from its css_set.

In addition, a new file system of type "cgroup" may be mounted, to
enable browsing and modifying the cgroups presently known to the
kernel.  When mounting a cgroup hierarchy, you may specify a
comma-separated list of subsystems to mount as the filesystem mount
options.  By default, mounting the cgroup filesystem attempts to
mount a hierarchy containing all registered subsystems.

If an active hierarchy with exactly the same set of subsystems already
exists, it will be reused for the new mount. If no existing hierarchy
matches, and any of the requested subsystems are in use in an existing
hierarchy, the mount will fail with -EBUSY. Otherwise, a new hierarchy
is activated, associated with the requested subsystems.

It's not currently possible to bind a new subsystem to an active
cgroup hierarchy, or to unbind a subsystem from an active cgroup
hierarchy. This may be possible in future, but is fraught with nasty
error-recovery issues.

When a cgroup filesystem is unmounted, if there are any
child cgroups created below the top-level cgroup, that hierarchy
will remain active even though unmounted; if there are no
child cgroups then the hierarchy will be deactivated.
```

```
No new system calls are added for cgroups - all support for
querying and modifying cgroups is via this cgroup file system.

Each task under /proc has an added file named 'cgroup' displaying,
for each active hierarchy, the subsystem names and the cgroup name
as the path relative to the root of the cgroup file system.

Each cgroup is represented by a directory in the cgroup file system
containing the following files describing that cgroup:

 - tasks: list of tasks (by PID) attached to that cgroup.  This list
   is not guaranteed to be sorted.  Writing a thread ID into this file
   moves the thread into this cgroup.
 - cgroup.procs: list of thread group IDs in the cgroup.  This list is
   not guaranteed to be sorted or free of duplicate TGIDs, and userspace
   should sort/uniquify the list if this property is required.
   Writing a thread group ID into this file moves all threads in that
   group into this cgroup.
 - notify_on_release flag: run the release agent on exit?
 - release_agent: the path to use for release notifications (this file
   exists in the top cgroup only)

Other subsystems such as cpusets may add additional files in each
cgroup dir.

New cgroups are created using the mkdir system call or shell
command.  The properties of a cgroup, such as its flags, are
modified by writing to the appropriate file in that cgroups
directory, as listed above.

The named hierarchical structure of nested cgroups allows partitioning
a large system into nested, dynamically changeable, "soft-partitions".

The attachment of each task, automatically inherited at fork by any
children of that task, to a cgroup allows organizing the work load
on a system into related sets of tasks.  A task may be re-attached to
any other cgroup, if allowed by the permissions on the necessary
cgroup file system directories.

When a task is moved from one cgroup to another, it gets a new
css_set pointer - if there's an already existing css_set with the
desired collection of cgroups then that group is reused, otherwise a new
css_set is allocated. The appropriate existing css_set is located by
looking into a hash table.

To allow access from a cgroup to the css_sets (and hence tasks)
that comprise it, a set of cg_cgroup_link objects form a lattice;
each cg_cgroup_link is linked into a list of cg_cgroup_links for
a single cgroup on its cgrp_link_list field, and a list of
cg_cgroup_links for a single css_set on its cg_link_list.

Thus the set of tasks in a cgroup can be listed by iterating over
each css_set that references the cgroup, and sub-iterating over
each css_set's task set.

The use of a Linux virtual file system (vfs) to represent the
cgroup hierarchy provides for a familiar permission and name space
for cgroups, with a minimum of additional kernel code.
```

```
1.4 What does notify_on_release do ?
------------------------------------

If the notify_on_release flag is enabled (1) in a cgroup, then
whenever the last task in the cgroup leaves (exits or attaches to
some other cgroup) and the last child cgroup of that cgroup
is removed, then the kernel runs the command specified by the contents
of the "release_agent" file in that hierarchy's root directory,
supplying the pathname (relative to the mount point of the cgroup
file system) of the abandoned cgroup.  This enables automatic
removal of abandoned cgroups.  The default value of
notify_on_release in the root cgroup at system boot is disabled
(0).  The default value of other cgroups at creation is the current
value of their parents' notify_on_release settings. The default value of
a cgroup hierarchy's release_agent path is empty.

1.5 What does clone_children do ?
---------------------------------

This flag only affects the cpuset controller. If the clone_children
flag is enabled (1) in a cgroup, a new cpuset cgroup will copy its
configuration from the parent during initialization.

1.6 How do I use cgroups ?
--------------------------

To start a new job that is to be contained within a cgroup, using
the "cpuset" cgroup subsystem, the steps are something like:

 1) mount -t tmpfs cgroup_root /sys/fs/cgroup
 2) mkdir /sys/fs/cgroup/cpuset
 3) mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset
 4) Create the new cgroup by doing mkdir's and write's (or echo's) in
    the /sys/fs/cgroup virtual file system.
 5) Start a task that will be the "founding father" of the new job.
 6) Attach that task to the new cgroup by writing its PID to the
    /sys/fs/cgroup/cpuset/tasks file for that cgroup.
 7) fork, exec or clone the job tasks from this founding father task.

For example, the following sequence of commands will setup a cgroup
named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,
and then start a subshell 'sh' in that cgroup:

  mount -t tmpfs cgroup_root /sys/fs/cgroup
  mkdir /sys/fs/cgroup/cpuset
  mount -t cgroup cpuset -ocpuset /sys/fs/cgroup/cpuset
  cd /sys/fs/cgroup/cpuset
  mkdir Charlie
  cd Charlie
  /bin/echo 2-3 > cpuset.cpus
  /bin/echo 1 > cpuset.mems
  /bin/echo $$ > tasks
  sh
  # The subshell 'sh' is now running in cgroup Charlie
  # The next line should display '/Charlie'
  cat /proc/self/cgroup

2. Usage Examples and Syntax
============================
```

```
2.1 Basic Usage
---------------

Creating, modifying, using cgroups can be done through the cgroup
virtual filesystem.

To mount a cgroup hierarchy with all available subsystems, type:
# mount -t cgroup xxx /sys/fs/cgroup

The "xxx" is not interpreted by the cgroup code, but will appear in
/proc/mounts so may be any useful identifying string that you like.

Note: Some subsystems do not work without some user input first.  For instance,
if cpusets are enabled the user will have to populate the cpus and mems files
for each new cgroup created before that group can be used.

As explained in section `1.2 Why are cgroups needed?' you should create
different hierarchies of cgroups for each single resource or group of
resources you want to control. Therefore, you should mount a tmpfs on
/sys/fs/cgroup and create directories for each cgroup resource or resource
group.

# mount -t tmpfs cgroup_root /sys/fs/cgroup
# mkdir /sys/fs/cgroup/rg1

To mount a cgroup hierarchy with just the cpuset and memory
subsystems, type:
# mount -t cgroup -o cpuset,memory hier1 /sys/fs/cgroup/rg1

While remounting cgroups is currently supported, it is not recommend
to use it. Remounting allows changing bound subsystems and
release_agent. Rebinding is hardly useful as it only works when the
hierarchy is empty and release_agent itself should be replaced with
conventional fsnotify. The support for remounting will be removed in
the future.

To Specify a hierarchy's release_agent:
# mount -t cgroup -o cpuset,release_agent="/sbin/cpuset_release_agent" \
  xxx /sys/fs/cgroup/rg1

Note that specifying 'release_agent' more than once will return failure.

Note that changing the set of subsystems is currently only supported
when the hierarchy consists of a single (root) cgroup. Supporting
the ability to arbitrarily bind/unbind subsystems from an existing
cgroup hierarchy is intended to be implemented in the future.

Then under /sys/fs/cgroup/rg1 you can find a tree that corresponds to the
tree of the cgroups in the system. For instance, /sys/fs/cgroup/rg1
is the cgroup that holds the whole system.

If you want to change the value of release_agent:
# echo "/sbin/new_release_agent" > /sys/fs/cgroup/rg1/release_agent

It can also be changed via remount.

If you want to create a new cgroup under /sys/fs/cgroup/rg1:
# cd /sys/fs/cgroup/rg1
# mkdir my_cgroup
```

```
Now you want to do something with this cgroup.
# cd my_cgroup

In this directory you can find several files:
# ls
cgroup.procs notify_on_release tasks
(plus whatever files added by the attached subsystems)

Now attach your shell to this cgroup:
# /bin/echo $$ > tasks

You can also create cgroups inside your cgroup by using mkdir in this
directory.
# mkdir my_sub_cs

To remove a cgroup, just use rmdir:
# rmdir my_sub_cs

This will fail if the cgroup is in use (has cgroups inside, or
has processes attached, or is held alive by other subsystem-specific
reference).

2.2 Attaching processes
-----------------------

# /bin/echo PID > tasks

Note that it is PID, not PIDs. You can only attach ONE task at a time.
If you have several tasks to attach, you have to do it one after another:

# /bin/echo PID1 > tasks
# /bin/echo PID2 > tasks
    ...
# /bin/echo PIDn > tasks

You can attach the current shell task by echoing 0:

# echo 0 > tasks

You can use the cgroup.procs file instead of the tasks file to move all
threads in a threadgroup at once. Echoing the PID of any task in a
threadgroup to cgroup.procs causes all tasks in that threadgroup to be
attached to the cgroup. Writing 0 to cgroup.procs moves all tasks
in the writing task's threadgroup.

Note: Since every task is always a member of exactly one cgroup in each
mounted hierarchy, to remove a task from its current cgroup you must
move it into a new cgroup (possibly the root cgroup) by writing to the
new cgroup's tasks file.

Note: Due to some restrictions enforced by some cgroup subsystems, moving
a process to another cgroup can fail.

2.3 Mounting hierarchies by name
--------------------------------

Passing the name=<x> option when mounting a cgroups hierarchy
associates the given name with the hierarchy.  This can be used when
mounting a pre-existing hierarchy, in order to refer to it by name
rather than by its set of active subsystems.  Each hierarchy is either
```

```
nameless, or has a unique name.

The name should match [\w.-]+

When passing a name=<x> option for a new hierarchy, you need to
specify subsystems manually; the legacy behaviour of mounting all
subsystems when none are explicitly specified is not supported when
you give a subsystem a name.

The name of the subsystem appears as part of the hierarchy description
in /proc/mounts and /proc/<pid>/cgroups.


3. Kernel API
=============

3.1 Overview
------------

Each kernel subsystem that wants to hook into the generic cgroup
system needs to create a cgroup_subsys object. This contains
various methods, which are callbacks from the cgroup system, along
with a subsystem ID which will be assigned by the cgroup system.

Other fields in the cgroup_subsys object include:

- subsys_id: a unique array index for the subsystem, indicating which
  entry in cgroup->subsys[] this subsystem should be managing.

- name: should be initialized to a unique subsystem name. Should be
  no longer than MAX_CGROUP_TYPE_NAMELEN.

- early_init: indicate if the subsystem needs early initialization
  at system boot.

Each cgroup object created by the system has an array of pointers,
indexed by subsystem ID; this pointer is entirely managed by the
subsystem; the generic cgroup code will never touch this pointer.

3.2 Synchronization
-------------------

There is a global mutex, cgroup_mutex, used by the cgroup
system. This should be taken by anything that wants to modify a
cgroup. It may also be taken to prevent cgroups from being
modified, but more specific locks may be more appropriate in that
situation.

See kernel/cgroup.c for more details.

Subsystems can take/release the cgroup_mutex via the functions
cgroup_lock()/cgroup_unlock().

Accessing a task's cgroup pointer may be done in the following ways:
- while holding cgroup_mutex
- while holding the task's alloc_lock (via task_lock())
- inside an rcu_read_lock() section via rcu_dereference()

3.3 Subsystem API
-----------------
```

```
Each subsystem should:

- add an entry in linux/cgroup_subsys.h
- define a cgroup_subsys object called <name>_subsys

If a subsystem can be compiled as a module, it should also have in its
module initcall a call to cgroup_load_subsys(), and in its exitcall a
call to cgroup_unload_subsys(). It should also set its_subsys.module =
THIS_MODULE in its .c file.

Each subsystem may export the following methods. The only mandatory
methods are css_alloc/free. Any others that are null are presumed to
be successful no-ops.

struct cgroup_subsys_state *css_alloc(struct cgroup *cgrp)
(cgroup_mutex held by caller)

Called to allocate a subsystem state object for a cgroup. The
subsystem should allocate its subsystem state object for the passed
cgroup, returning a pointer to the new object on success or a
ERR_PTR() value. On success, the subsystem pointer should point to
a structure of type cgroup_subsys_state (typically embedded in a
larger subsystem-specific object), which will be initialized by the
cgroup system. Note that this will be called at initialization to
create the root subsystem state for this subsystem; this case can be
identified by the passed cgroup object having a NULL parent (since
it's the root of the hierarchy) and may be an appropriate place for
initialization code.

int css_online(struct cgroup *cgrp)
(cgroup_mutex held by caller)

Called after @cgrp successfully completed all allocations and made
visible to cgroup_for_each_child/descendant_*() iterators. The
subsystem may choose to fail creation by returning -errno. This
callback can be used to implement reliable state sharing and
propagation along the hierarchy. See the comment on
cgroup_for_each_descendant_pre() for details.

void css_offline(struct cgroup *cgrp);
(cgroup_mutex held by caller)

This is the counterpart of css_online() and called iff css_online()
has succeeded on @cgrp. This signifies the beginning of the end of
@cgrp. @cgrp is being removed and the subsystem should start dropping
all references it's holding on @cgrp. When all references are dropped,
cgroup removal will proceed to the next step - css_free(). After this
callback, @cgrp should be considered dead to the subsystem.

void css_free(struct cgroup *cgrp)
(cgroup_mutex held by caller)

The cgroup system is about to free @cgrp; the subsystem should free
its subsystem state object. By the time this method is called, @cgrp
is completely unused; @cgrp->parent is still valid. (Note - can also
be called for a newly-created cgroup if an error occurs after this
subsystem's create() method has been called for the new cgroup).

int can_attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
```

```
(cgroup_mutex held by caller)

Called prior to moving one or more tasks into a cgroup; if the
subsystem returns an error, this will abort the attach operation.
@tset contains the tasks to be attached and is guaranteed to have at
least one task in it.

If there are multiple tasks in the taskset, then:
  - it's guaranteed that all are from the same thread group
  - @tset contains all tasks from the thread group whether or not
    they're switching cgroups
  - the first task is the leader

Each @tset entry also contains the task's old cgroup and tasks which
aren't switching cgroup can be skipped easily using the
cgroup_taskset_for_each() iterator. Note that this isn't called on a
fork. If this method returns 0 (success) then this should remain valid
while the caller holds cgroup_mutex and it is ensured that either
attach() or cancel_attach() will be called in future.

void cancel_attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
(cgroup_mutex held by caller)

Called when a task attach operation has failed after can_attach() has succeeded.
A subsystem whose can_attach() has some side-effects should provide this
function, so that the subsystem can implement a rollback. If not, not necessary.
This will be called only about subsystems whose can_attach() operation have
succeeded. The parameters are identical to can_attach().

void attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
(cgroup_mutex held by caller)

Called after the task has been attached to the cgroup, to allow any
post-attachment activity that requires memory allocations or blocking.
The parameters are identical to can_attach().

void fork(struct task_struct *task)

Called when a task is forked into a cgroup.

void exit(struct task_struct *task)

Called during task exit.

void bind(struct cgroup *root)
(cgroup_mutex held by caller)

Called when a cgroup subsystem is rebound to a different hierarchy
and root cgroup. Currently this will only involve movement between
the default hierarchy (which never has sub-cgroups) and a hierarchy
that is being created/destroyed (and hence has no sub-cgroups).

4. Extended attribute usage
===========================

cgroup filesystem supports certain types of extended attributes in its
directories and files.  The current supported types are:
    - Trusted (XATTR_TRUSTED)
    - Security (XATTR_SECURITY)
```

```
Both require CAP_SYS_ADMIN capability to set.

Like in tmpfs, the extended attributes in cgroup filesystem are stored
using kernel memory and it's advised to keep the usage at minimum.  This
is the reason why user defined extended attributes are not supported, since
any user can do it and there's no limit in the value size.

The current known users for this feature are SELinux to limit cgroup usage
in containers and systemd for assorted meta data like main PID in a cgroup
(systemd creates a cgroup per service).

5. Questions
============

Q: what's up with this '/bin/echo' ?
A: bash's builtin 'echo' command does not check calls to write() against
   errors. If you use it in the cgroup file system, you won't be
   able to tell whether a command succeeded or failed.

Q: When I attach processes, only the first of the line gets really attached !
A: We can only return one error code per call to write(). So you should also
   put only ONE PID.
```

11.5 Docker Containers

11.5.1 Introduction to Docker Containers

11.5.1.1 Overview

This section is a guide and tutorial to building and using Docker Containers. Docker Containers are only available on ARM64 platforms, with the exception of LS1043 Big Endian.

Docker is a different set of userspace tools implementing Linux containers and focusing on a different set of use cases. The highlights of this open source project are ease of use, shared contributions and fast deployment. In the Docker ecosystem, containers are application environment packages, which can be easily distributed and developed collaboratively, and are guaranteed to be reproducible on any supporting platform, from the development stage to production. Currently, Docker containers are mainly targeting cloud environments.

Docker can be viewed as a set of separate components:

- **Images** - the "build" component of Docker. These are read-only copies of container root filesystems, consisting of the designed application and it's userspace dependencies. For example, an image can contain an Ubuntu application, an Apache server and a user web app. This image can be used to get a webserver running.
- **Registries** - the "distribution" component of Docker. These are public or private stores where users can upload / download images. The images are versioned, and are built from layers. When sharing images, the layers are first downloaded separately, and the image is assembled at runtime. Each layer corresponds to a specific user commit. Images can also be built using buildfiles. The most representative registry example is the Docker Hub. The current Docker installation does not support registry configuration.
- **Containers** - the "run" component of Docker. These are very similar to the containers provided by the LXC package. The main difference is that Docker containers use an overlay filesystem as container support. The layers are taken as is from the image and marked read-only, with a topmost read-write layer on top. This means that no container makes any persistent changes to the image by default - these need to be explicitly committed by the user when the environment is in the desired state. Docker containers are designed to work as application containers by default.

Docker uses a client-server architecture. The client takes the user commands and talks to a daemon, which does the entire container management work. A Linux host running the daemon is called a Docker Host. The client and daemon can run on the same machine, or on different ones, communicating through sockets or a RESTful API.

The Docker official page advertises a set of use cases, mostly relevant in cloud environments: continuous integration, continuous delivery, devops, big data and infrastructure optimization. These can be easily adapted to embedded distributions as well. As for the containers themselves, the Linux Containers chapter use cases apply, with a focus on ease of use, fast deployment and distributed usage.

11.5.2 Build and Installation

11.5.2.1 Building with Yocto

Docker is a Linux user space package that can easily be added to a rootfs using the Yocto build system.

In the NXP SDK, Docker and all pre-requisite user space packages are included when building the "virt" image type:

```
bitbake fsl-image-virt
```

Docker can be easily added to any rootfs image by updating the `IMAGE_INSTALL_append` variable in the conf/local.conf file in the Yocto build environment. For example, append the following line to local.conf:

```
IMAGE_INSTALL_append = " docker docker-registry"
```

If you are building for ARM64 platforms, you need to perform an additional step. You need to update the rootfs target in the board image - by default it is fsl-image-core. If you plan to update this rootfs to, say, fsl-image-virt, you need to add the following line in to build_<machine_release>/conf/local.conf:

```
ROOTFS_IMAGE = "fsl-image-virt”
```

After enabling the required Linux options mentioned in the following chapter, generate the kernel itb:

```
bitbake fsl-image-kernelitb
```

11.5.2.2 Building the Linux Kernel

In order to use Docker, the Linux kernel must be configured with the options required by LXC containers, and some others that are specific to the filesystem overlay and Docker's internal networking.

These options can be enabled automatically by building the Linux kernel with an additional config fragment. In order to do this, add the following line in the **conf/local.conf** file in the Yocto build environment:

```
DELTA_KERNEL_DEFCONFIG_append = " <sdk-devel>/sources/meta-freescale/recipes-kernel/linux/files/
containers.config"
```

Alternatively, you can enable the options manually. Please make sure you have enabled the options required by LXC containers first. To configure and build the Linux kernel:

```
bitbake virtual/kernel -c cleansstate
bitbake virtual/kernel -c menuconfig
bitbake virtual/kernel
```

Make sure that the following options are also enabled (besides the ones mentioned in the LXC chapter):

```
[*] Networking support  --->
  Networking options  --->
    [*] TCP/IP networking
    < >   The IPv6 protocol
    [*] Network packet filtering framework (Netfilter)  --->
      Core Netfilter Configuration  --->
        <*> Netfilter connection tracking support
        <*> Netfilter nf_tables support
        <*>   Netfilter nf_tables conntrack module
        <*>   Netfilter nf_tables rbtree set module
        <*>   Netfilter nf_tables masquerade support
        <*>   Netfilter nf_tables nat module
        <*> Netfilter x_tables over nf_tables module
        <*> Netfilter Xtables support (required for ip_tables)
        <*>   "CONNMARK" target support
        <*>   "addrtype" address type match support
        <*>   "connmark" connection mark match support
        <*>   "conntrack" connection tracking match support
      IP: Netfilter Configuration  --->
        <*> IPv4 connection tracking support (required for NAT)
        <*> IPv4 nf_tables support
        <*> IP tables support (required for filtering/masq/NAT)
        <*>   Packet filtering
        <*>   iptables NAT support
        <*>     MASQUERADE target support
        <*>   Packet mangling
      <*>   Ethernet Bridge nf_tables support  --->
      <*>   Ethernet Bridge tables (ebtables) support  --->
        <*>   ebt: nat table support
        <*>   ebt: dnat target support
        <*>   ebt: snat target support
File systems  --->
  <*> Overlay filesystem support
  Pseudo filesystems  --->
    [*] Tmpfs virtual memory file system support (former shm fs)
    [*]   Tmpfs POSIX Access Control Lists
    -*-   Tmpfs extended attributes
```

11.5.3 Docker How To's

11.5.3.1 Running a webserver container

The following article describes the necessary steps to deploy a web server service using a Docker container. This is based on downloading a prepared image from the Docker hub and using it to start a container.

1. Make sure you have sufficient space on the underlying filesystem where `/var/lib` resides. In the test setup, the root filesystem is mounted on a small flash memory, not sufficient for downloading the Docker image. Therefore, `/var/lib` is transferred to a RAM filesystem mount.

   ```
   # Docker - temporary /var/lib mount
   mkdir ~/var_lib
   cp -R /var/lib/* ~/var_lib
   ```

```
mount -t tmpfs var_lib_tmpfs /var/lib
cp -R ~/var_lib/* /var/lib/
```

2. Start the docker daemon. Make sure that the board has Internet access - this will be required to download the image from the Docker Hub. The daemon will configure a Linux bridge for the containers with a private network and NAT.

```
:~# docker daemon
ERRO[0000] Failed to built-in GetDriver graph btrfs /var/lib/docker
ERRO[0000] Failed to built-in GetDriver graph devicemapper /var/lib/docker
INFO[0000] API listen on /var/run/docker.sock
INFO[0000] Default bridge (docker0) is assigned with an IP address 172.17.0.0/16. Daemon option
--bip can be used to set a preferred IP address
INFO[0000] Loading containers: start.

INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=7e5506d-dirty
execdriver=native-0.2 graphdriver=overlay version=1.9.0
```

3. You can search the registry for available **arm64** images, or using any other keyword.

```
:~# docker search arm64
NAME                                      DESCRIPTION
STARS     OFFICIAL   AUTOMATED
ericvh/arm64-ubuntu                       Base image for arm64 (armv8 aka aarch64) U...
5
ericvh/arm64-ubuntu-dev                   arm64 Ubuntu install with basic developmen...
3
ericvh/arm64-ubuntu-hpc                   arm64 (armv8) root file system with multia...
2
markusk/arm64-crosscompile                A debian image with the necessary tools in...
1                    [OK]
jefby/arm64                               arm64 develop
0
mickaelguene/arm64-debian                 arm64 debian with umeq inside. To be use w...
0
mickaelguene/arm64-debian-dev             arm64 debian development image
0
inaz2/debian-arm64                        Debian arm64 with QEMU user-mode emulation
0
mickaelguene/arm64-debian-jenkins-slave   This is a basic container to be used as a ...
0
bobsense/redis-arm64                      A Docker Image for Redis On ARM64
0
zlim/arm64-ubuntu                         aarch64 base (Ubuntu 15.04)
0
alisw/arm64-builder
0
varakur/arm64-coreos                      hand built image of arm64  CoreOS includin...
0
djtm/syncthing-scratch-arm64              syncthing scratch image
0
jefby/centos-arm64                        centos7 for aarch64
0
qoriq/arm64-ubuntu
0
```

```
justinzh/arm64-vivid
0
kickinz1/ubuntu-arm64
0
zlim/arm64-ubuntu-dev                      aarch64 development
0
vducuy/ubuntu-wily-arm64                   First test with ubuntu-15.10 for ARM64
0
myejeo01/arm64-ubuntu-bench                Working copy of arm aarch64 for HPC.
0
bobsense/nginx-arm64                       A Docker Image for Nginx on ARM64
0
arm64el/helloworld-arm64el                 hello world for arm64 el platform
0                    [OK]
arm64el/busybox-arm64el                    busybox  image for arm64
0                    [OK]
arm64el/unshare-arm64el                    unshare image for arm64el platform
0                    [OK]
```

4. In this example, `qoriq/arm64-ubuntu` is used. It's a standard Ubuntu compiled for ARM64, with a lighttpd webserver installed and with a home page configured to display some information on the LS2085A boards, processes and networking in the container. First download the image.

```
:~# docker pull qoriq/arm64-ubuntu
Using default tag: latest
latest: Pulling from qoriq/arm64-ubuntu

0441656ea204: Pull complete
79560570ed4e: Pull complete
938b8dde536d: Pull complete
5fa49f1394a1: Pull complete
a6acd691f8a3: Pull complete
276092d595eb: Pull complete
e5a142352d65: Pull complete
bd635402e399: Pull complete
6f799e234e33: Pull complete
b114273a61d4: Pull complete
8c7ce7746811: Pull complete
93e0b6a1ed99: Pull complete
4e549f05ce07: Pull complete
ae2db5c579b6: Pull complete
12ce21189ebb: Pull complete
efe3b8f0d401: Pull complete
f579adcec7f6: Pull complete
885db99879a8: Pull complete
Digest: sha256:eaef3a08336f59155e6cfb61bf55688711214561ddf00817b5c848211ac66b00
Status: Downloaded newer image for qoriq/arm64-ubuntu:latest
```

You can check the image is available using `docker images`:

```
:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
qoriq/arm64-ubuntu  latest              885db99879a8        59 minutes ago      326.4 MB
```

5. Start a container using the following command:

```
:~# docker run -d -p 30081:80 --name=sandbox1 -h sandbox1 qoriq/arm64-ubuntu bash -c "lighttpd -
f /etc/lighttpd/lighttpd.conf -D"
468a1e4c474414379cd0dab810a7883d8037db4795b3287526ad051741ef2525
```

- `run` - create and start the container. Optionally, download the image if not available on the host.
- `-d` - start the container as a daemon.
- `-p 30081:80` - forward port 80 in the container to port 30081 on the board.
- `--name=sandbox1` - the name of the container (as visible to Docker).
- `-h sandbox1` - the hostname inside the container.
- `qoriq/arm64-ubuntu` - the base image for the container.
- `bash -c "lighttpd -f /etc/lighttpd/lighttpd.conf -D"` - the command to execute as PID 1 in the container.

The command will return a unique SHA for the container. You can check that the webserver is up and running by accessing `http://BOARD_IP:30081/` from a browser. You can also check the container is running using docker:

```
:~# docker ps -a
CONTAINER ID        IMAGE                      COMMAND                  CREATED
STATUS              PORTS                   NAMES
468a1e4c4744        qoriq/arm64-ubuntu         "bash -c 'lighttpd -f"   About a minute ago
Up About a minute   0.0.0.0:30081->80/tcp   sandbox1
```

6. Stopping and deleting the container are easy operations:

```
:~# docker stop sandbox1
sandbox1
:~# docker rm sandbox1
sandbox1
```

7. A similar command can be used to delete the image from the board.

```
:~# docker rmi qoriq/arm64-ubuntu
Untagged: qoriq/arm64-ubuntu:latest
Deleted: 885db99879a87371cf08e6fff99a89af6306cb410c013a967461c387a2d0d638
Deleted: f579adcec7f6e5589e9ded8e6d08db020e1475984c6a9c3dc72c43726c51d015
Deleted: efe3b8f0d40139a4218331d12172087b8a77feba7944c07343f994746e09e71c
Deleted: 12ce21189ebb2ccfdf36c12a349dfe4ff1095873cb3896a3b7bf6eb6bfc390c7
Deleted: ae2db5c579b6c5da7c83fc8c37a2247c7b268323fdf38f97da64daec77c9cc77
Deleted: 4e549f05ce071ec8067b44848c84d037fa74ddad785a8ee3fff86fcaa8151ca2
Deleted: 93e0b6a1ed99925ad1b0c1cdb94b3fd2a35b17379dc0723931fcc2c958556ebe
Deleted: 8c7ce774681151e92ee43e908e492488e5ec2928268ab852c657058f03ec6dc6
Deleted: b114273a61d4cbd55c702df480983e96e192ce9d5acbb1f1a2b1758c4bd07286
Deleted: 6f799e234e330ab6b4aac71de0dcd95733af609fb72ac4aae0f30dc783a5fa87
Deleted: bd635402e3999f95efc850a52c9039b720628043ff8e8e9569636f6fc7476d6f
Deleted: e5a142352d65e82ee9ee37855e210853d4db3ad7778ec3207dac91197e71b8e2
Deleted: 276092d595eb32656b16717a2750bcd88d6fe92cc8d00c8f52651548f76da3e6
Deleted: a6acd691f8a3a2717bdec3d1c930c3eef7df2ecd3b41bef4bf7376a74df426c5
Deleted: 5fa49f1394a1b10ab403398ad5768187ed60ea015f9f0110f6ef03bb3aa4a575
```

```
Deleted: 938b8dde536d2ddcf236efd423170a1d827690b8f19b9c0486364e08dbd8fedf
Deleted: 79560570ed4ec7d9ae02a3f1d904cbae9427bc368580fd324c2273e6bce8b274
Deleted: 0441656ea20494181a7b0b2eaf90381628ab4b5621018bb10d4c60ef03a3b412
```

11.6 User Space DPDK with OVS & Virtio Devices

For more information on using OVS-DPDK in user space with VIRTIO interfaces, refer to OVS-DPDK and DPDK in VM with VIRTIO Interfaces on page 1333.

Chapter 12
Benchmark Reproducibility Guides

12.1 Linux Core

Linux Core - Benchmark Reproducibility Guides

12.1.1 Coremark

How to setup a board to baseline Coremark performance for QorIQ processor development boards.

12.1.1.1 Test Environment

Objectives

The first objective is to baseline the Coremark performance on QorIQ platform. Next identify any optimizations that have been discovered and ensure they are implemented on the those platforms. Investigate other changes that may improve performance.

Hardware Platform Identification

Table 594. Target Platform

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR) in MHz	Core Type
P2040RDB	Rev2.0	1200/600/1200	e500mc
P3041DS	Rev2.0	1500/750/1333	e500mc
P4080DS	Rev3.0	1500/800/1300	e500mc
P5020DS	Rev2.0	2000/800/1333	e5500
P5040DS	Rev2.1	2266/800/1600	e5500
T4240QDS	Rev2.0	1800/733/1867	e6500
T4240RDB	Rev2.0	1800/733/1867	e6500
B4860QDS	Rev2.2	1600/667/1867	e6500
T2080QDS	Rev1.1	1800/600/1867	e6500
T2080RDB	Rev1.1	1800/600/1867	e6500
T1040D4RDB	Rev1.1	1400/600/1600	e5500
T1024RDB	Rev1.0	1400/400/1600	e5500
T1023RDB	Rev1.0	1200/400/1600	e5500
LS1021ATWR	Rev2.0	1000/300/1600	cortex A7
LS1043ARDB	Rev1.0	1600/400/1600	cortex A53
LS2080ARDB	Rev1.0	1867/533/1867	cortex A57

For board switch settings, please see to the corresponding reference manual.

Software Platform Identification

All software was built from QorIQ SDK ISO.

Boot Loader

U-boot 2016.01 with NXP-specific patches on top;

Coremark Application

Source Code Download

Coremark Source code can be downloaded from www.coremark.org

Toolchain version is gcc-4.9.2 with glibc-2.21 for SDK v2.0

Build coremark

1. If you are compiling CoreMark on 64 bit linux machine (machine on which you have the intended compiler) go to coremark_v1.0/linux64 directory, else go to coremark_v1.0/linux directory
2. Give the complete compiler path under "CC" flag in core_portme.mak file

```
For example: e6500 toolchain path of sdkv1.9:
CC = /opt/fsl-qoriq/1.9/sysroots/x86_64-fsl_networking_sdk-linux/usr/bin/powerpc-
fsl_networking-linux-gcc
```

3. Go back to coremark_v1.0 directory. Give following command on the command line
   - For e500mc:

```
make PORT_CFLAGS="-mcpu=e500mc -O3 -fipa-pta -funroll-all-loops -ftree-loop-ivcanon -
fvariable-expansion-in-unroller -fpeel-loops -floop-interchange --param max-inline-
insns-auto=1000 --param max-average-unrolled-insns=99999999 -fgcse-las -fgcse-after-
reload  -fwiden-types -static --sysroot=/opt/fsl-qoriq/2.0/sysroots/ppce500mc-fsl-
linux”
```

   - For e5500-32bit:

```
make PORT_CFLAGS="-mcpu=e5500 -m64 -mcmodel=small -O3 -fipa-pta -funroll-all-loops -
ftree-loop-ivcanon -fvariable-expansion-in-unroller -floop-block -fno-auto-inc-dec --
param max-inline-insns-auto=1000 --param max-average-unrolled-insns=99999999 --param
iv-max-considered-uses=9999999 -fno-ira-loop-pressure -fsched-pressure -fmodulo-sched
-fmodulo-sched-allow-regmoves -fwiden-types -fno-align-loops -static --sysroot=/opt/
fsl-qoriq/2.0/sysroots/ppc64e5500-fsl-linux”
```

   - For e5500-64bit:

```
make PORT_CFLAGS="-mcpu=e5500 -m64 -mcmodel=small -O3 -fipa-pta -funroll-all-loops -
ftree-loop-ivcanon -fvariable-expansion-in-unroller -floop-block -fno-auto-inc-dec --
param max-inline-insns-auto=1000 --param max-average-unrolled-insns=99999999 --param
iv-max-considered-uses=9999999 -fno-ira-loop-pressure -fsched-pressure -fmodulo-sched
-fmodulo-sched-allow-regmoves -fwiden-types -fno-align-loops -static --sysroot=/opt/
fsl-qoriq/2.0/sysroots/ppc64e5500-fsl-linux "
```

   - For e6500-32bit:

```
Single thread:
make PORT_CFLAGS="-mcpu=e6500 -m32 -O3 -finline-limit=99999 -fgcse-sm -fgcse-las  -
fopt-array-offset -frename-registers -fsched-spec-load -fsched-spec-load-dangerous -
funroll-all-loops -ftree-loop-ivcanon -fvariable-expansion-in-unroller -funsafe-loop-
optimizations -funswitch-loops -static --sysroot=/opt/fsl-qoriq/2.0/sysroots/ppce6500-
fsl-linux"
Multithread:
make PORT_CFLAGS="-mcpu=e6500 -m32 -O3 -finline-limit=99999 -fgcse-sm -fgcse-las  -
fopt-array-offset -frename-registers -fsched-spec-load -fsched-spec-load-dangerous -
funroll-all-loops -ftree-loop-ivcanon -fvariable-expansion-in-unroller -funsafe-loop-
```

```
optimizations -funswitch-loops -static --sysroot=/opt/fsl-qoriq/2.0/sysroots/ppce6500-
fsl-linux -DMULTITHREAD=2/4/8/16/24 -DUSE_FORK=1"
```

- For e6500-64bit:

```
Single thread:
make PORT_CFLAGS="-mcpu=e6500 -m64 -mcmodel=small -O3 -finline-limit=99999 -fsched-
pressure -funroll-all-loops -ftree-loop-ivcanon -fvariable-expansion-in-unroller -
funsafe-loop-optimizations --param max-average-unrolled-insns=99999999 --sysroot=/opt/
fsl-qoriq/2.0/sysroots/ppc64e6500-fsl-linux -static"
Multithread:
make PORT_CFLAGS="-mcpu=e6500 -m64 -mcmodel=small -O3 -finline-limit=99999 -fsched-
pressure -funroll-all-loops -ftree-loop-ivcanon -fvariable-expansion-in-unroller -
funsafe-loop-optimizations --param max-average-unrolled-insns=99999999 --sysroot=/opt/
fsl-qoriq/2.0/sysroots/ppc64e6500-fsl-linux -static -DMULTITHREAD=<2/4/8/12/16/20/24>
-DUSE_FORK=1"
```

- For cortex A53 (LS1043A)

```
Single thread:
make PORT_CFLAGS = "-mcpu=cortex-a53 -march=armv8-a+fp+simd+crc+crypto -O3 -funroll-
all-loops --param max-inline-insns-auto=550 --param case-values-threshold=30 -falign-
functions=32 -ftracer --sysroot=/opt/fsl-qoriq/2.0/sysroots/aarch64-fsl-linux"
Multithread:
make PORT_CFLAGS = "-mcpu=cortex-a53 -march=armv8-a+fp+simd+crc+crypto -O3 -funroll-
all-loops --param max-inline-insns-auto=550 --param case-values-threshold=30 -falign-
functions=32 -ftracer --sysroot=/opt/fsl-qoriq/2.0/sysroots/aarch64-fsl-linux -
DMULTITHREAD=4 -DUSE_FORK=1"
```

- For cortex A57 (LS2080A)

```
Single thread:
make PORT_CFLAGS = "-mcpu=cortex-a57 -march=armv8-a+fp+simd+crc+crypto -O3 -funroll-
all-loops --param max-inline-insns-auto=550 --param case-values-threshold=30 -falign-
functions=32 -ftracer --sysroot=/opt/fsl-qoriq/2.0/sysroots/aarch64-fsl-linux"
Multithread:
make PORT_CFLAGS = "-mcpu=cortex-a57 -march=armv8-a+fp+simd+crc+crypto -O3 -funroll-
all-loops --param max-inline-insns-auto=550 --param case-values-threshold=30 -falign-
functions=32 -ftracer --sysroot=/opt/fsl-qoriq/2.0/sysroots/aarch64-fsl-linux -
DMULTITHREAD=4 -DUSE_FORK=1"
```

- For cortex A7 (LS1021A and LS1043A/LS2080A 32bit)

```
Single thread:
make PORT_CFLAGS="-O3 -march=armv7-a -mfloat-abi=hard -mfpu=neon -mtune=cortex-a7 -
funroll-all-loops --param max-inline-insns-auto=300 -static --sysroot=/opt/fsl-
qoriq/2.0/sysroots/cortexa7hf-vfp-neon-fsl-linux-gnueabi"
Multithread:
make PORT_CFLAGS="-O3 -march=armv7-a -mfloat-abi=hard -mfpu=neon -mtune=cortex-a7 -
funroll-all-loops --param max-inline-insns-auto=300 -static --sysroot=/opt/fsl-
qoriq/2.0/sysroots/cortexa7hf-vfp-neon-fsl-linux-gnueabi -DMULTITHREAD=2 -DUSE_FORK=1"
```

4. The command will first compile, generate the executable file (coremark.exe) and try to run the benchmark. Transfer the executable file to the target

12.1.1.2 Test Procedure

Test procedure for lmbench.

Running test and result collection

Deploy target board with corresponding software mentioned in previous section.

Put coremark binary compiled with optimized flags mentioned in section test environment on target board

Run the benchmark

```
coremark.exe
```

and result will like as followings:

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 19508
Total time (secs): 19.508000
Iterations/Sec   : 5638.712323
Iterations       : 110000
Compiler version : GCC4.6.2
Compiler flags   : -O3 -mcpu=e500mc -misel=yes -funroll-all-loops  --param max-inline-
insns-auto=550 -ftracer -falign-jumps=16 -ftree-loop-im -fivopts -ftree-loop-ivcanon -
fvect-cost-model -fvariable-expansion-in-unroller --param max-unrolled-insns=999999 --
param max-average-unrolled-insns=99999999 --param iv-max-considered-uses=9999999 --param
iv-consider-all-candidates-bound=99999 --param iv-always-prune-cand-set-bound=999999   -
fmodulo-sched -fmodulo-sched-allow-regmoves -fgcse-lm -fgcse-sm -fgcse-las -funsafe-loop-
optimizations -freschedule-modulo-scheduled-loops -ftree-vectorize -DPERFORMANCE_RUN=1  -
lrt
Memory location  : Please put data memory location here
                        (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x33ff
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 5638.712323 / GCC4.6.2 -O3 -mcpu=e500mc -misel=yes -funroll-all-loops  --
param max-inline-insns-auto=550 -ftracer -falign-jumps=16 -ftree-loop-im -fivopts -ftree-
loop-ivcanon -fvect-cost-model -fvariable-expansion-in-unroller --param max-unrolled-
insns=999999 --param max-average-unrolled-insns=99999999 --param iv-max-considered-
uses=9999999 --param iv-consider-all-candidates-bound=99999 --param iv-always-prune-cand-
set-bound=999999   -fmodulo-sched -fmodulo-sched-allow-regmoves -fgcse-lm -fgcse-sm -fgcse-
las -funsafe-loop-optimizations -freschedule-modulo-scheduled-loops -ftree-vectorize -
DPERFORMANCE_RUN=1  -lrt / Heap
```

This test measures the variation of the benchmark results, so an average across 5 runs was taken for every result.

12.1.2 Dhrystone

How to setup a board to baseline Dhrystone performance for QorIQ processor development boards.

12.1.2.1 Test Environment

Objectives

The first objective is to baseline the Dhrystone performance on QorIQ platform. Next identify any optimizations that have been discovered and ensure they are implemented on the those platforms. Investigate other changes that may improve performance. Compare these results with other NXP and competitors products. Provide assistance to any customer activities that could result in design wins. Review the results with. Finally post the results and this guild so that groups within NXP may share them with customers.

Hardware Platform Identification

Table 595. Target Platform

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR)	Core Type
P2040RDB	Rev2.0	1200/600/1200	e500mc
P3041DS	Rev2.0	1500/750/1333	e500mc
P4080DS	Rev3.0	1500/800/1300	e500mc

Table continues on the next page...

Table 595. Target Platform (continued)

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR)	Core Type
P5020DS	Rev2.0	2000/800/1333	e5500
P5040DS	Rev2.1	2266/800/1600	e5500
T4240QDS	Rev2.0	1800/733/1867	e6500
T4240RDB	Rev2.0	1800/733/1867	e6500
B4860QDS	Rev2.2	1600/667/1867	e6500
T2080QDS	Rev1.1	1800/600/1867	e6500
T2080RDB	Rev1.1	1800/600/1867	e6500
T1040D4RDB	Rev1.1	1400/600/1600	e5500
T1024RDB	Rev1.0	1400/400/1600	e5500
T1023RDB	Rev1.0	1200/400/1600	e5500
LS1021ATWR	Rev2.0	1000/300/1600	cortex A7

For board switch settings, please see to the corresponding reference manual.

Software Platform Identification

All software was built from QorIQ SDK ISO.

Boot Loader

U-boot 2016.01 with NXP-specific patches on top.

Dhrystone Application

Benchmark methodology

Dhrystone is a synthetic computing benchmark program intended to be representative of system (integer) programming. It is a simple program that is carefully designed to statistically mimic the processor usage of some common set of programs. It is also have some pills, for the performance will be affected by many factors such as the compiler, libraries and so on.

Source Code Download

Dhrystone 1.0 Source code can be downloaded from `http://www.xanthos.se/~joachim/vaxmips.html (Dhrystone-src.tar.gz)`

Toolchain version is gcc-4.9.2 with glibc-2.21 for SDK v2.0

Build source code

1. Download the source code from `http://www.xanthos.se/~joachim/vaxmips.html (Dhrystone-src.tar.gz)`
2. Unpack the package
3. Go back to dhrystone_v1.0 directory. Apply patches for different platform:
   - For e500v2:

     Please contact your sales representatives for this patch.
   - For e500mc and e5500 32bit:

Please contact your sales representatives for this patch.

- For e5500 64bit:

  Please contact your sales representatives for this patch.

- For e6500:

  Please contact your sales representatives for this patch.

- For e6500 64bit:

  Please contact your sales representatives for this patch.

- For LS1021A:

  Please contact your sales representatives for this patch.

The command will first compile, generate the executable file (dhrystone.exe) and try to run the benchmark. Transfer the executable file to the target.

12.1.2.2 Test Procedure

Test procedure for lmbench.

Running test and result collection

Deploy target board with corresponding software mentioned in previous section.

Put dhrystone binary compiled with optimized flags mentioned in section 2.2.2.3 on target board

Run the benchmark

```
echo 50000000 | ./dhrystone.exe
```

and result will like as followings:

```
Dhrystone Benchmark, Version 2.1 (Language: C)

Please give the number of runs through the benchmark:
Execution starts, 50000000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    50000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          268525944
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
```

```
  Ptr_Comp:          268525944
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

Register option selected?  NO
Microseconds for one run through Dhrystone:     0.4
Dhrystones per Second:                       2689328.7
VAX MIPS rating =   1530.637
```

There is a run to run variation in benchmark result, so an average across 5 runs was taken for every result

12.1.3 EEMBC

How to setup a board to baseline EEMBC Netmark performance for QorIQ processor development boards.

12.1.3.1 Test Environment

Objectives

The first objective is to baseline the EEMBC Netmark performance on QorIQ platform. Next identify any optimizations that have been discovered and ensure they are implemented on the those platforms. Investigate other changes that may improve performance. Compare these results with other NXP and competitors products. Provide assistance to any customer activities that could result in design wins. Review the results with. Finally post the results and this guild so that groups within NXP may share them with customers.

Hardware Platform Identification

Table 596. Target Platform

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR)	Core type
P2040RDB	Rev2.0	1200/600/1200	e500mc
P3041DS	Rev2.0	1500/750/1333	e500mc
P4080DS	Rev3.0	1500/800/1300	e500mc
P5020DS	Rev2.0	2000/800/1333	e5500
P5040DS	Rev2.1	2266/800/1600	e6500
T4240QDS	Rev2.0	1800/733/1867	e6500

Table continues on the next page...

Table 596. Target Platform (continued)

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR)	Core type
T4240RDB	Rev2.0	1800/733/1867	e6500
B4860QDS	Rev2.2	1600/667/1867	e6500
T2080QDS	Rev1.1	1800/600/1867	e6500
T2080RDB	Rev1.1	1800/600/1866	e6500
T1040D4RDB	Rev1.1	1400/600/1600	e5500
T1024RDB	Rev1.0	1400/400/1600	e5500
T1023RDB	Rev1.0	1200/400/1600	e5500
LS1021ATWR	Rev2.0	1000/300/1600	cortex A7
LS1043ARDB	Rev1.0	1600/400/1600	cortex A53
LS2080ARDB	Rev1.0	1867/533/1867	cortex A57

For board switch settings, please see to the corresponding reference manual.

Software Platform Identification

All software was built from QorIQ SDK ISO.

Boot Loader

U-boot 2016.01 with NXP-specific patches on top; Configuration for different platforms is as follows:

- EEMBC Netmark Application
- Source Code Download
- EEMBC Netmark version2 Source code can be downloaded from `www.eembc.org`
- Toolchain version: see Components section

Endianness

```
For PowerPC architecture:
Define correct Endianness by either modifying th_lite/<platform>/al/thcfg.h to

#if !defined( EE_BIG_ENDIAN ) && !defined( EE_LITTLE_ENDIAN)
#define EE_BIG_ENDIAN    (TRUE)
#define EE_LITTLE_ENDIAN (FALSE)
#endif

Or by modifying /util/make/gcc.mak to

COMPILER_DEFINES += -DEE_BIG_ENDIAN=1 -DEE_LITTLE_ENDIAN=0
```

```
For ARM architecture:
Define correct Endianness by either modifying th_lite/<platform>/al/thcfg.h to

#if !defined( EE_BIG_ENDIAN ) && !defined( EE_LITTLE_ENDIAN)
#define EE_BIG_ENDIAN    (FALSE)
#define EE_LITTLE_ENDIAN (TRUE)
#endif

Or by modifying /util/make/gcc.mak to

COMPILER_DEFINES += -DEE_BIG_ENDIAN=1 -DEE_LITTLE_ENDIAN=1
```

Data Types

```
th_lite/<platform>/al/eembc_dt.h has various data types definitions
If not already done, do:
#define HAVE_64    (1)

The default definition for data types is:
typedef unsigned long          e_u24;
typedef signed   long       e_s24;

typedef unsigned long         e_u32;
typedef signed   long       e_s32;

However, as ppc follows I32LP64, in case of 64 bit execution long will be considered 64 bit.
So the data type definitions should be changed to:

typedef unsigned int         e_u24;
typedef signed   int       e_s24;

typedef unsigned int        e_u32;
typedef signed   int       e_s32;
```

Build the benchmark with following compiler flags:

```
Cortex A7:

TOOLS = /opt/fsl-qoriq/2.0/sysroots/x86_64-fslsdk-linux/usr/bin/arm-fsl-linux-gnueabi/
CC = $(TOOLS)/arm-fsl-linux-gnueabi-gcc
AS = $(TOOLS)/arm-fsl-linux-gnueabi-as
LD = $(TOOLS)/arm-fsl-linux-gnueabi-gcc
AR = $(TOOLS)/arm-fsl-linux-gnueabi-ar
SIZE = $(TOOLS)/arm-fsl-linux-gnueabi-size

#Both TCPMark and IPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=cortex-a7 -mtune=cortex-a7 -O3 -funroll-all-loops -ftree-vectorize -flto -
fwhole-program -fgcse-las
LINKER_FLAGS   =  -lm -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppce500v2-fsl-linux-gnuspe

Cortex A53:
TOOLS = /opt/fsl-qoriq/2.0/sysroots/x86_64-fslsdk-linux/usr/bin/aarch64-fsl-linux

CC = $(TOOLS)/aarch64-fsl-linux-gcc --sysroot=/opt/fsl-qoriq/2.0/sysroots/aarch64-fsl-linux
AS = $(TOOLS)/aarch64-fsl-linux-as
LD = $(TOOLS)/aarch64-fsl-linux-gcc
AR = $(TOOLS)/aarch64-fsl-linux-ar

COMPILER_FLAGS = -mcpu=cortex-a53 -march=armv8-a+fp+simd+crc+crypto -O3 -funroll-all-loops -ftree-
vectorize
LINKER_FLAGS   =  -lm -static --sysroot=/opt/fsl-qoriq/2.0/sysroots/aarch64-fsl-linux
SIZE = $(TOOLS)/aarch64-fsl-linux-size


Cortex A57:
TOOLS = /opt/fsl-qoriq/2.0/sysroots/x86_64-fslsdk-linux/usr/bin/aarch64-fsl-linux

CC = $(TOOLS)/aarch64-fsl-linux-gcc --sysroot=/opt/fsl-qoriq/2.0/sysroots/aarch64-fsl-linux
AS = $(TOOLS)/aarch64-fsl-linux-as
LD = $(TOOLS)/aarch64-fsl-linux-gcc
AR = $(TOOLS)/aarch64-fsl-linux-ar
```

```
COMPILER_FLAGS = -mcpu=cortex-a57 -march=armv8-a+fp+simd+crc+crypto -O3 -funroll-all-loops -ftree-
vectorize
LINKER_FLAGS   =  -lm -static --sysroot=/opt/fsl-qoriq/2.0/sysroots/aarch64-fsl-linux
SIZE = $(TOOLS)/aarch64-fsl-linux-size

E500mc:

TOOLS = /opt/fsl-qoriq/1.9/sysroots/x86_64-fslsdk-linux/usr/bin/powerpc-fsl-linux/
CC = $(TOOLS)/powerpc-fsl-linux-gcc
AS = $(TOOLS)/powerpc-fsl-linux-as
LD = $(TOOLS)/powerpc-fsl-linux-gcc
AR = $(TOOLS)/powerpc-fsl-linux-ar

For TCPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=e500mc -O3 -funroll-all-loops -ftree-loop-ivcanon -fvariable-expansion-in-
unroller -fbranch-target-load-optimize -fivopts -ftree-loop-distribution -frename-registers -fno-
ira-loop-pressure -fsched-pressure -fsched-spec-load -fsched-spec-load-dangerous -fmodulo-sched -
fsched2-use-superblocks -fselective-scheduling -fselective-scheduling2 -fsel-sched-pipelining --
param max-inline-insns-auto=1000 --param max-unrolled-insns=999999 -fdelete-null-pointer-checks -
fbypass-load-on-store -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppce500mc-fsl-linux


For IPMark, use following compilerf flags:
COMPILER_FLAGS = -mcpu=e500mc -O3 -flto -ffat-lto-objects -fwhole-program -frename-registers -fno-
ira-loop-pressure -fsched-pressure -fsched-spec-load -fsched-spec-load-dangerous -fmodulo-sched -
fmodulo-sched -fmodulo-sched-allow-regmoves -fsched2-use-superblocks -fselective-scheduling -
fselective-scheduling2 -fsel-sched-pipelining -fno-builtin-memcpy -fgcse-sm -fgcse-las -fgcse-
after-reload -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppce500mc-fsl-linux


LINKER_FLAGS   =  -lm -m32 -mcpu=e500mc -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppce500mc-
fsl-linux
SIZE   = $(TOOLS)/powerpc-fsl-linux-size

E5500 32bit:

TOOLS = /opt/fsl-qoriq/1.9/sysroots/x86_64-fslsdk-linux/usr/bin/powerpc-fsl-linux/
CC = $(TOOLS)/powerpc-fsl-linux-gcc
AS = $(TOOLS)/powerpc-fsl-linux-as
LD = $(TOOLS)/powerpc-fsl-linux-gcc
AR = $(TOOLS)/powerpc-fsl-linux-ar

For TCPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=e5500 -m32 -O3 -ftracer -funroll-all-loops -ftree-loop-ivcanon -fvariable-
expansion-in-unroller -ffunction-sections -fdata-sections -fbranch-target-load-optimize -fsection-
anchors -floop-strip-mine -ftree-loop-distribution -ftree-loop-if-convert-stores -ftree-loop-
distribute-patterns -finline-limit=40 -fira-region=all --param sched-mem-true-dep-cost=4 --param
max-inline-insns-auto=10 --param max-inline-insns-auto=1000 --param max-cselib-memory-
locations=9999 --param max-average-unrolled-insns=99999999 --param iv-max-considered-uses=9999999
--param iv-consider-all-candidates-bound=99999 --param iv-always-prune-cand-set-bound=999999 -fno-
ira-loop-pressure -fsched-spec-load -fsched-spec-load-dangerous -fmodulo-sched -fmodulo-sched-
allow-regmoves -fuse-load-updates -fno-builtin-strcpy -falign-functions=16 -fmerge-all-constants -
static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppce5500-fsl-linux


For IPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=e5500 -m32 -O3 -flto -ffat-lto-objects -fwhole-program -falign-loops=16 -
static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppce5500-fsl-linux
```

```
LINKER_FLAGS   =  -lm -m32 -mcpu=e5500 -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppce5500-fsl-
linux
SIZE   = $(TOOLS)/powerpc-fsl-linux-size


E5500 64bit:

TOOLS = /opt/fsl-qoriq/1.9/sysroots/x86_64-fslsdk-linux/usr/bin/powerpc64-fsl-linux
CC             = $(TOOLS)/powerpc64-fsl-linux-gcc
AS              = $(TOOLS)/powerpc64-fsl-linux-as
LD              = $(TOOLS)/powerpc64-fsl-linux-gcc
AR              = $(TOOLS)/powerpc64-fsl-linux-ar

For TCPMark, use following compiler flags:
COMPILER_FLAGS = -m64 -mcpu=e5500 -mcmodel=small -O3 -falign-loops=4 -fivopts -fipa-pta -funroll-
all-loops --param max-inline-insns-auto=1000  --param case-values-threshold=30  --param max-
average-unrolled-insns=99999999 -falign-loops=4 -fivopts -fipa-pta -funroll-all-loops --param max-
inline-insns-auto=1000  --param case-values-threshold=30  --param max-average-unrolled-
insns=99999999 -fdisable-c11-self-mod-expr --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppc64e5500-fsl-
linux -static

For IPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=e5500 -m64 -mcmodel=small -O3 -flto -ffat-lto-objects -fwhole-program -
fbranch-target-load-optimize -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppc64e5500-fsl-linux



LINKER_FLAGS   =  -lm -m64 -mcpu=e5500 -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppc64e5500-
fsl-linux
SIZE = $(TOOLS)/powerpc64-fsl-linux-size


E6500 32bit:

TOOLS = /opt/fsl-qoriq/1.9/sysroots/x86_64-fslsdk-linux/usr/bin/powerpc-fsl-linux/
CC = $(TOOLS)/powerpc-fsl-linux-gcc
AS = $(TOOLS)/powerpc-fsl-linux-as
LD = $(TOOLS)/powerpc-fsl-linux-gcc
AR = $(TOOLS)/powerpc-fsl-linux-ar

For TCPMark, use following compiler flags;
COMPILER_FLAGS = -mcpu=e6500 -m32 -O3 -funroll-all-loops -ftree-loop-ivcanon -fvariable-expansion-
in-unroller -finline-limit=40 -fipa-pta -ftracer -fno-auto-inc-dec -static --sysroot=/opt/fsl-
qoriq/1.9/sysroots/ppce6500-fsl-linux

For IPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=e6500 -m32 -O3 -flto -ffat-lto-objects -fwhole-program -fipa-struct-reorg -
fipa-matrix-reorg -falign-jumps=16 -fno-builtin-strcpy -fmodulo-sched -fmodulo-sched -fmodulo-
sched-allow-regmoves -fno-ira-loop-pressure -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/
ppce6500-fsl-linux

LINKER_FLAGS   =  -lm -m32 -mcpu=e6500 -static -maltivec --sysroot=/opt/fsl-qoriq/1.9/sysroots/
ppce6500-fsl-linux
SIZE   = $(TOOLS)/powerpc-fsl-linux-size

E6500 64bit:
TOOLS = /opt/fsl-qoriq/1.9/sysroots/x86_64-fslsdk-linux/usr/bin/powerpc64-fsl-linux
CC             = $(TOOLS)/powerpc64-fsl-linux-gcc
AS              = $(TOOLS)/powerpc64-fsl-linux-as
LD              = $(TOOLS)/powerpc64-fsl-linux-gcc
```

```
AR              = $(TOOLS)/powerpc64-fsl-linux-ar

For TCPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=e6500 -m64 -mcmodel=small -O3 -flto -ffat-lto-objects -fwhole-program -
finline-limit=40 -fgcse-sm -fgcse-las  -falign-loops=9 -fwiden-types -mslow-mfocr -frename-
registers -finline-limit=40 -fsection-anchors -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/
ppc64e6500-fsl-linux

For IPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=e6500 -m64 -mcmodel=small -O3 -flto -ffat-lto-objects -fwhole-program -
ftree-loop-distribution -fmodulo-sched -fsched-spec-load -fsched-spec-load-dangerous -fno-ira-
loop-pressure -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppc64e6500-fsl-linux

LINKER_FLAGS   =  -lm -m64 -mcpu=e6500 --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppc64e6500-fsl-linux

SIZE   = $(TOOLS)/powerpc64-fsl-linux-size
```

Generate EEMBC binary for target board

1. Create a working directory.
2. Retrieve EEMBC v2.0 from www.eembc.org
3. Extract the EEMBC v2.0 source code
4. Edit the util/make/gcc.mak so that the CC variable points to the correct location of your compiler. See above table for detailed compiler flags and configuration
5. Build binary using the make command:

   ```
   make VER=v2 TOOLCHAIN=gcc THLITE=_lite all-lite
   ```

6. After the build is complete, copy binary files under `<EEMBC_2.0_INSTALL_DIR>/networking/gcc/bin_lite` to target board

12.1.3.2 Test Procedure

Test procedure for lmbench.

Running test and result collection

Deploy target board with corresponding software mentioned in previous section.

Put EEMBC Netmark binary compiled with optimized flags mentioned in section test environment on target board. EEMBC Netmark binary file list are as followings:

```
networking/tcpbulk_lite.exe
networking/tcpjumbo_lite.exe
networking/tcpmixed_lite.exe
networking/ip_pktcheckb1m_lite.exe
networking/ip_pktcheckb2m_lite.exe
networking/ip_pktcheckb4m_lite.exe
networking/ip_pktcheckb512k_lite.exe
networking/ip_reassembly_lite.exe
networking/ip_reassembly_lite.exe
networking/nat_lite.exe -INITTIME
networking/nat_lite.exe
networking/ospfv2_lite.exe
networking/qos_lite.exe
networking/routelookup_lite.exe
```

Run the benchmark

```
networking/tcpbulk_lite.exe -i 200000
networking/tcpjumbo_lite.exe -i 300000
networking/tcpmixed_lite.exe -i 100000
networking/ip_pktcheckb1m_lite.exe -i 50000
networking/ip_pktcheckb2m_lite.exe -i 30000
networking/ip_pktcheckb4m_lite.exe -i 10000
networking/ip_pktcheckb512k_lite.exe -i 100000
networking/ip_reassembly_lite.exe -INITTIME -i 5000
networking/ip_reassembly_lite.exe -i 5000
networking/nat_lite.exe -INITTIME -i 10000
networking/nat_lite.exe -i 10000
networking/ospfv2_lite.exe -i 10000
networking/qos_lite.exe -i 300
networking/routelookup_lite.exe -i 20000
```

The result will display as followings:

```
root@t4240qds:netv2_e6500_O2_bin# ./tcpbulk_lite -i 167000
 Configure benchmark for bulk data transfer test
Initialize network buffer pools
INFO: Initializing client and server NIF
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 167000
>> Bench Mark               : TCP-BM bulk V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 167000
--  Target Duration   = 1600000
--  Target Timer Rate = 1000000
--  v1                = -4280
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =   104375.000
--  Total Run Time    =        1.600sec
--  Time / Iter       =        0.000009581sec
>> DONE!
>> BM: TCP-BM bulk V2.0R1
>> ID: NTW tcp

root@t4240qds:netv2_e6500_O2_bin# ./tcpjumbo_lite -i 250500
 Configure benchmark for jumbo packet transfer test
Initialize network buffer pools
INFO: Initializing client and server NIF
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
```

```
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 250500
>> Bench Mark               : TCP-BM jumbo V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 250500
--  Target Duration   = 1540000
--  Target Timer Rate = 1000000
--  v1                = -24000
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =   162662.338
--  Total Run Time    =        1.540sec
--  Time / Iter       =        0.000006148sec
>> DONE!
>> BM: TCP-BM jumbo V2.0R1
>> ID: NTW tcp

root@t4240qds:netv2_e6500_O2_bin# ./tcpmixed_lite -i 83500
 Configure benchmark for mixed packet size test
Initialize network buffer pools
INFO: Initializing client and server NIF
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 83500
>> Bench Mark               : TCP-BM mixed V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 83500
--  Target Duration   = 1940000
--  Target Timer Rate = 1000000
--  v1                = -2736
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =    43041.237
--  Total Run Time    =        1.940sec
--  Time / Iter       =        0.000023234sec
>> DONE!
>> BM: TCP-BM mixed V2.0R1
>> ID: NTW tcp


root@t4240qds:netv2_e6500_O2_bin# ./ip_pktcheckb1m_lite -i 41750
>> Datagram buffer size          : 0x0100000
>> Datagram alignment            :         4
>> Descriptor padd size          :         8
>> Number of Datagrams allocated :       720
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
```

```
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 41750
>> Bench Mark               : Networking: IP Packet Check Benchmark, 1.0M V2.0R1
--  Non-Intrusive CRC = e3b5
--  Iterations        = 41750
--  Target Duration   = 2260000
--  Target Timer Rate = 1000000
--  v1                = 30060000
--  v2                = 1670000
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =    18473.451
--  Total Run Time    =        2.260sec
--  Time / Iter       =        0.000054132sec
>> DONE!
>> BM: Networking: IP Packet Check Benchmark, 1.0M V2.0R1
>> ID: NTW ip_pktchec

root@t4240qds:netv2_e6500_O2_bin# ./ip_pktcheckb2m_lite -i 25050
>> Datagram buffer size          : 0x0200000
>> Datagram alignment            :         4
>> Descriptor padd size          :         8
>> Number of Datagrams allocated :      1412
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 25050
>> Bench Mark               : Networking: IP Packet Check Benchmark, 2.0M V2.0R1
--  Non-Intrusive CRC =  48b
--  Iterations        = 25050
--  Target Duration   = 2720000
--  Target Timer Rate = 1000000
--  v1                = 35370600
--  v2                = 1828650
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     9209.559
--  Total Run Time    =        2.720sec
--  Time / Iter       =        0.000108583sec
>> DONE!
>> BM: Networking: IP Packet Check Benchmark, 2.0M V2.0R1
>> ID: NTW ip_pktchec


root@t4240qds:netv2_e6500_O2_bin# ./ip_pktcheckb4m_lite -i 8350
>> Datagram buffer size          : 0x0400000
>> Datagram alignment            :         4
>> Descriptor padd size          :         8
>> Number of Datagrams allocated :      2824
```

```
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 8350
>> Bench Mark               : Networking: IP Packet Check Benchmark, 4.0M V2.0R1
--  Non-Intrusive CRC = d86c
--  Iterations        = 8350
--  Target Duration   = 1800000
--  Target Timer Rate = 1000000
--  v1                = 23580400
--  v2                = 1244150
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     4638.889
--  Total Run Time    =        1.800sec
--  Time / Iter       =        0.000215569sec
>> DONE!
>> BM: Networking: IP Packet Check Benchmark, 4.0M V2.0R1
>> ID: NTW ip_pktchec

root@t4240qds:netv2_e6500_O2_bin# ./ip_pktcheckb512k_lite -i 83500
>> Datagram buffer size          : 0x0080000
>> Datagram alignment            :         4
>> Descriptor padd size          :         8
>> Number of Datagrams allocated :       374
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 83500
>> Bench Mark               : Networking: IP Packet Check Benchmark, 0.5M V2.0R1
--  Non-Intrusive CRC = 3e1d
--  Iterations        = 83500
--  Target Duration   = 2160000
--  Target Timer Rate = 1000000
--  v1                = 31229000
--  v2                = 1753500
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =    38657.407
--  Total Run Time    =        2.160sec
--  Time / Iter       =        0.000025868sec
>> DONE!
>> BM: Networking: IP Packet Check Benchmark, 0.5M V2.0R1
>> ID: NTW ip_pktchec


root@t4240qds:netv2_e6500_O2_bin# ./ip_reassembly_lite -INITTIME -i 4175
*** Initialization Timing Run, Subtract from normal run for score ***
```

```
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 4175
>> Bench Mark               : INITIALIZATION Networking: IP Reassembly Benchmark V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 4175
--  Target Duration   = 640000
--  Target Timer Rate = 1000000
--  v1                = 200
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     6523.438
--  Total Run Time    =        0.640sec
--  Time / Iter       =        0.000153293sec
>> DONE!
>> BM: INITIALIZATION Networking: IP Reassembly Benchmark V2.0R1
>> ID: NTW ip_reasmIT

root@t4240qds:netv2_e6500_O2_bin# ./ip_reassembly_lite -i 4175
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 4175
>> Bench Mark               : Networking: IP Reassembly Benchmark V2.0R1
--  Non-Intrusive CRC = 678e
--  Iterations        = 4175
--  Target Duration   = 1940000
--  Target Timer Rate = 1000000
--  v1                = 200
--  v2                = 4939025
--  v3                = 835000
--  v4                = 0
--  Iterations/Sec    =     2152.062
--  Total Run Time    =        1.940sec
--  Time / Iter       =        0.000464671sec
>> DONE!
>> BM: Networking: IP Reassembly Benchmark V2.0R1
>> ID: NTW ip_reasm


root@t4240qds:netv2_e6500_O2_bin# ./nat_lite -INITTIME -i 8350
*** Initialization Timing Run, Subtract from normal run for score ***
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
```

```
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 8350
>> Bench Mark               : INITIALIZATION Network Address Translation V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 8350
--  Target Duration   = 960000
--  Target Timer Rate = 1000000
--  v1                = 1000
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     8697.917
--  Total Run Time    =        0.960sec
--  Time / Iter       =        0.000114970sec
>> DONE!
>> BM: INITIALIZATION Network Address Translation V2.0R1
>> ID: NTW NATIT

root@t4240qds:netv2_e6500_O2_bin# ./nat_lite -i 8350
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 8350
>> Bench Mark               : Network Address Translation V2.0R1
--  Non-Intrusive CRC = 9d46
--  Iterations        = 8350
--  Target Duration   = 2320000
--  Target Timer Rate = 1000000
--  v1                = 1000
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     3599.138
--  Total Run Time    =        2.320sec
--  Time / Iter       =        0.000277844sec
>> DONE!
>> BM: Network Address Translation V2.0R1
>> ID: NTW NAT


root@t4240qds:netv2_e6500_O2_bin# ./ospfv2_lite -i 8350
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
```

```
>> Recommended Iterations   : 8350
>> Bench Mark               : Networking: OSPF Benchmark V2.0R1
--  Non-Intrusive CRC = 7f12
--  Iterations        = 8350
--  Target Duration   = 1480000
--  Target Timer Rate = 1000000
--  v1                = 400
--  v2                = 4
--  v3                = 8
--  v4                = 32000
--  Iterations/Sec    =     5641.892
--  Total Run Time    =        1.480sec
--  Time / Iter       =        0.000177246sec
>> DONE!
>> BM: Networking: OSPF Benchmark V2.0R1
>> ID: NTW ospf


root@t4240qds:netv2_e6500_O2_bin# ./qos_lite -i 250
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 250
>> Bench Mark               : Networking: QoS V2.0R1
--  Non-Intrusive CRC = fa81
--  Iterations        = 250
--  Target Duration   = 1000000
--  Target Timer Rate = 1000000
--  v1                = 100
--  v2                = 100
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =      250.000
--  Total Run Time    =        1.000sec
--  Time / Iter       =        0.004000000sec
>> DONE!
>> BM: Networking: QoS V2.0R1
>> ID: NTW QoS


root@t4240qds:netv2_e6500_O2_bin# ./routelookup_lite -i 16700
>> self-check completed ok.
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 16700
>> Bench Mark               : Networking: Route Lookup Benchmark V2.0R1
--  Non-Intrusive CRC = 407d
```

```
--  Iterations        = 16700
--  Target Duration   = 2100000
--  Target Timer Rate = 1000000
--  v1                = 0
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     7952.381
--  Total Run Time    =        2.100sec
--  Time / Iter       =        0.000125749sec
>> DONE!
>> BM: Networking: Route Lookup Benchmark V2.0R1
>> ID: NTW routelookup EEMBC
```

NOTE

For other platforms, use the ratio (CPU_Freq_target_platform/2000) to get the corresponding parameter. For example, if P4080DS's CPU freq is 1500MHz, then use 0.75 to multiply each parameter for above workloads.

There is a run to run variation in. So an average across 5 runs was taken for every result

Networking Version 2.0 Calculation

Networking Version 2.0 produces two aggregate "mark" scores: the TCPmark™ and the IPmark™

The IPmark is intended for developers of infrastructure equipment, while the TCPmark, which includes the TCP benchmark, focuses on client- and server-based network hardware.

The IPmark is the geometric mean of the scores for QoS, Route Lookup, OSPF, IP Reassembly, Network Address Translation, and the geometric mean of the individual scores for IP Packet check, all divided by 10:

```
IPmark = Geomean ((Geomean (IP Packet Check [0.5MB], IP Packet Check [1MB], IP Packet Check
[2MB], IP Packet Check [4MB]), QoS, Route Lookup, OSPF, IP Reassembly, NAT))/10
```

The TCPmark is the geometric mean of the scores for TCP Jumbo, TCP Bulk, and TCP Mixed divided by 100:

```
TCPmark = Geomean (TCP jumbo, TCP bulk, TCP Mixed)/100
```

To calculate a geometric mean, multiply all the results of the tests together and take the nth root of the product, where n equals the number of tests.

NAT and IP Reassembly Benchmarks

To calculate the iterations per second for the NAT and IP Reassembly benchmarks, it's necessary to run the benchmarks twice:

1. Run the benchmark with the flag -INITTIME supplied on the command line.
2. Run the benchmark without the flag -INITTIME supplied on the command line. The same executable must be run both times and the number of iterations must be identical.
3. Subtract the time of the first run from the time of the second run and calculate the iterations per second based on the calculated time.

The score reported for each device is a single-number figure of merit calculated by taking the geometric mean of the individual Networking scores and dividing by 395.184 . (This normalization factor is derived from the lowest score in this category on December 5, 2000.) Scores for each of the individual benchmarks within this suite allow designers to weight and aggregate the benchmarks to suit specific application requirements.

To calculate a geometric mean, multiply all the results (*) of the tests together and take the nth root of the product, where n equals the number of tests.

(*) Scores included in geometric mean:

```
•OSPF
•Route Lookup
•Packet Flow - 512 kbytes
•Packet Flow - 1 Mbyte
•Packet Flow - 2 Mbytes
```

NOTE

This calculation can also be found on EEMBC website:

```
http://eembc.org/benchmark/reports/mark.php?suite=NT2
```

12.1.4 LMBench

How to setup a board to baseline LMBench performance for QorIQ processor development boards.

12.1.4.1 Test Environment

Objectives

The first objective is to baseline the lmbench performance on QorIQ platform. Next identify any optimizations that have been discovered and ensure they are implemented on the those platforms. Investigate other changes that may improve performance. Compare these results with other NXP and competitors products. Provide assistance to any customer activities that could result in design wins. Review the results with. Finally post the results and this guild so that groups within NXP may share them with customers.

Hardware Platform Identification

Table 597. Target Platform

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR)	Core Type
P2040RDB	Rev2.0	1200/600/1200	e500mc
P3041DS	Rev2.0	1500/750/1333	e500mc
P4080DS	Rev3.0	1500/800/1300	e500mc
P5020DS	Rev2.0	2000/800/1333	e5500
P5040DS	Rev2.1	2266/800/1600	e5500
T4240QDS	Rev2.0	1800/733/1867	e6500
T4240RDB	Rev2.0	1800/733/1867	e6500
B4860QDS	Rev2.2	1600/667/1867	e6500
T2080QDS	Rev1.1	1800/600/1867	e6500
T2080RDB	Rev1.1	1800/600/1866	e6500
T1040D4RDB	Rev1.1	1400/600/1600	e5500
T1024RDB	Rev1.0	1400/400/1600	e5500
T1023RDB	Rev1.0	1200/400/1600	e5500
LS1021ATWR	Rev2.0	1000/300/1600	cortex A7

Table continues on the next page...

Table 597. Target Platform (continued)

Board	Silicon Revision	Default Freqeuncy(Core/CCB/ DDR)	Core Type
LS1043A	Rev1.0	1600/400/1600	cortex A53
LS2080A	Rev1.0	1867/533/1867	cortex A57

NOTE

For board switch settings, please see the corresponding reference manual.

Software Platform Identification

All software was built from QorIQ SDK ISO.

Boot Loader

U-boot 2016.01 with NXP-specific patches on top; Configuration for different platform like as following

For P2040/P2041/P3041, enable chip select interleaving and address hash, disabling ecc by hwconfig with syntax:

```
"hwconfig=fsl_ddr:bank_intlv=cs0_cs1,addr_hash=true,ecc=off"
```

For P4080/P5020/P5040, enable cacheline interleaving, chip select interleaving and address hash, disabling ecc by hwconfig with syntax:

```
 "hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1,ecc=off,addr_hash=true"
```

For T4240/B4860/B4420/T1040/T2080, keep default configuration for DDR controller.

Lmbench Application

The roofs includes LMBench binaries which were built without optimization. For general latency performance, the default lmbench binary file in root file system will be ok. To get better bandwidth performance result, we need to modify compiler flags to enable "O3" optimization.

Followings are steps with assumption that SDK ISO had been installed and corresponding platform Yocto project is ready. Meanwhile, take p4080ds as one example in this document.

Get lmbench package with Yocto:

```
bitbake lmbench -c patch
```

Modify the Lmbench compiler flags in lmbench bitbake configuration file

```
find lmbench_3.0-a9.bb under directory {Yocto_install_dir}/{build_project}/meta-oe/meta-
oe/recipes-benchmark/lmbench
Append "-O3" in CC of line 24 in lmbench_3.0-a9.bb:

EXTRA_OEMAKE = 'CC="${CC} -O3" AR="${AR}" RANLIB="${RANLIB}" CFLAGS="${CFLAGS}" \
                LDFLAGS="${LDFLAGS}" LD="${LD}" OS="${TARGET_SYS}" \
```

```
For e6500-32bit:
Append "-m32 -O3 -fno-tree-vectorize " in CC of line 24 in lmbench_3.0-a9.bb:

EXTRA_OEMAKE = 'CC="${CC} -m32 -O3 -fno-tree-vectorize" AR="${AR}" RANLIB="${RANLIB}"
CFLAGS="${CFLAGS}" \
                LDFLAGS="${LDFLAGS}" LD="${LD}" OS="${TARGET_SYS}" \
For e6500-64bit:
Append "-m64 -O3 -fno-tree-vectorize " in CFLAGS of line 24 in lmbench_3.0-a9.bb:

EXTRA_OEMAKE = 'CC="${CC} -m64 -O3 -fno-tree-vectorize" AR="${AR}" RANLIB="${RANLIB}"
```

```
CFLAGS="${CFLAGS}" \
                LDFLAGS="${LDFLAGS}" LD="${LD}" OS="${TARGET_SYS}" \
```

Clean previous lmbench binary file without optimization flags.

```
rm -rf {Yocto_install_dir}/build_p4080ds_release/tmp/work/ppce500mc-fsl_networking-linux/
lmbench/3.0-a9-r2/lmbench-3.0-a9/bin/powerpc-fsl_networking-linux
```

Rebuild lmbench with optimization

```
bitbake -c compile -f lmbench
```

Get new lmbench binary files

1. For those only care bandwidth customer, they can pickup bandwidth benchark application from lmbench package directly under following folder:

   ```
   {Yocto_install_dir}/build_p4080ds_release/tmp/work/ppce500mc-fsl_networking-linux/
   lmbench/3.0-a9-r2/lmbench-3.0-a9/bin/powerpc-fsl_networking-linux/bw_mem
   ```

2. or they can deploy new lmbench binary files into rootfs

   ```
   bitbake fsl-image-core
   ```

12.1.4.2 Test Procedure

Test procedure for lmbench.

Running test and result collection

As described in section 2.2.2.1 and in Boot loader, separate u-boot image was burnt in alternate u-boot flash bank and the DUT is booted out of that bank.

Binaries compiled with different flags were run separately and then the data was collected for the binary showing the best results

There is a run to run variation in. So an average across 5 runs was taken for every result

Scripts for lmbench test.

Execution latency test script:

```
for i in `seq 1 5`
do
echo “Integer, integer64,float,double float execution latency”
lat_ops
done
```

Memory read latency test script:

```
for i in `seq 1 5`
do
echo “L1, L2, L3 and DDR read latency”
lat_mem_rd 100M
done
```

Memory bandwidth test script:

```
#!/bin/sh

for opt in rd wr rdwr cp frd fwr fcp bzero bcopy
do
   echo "L1 cache bandwidth $opt test with #$proc process"
   for idx in `seq 1 5`
   do
#8k is fit for all platform
       bw_mem -P 1 8k $opt
   done
       echo "L2 cache bandwidth $opt test"
#no L2 cache on P2040RDB, so it doesn’t need this.
# For different platform, L2 cache size is different. So it need to modify the data size.
```

```
# Here 40k is fit for P2041/ P3041/P4080/ /P5020/P5040
   for idx in `seq 1 5`
   do
       bw_mem -P 1 40k $opt
   done
       echo "L3 cache bandwidth $opt test"
# on dpaa platform, the minimum L3 cache size is 1M, so choose 750k for L3 cache test data
size
  for idx in `seq 1 5`
   do
       bw_mem -P 1 750k $opt
   done
       echo "Main mem bandwidth $opt test"
  for idx in `seq 1 5`
   do
       bw_mem -P 1 100m $opt
   done

done
```

12.2 Linux Networking and Storage

Linux Networking and Storage - Benchmark Reproducibility Guides

12.2.1 IPv4 Forward - DPAA

12.2.1.1 Linux DPAA IPv4

12.2.1.1.1 Test Environment

Benchmarking Objectives

The purpose of this guide is to show how to setup a Linux IPv4 Forwarding system on a NXP QorIQ DPAA platform and how to measure its performance.

NOTE

P4080DS was used for the demo. Please refer to the board's manual for other boards.

Platform Configurations

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/FMan	SerDes Protocol	Config. Name
P2040RDB	Rev B	Rev 2.0	• CPU - 1200 MHz • CCB - 600 MHz • DDR - 600 MHz • FMAN - 500 MHz	0x19	RR_PH_0x19
P3041DS	Rev X1	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 666.667 MHz • FMAN - 583.333 MHz	0x36	RR_HXAPNSP_0x36

Table continues on the next page...

Table continued from the previous page...

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/FMan	SerDes Protocol	Config. Name
P4080DS	Rev X3	Rev 3.0	• CPU - 1499.985 MHz • CCB - 799.992 MHz • DDR - 649.994 MHz • FMAN - 599.994 MHz	0xe	R_PPSXX_0xe
P5020DS	Rev X7	Rev 2.0	• CPU - 2000 MHz • CCB - 800 MHz • DDR - 666.667 MHz • FMAN - 600 MHz	0x36	RR_HXAPNSP_0x36
P5040DS	Rev X7	Rev 2.1	• CPU - 2266.667 MHz • CCB - 800 MHz • DDR - 800 MHz • FMAN - 600 MHz	0x02	RR_XXSNSpP_0x02
B4860QDS	Rev B2	Rev 2.2	• CPU - 1600 MHz • CCB - 666.667 MHz • DDR - 933.333 MHz • FMAN - 666.667 MHz	0x2A_0x8D	N_RSSS_0x2A_0x8D
T4240QDS	Rev X4	Rev 2.0	• CPU - 1800 MHz • CCB - 733.333 MHz • DDR - 933.333 MHz • FMAN - 733.333 MHz	1_1_5_5	RR_XXXXPRPR_1_1_5_5
T4240RDB	Rev D	Rev 2.0	• CPU - 1800 MHz • CCB - 733.333 MHz • DDR - 933.333 MHz • FMAN - 733.333 MHz	27_55_1_9	SSFFPPH_27_55_1_9
T2080QDS	Rev X5	Rev 1.1	• CPU - 1800 MHz • CCB - 600 MHz • DDR - 933.333 MHz • FMAN - 700 MHz	66_15	RR_PNNPPH_66_15

Table continues on the next page...

Table continued from the previous page...

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/FMan	SerDes Protocol	Config. Name
T2080RDB	Rev C	Rev1.1	• CPU - 1799.820 MHz • CCB - 600 MHz • DDR - 933.333 MHz • FMAN - 700 MHz	66_15	RRFFXX_P_66_15
T1023RDB	Rev B	Rev 1.0	• CPU - 1200 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 600 MHz	0x77	RNSS_NPP_77
T1042D4RDB	Rev A	Rev 1.1	• CPU - 1400 MHz • CCB - 600 MHz • DDR - 800 MHz • FMan - 600 MHz	0x86	RR_P_86

Network Simulator

Spirent TestCenter Performance Analysis System

- Spirent Testcenter version: 4.41
- Choose Bound Stream Block for Traffic generator
- Choose Wizard -> Custom Test -> Throughput from Spirent Testcenter to create command sequencer for throughput test
- Testcenter configuration:
  - Test Duration: 60 Seconds
  - Resolution: 0.1%
  - Packet Loss Rate: 0.001 %
  - Packet Size: 64, 128, 256, 390, 512, 1024, 1280, 1518 bytes

Software Platform Identification

All software was built from SDK ISO. Please refer to following images:

Table 598. Images

Target Development System Software	Image file name/Description
FMan ucode	Please refer to SDK document to use correct ucode version

Table continues on the next page...

Table 598. Images (continued)

Target Development System Software	Image file name/Description
RCW	P2041RDB: RR_PH_0x19/rcw_5g_1200mhz.bin P3041DS: RR_HXAPNSP_0x36/rcw_15g_1500mhz.bin P4080DS: R_PPSXX_0xe/rcw_2sgmii_1500mhz_rev3.bin P5020DS: RR_HXAPNSP_0x36/rcw_15g_2000mhz.bin P5040DS: RR_XXSNSpP_0x02/rcw_26g_2267mhz.bin B4860QDS: N_RSSS_0x2A_0x8D/ rcw_4sgmii_4srio_2xfi_1600mhz_1866ddr_rev2.bin T4240QDS: RR_XXXXPRPR_1_1_5_5/rcw_1_1_5_5_1666MHz_rev2.bin T4240RDB: SSFFPPH_27_55_1_9/rcw_27_55_1_9_1666MHz.bin T2080QDS: RR_PNNPPH_66_15/rcw_66_15_1800MHz_rev11.bin T2080RDB: RRFFXX_P_66_15/rcw_66_15_1800MHz_rev11.bin T1023RDB: RNSS_NPP_77/rcw_77_1400MHz.rcw T1042D4RDB: RR_P_86/rcw_1400MHz.bin
Boot loader	u-boot-${PLATFORM}.bin
Linux Kernel	uImage-${PLATFORM}.bin
Device Tree	uImage-${PLATFORM}.dtb
File system	fsl-image-core-${PLATFORM}.ext2.gz.u-boot

NOTE

Target Development System Software

- **Boot loader**
  - U-boot 2016.01 with default ddr configurations:
  - For example, on P4080DS with two DDR controllers
    - Enable chip select interleaving mode: CS0+CS1
    - Enable DDR Controller interleaving mode: cache line (If there are two DDR controllers)
- **FMAN ucode**

  The SDK ISO contains the default ucode versions for all the supported platforms. Please refer to the “Linux Kernel and Device Drivers” section from the “Components” documentation chapter.
- **Operating system**

  Default kernel in ISO
- **Additional root filesystem considerations**

  The tests procedure for IPv4 forwarding makes use of the "arp" and "ifconfig" utilities. These are part of the net-tools package, which the root filesystem stored in the hard disk should have.

12.2.1.1.2 Test Procedure

This section describes the test procedure for various configurations of IPv4 forwarding

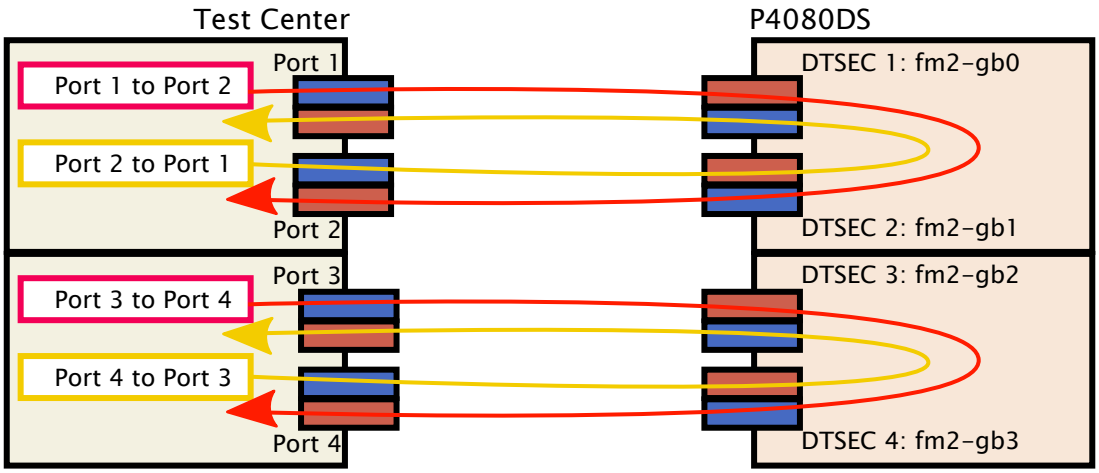
Test configuration

Different platforms have different Ethernet interfaces, please refer to SDK documentation (Selecting Ethernet interfaces) to configure Ethernet interfaces for Linux IPFWD.

At this point, we refer to P4080DS platform.

P4080DS have five 1Gb Ethernet interfaces(with 0x10 serdes protocol) and 2 10Gb Ethernet interfaces(with 0xe serdes protocol). fm1-gb1, fm2-gb0, fm2-gb1, fm2-gb2, fm2-gb3, fm1-10g, fm2-10g.

1. **4x1Gb Ethernet Ipv4 Forward 1024 flows(bi-directional) configuration:**
   - Spirent Testcenter configuration:
     - Port 1 to Port 2: Src: 192.168.1.2 – 192.168.1.17 -> Dst: 192.168.2.2 – 192.168.2.17
     - Port 2 to Port 1: Src: 192.168.2.2 – 192.168.2.17 -> Src: 192.168.1.2 – 192.168.1.17
     - Port 3 to Port 4: Src: 192.168.3.2 – 192.168.3.17 -> Dst: 192.168.4.2 – 192.168.4.17
     - Port 4 to Port 3: Src: 192.168.4.2 – 192.168.4.17 -> Src: 192.168.3.2 – 192.168.3.17



   - Linux configuration:

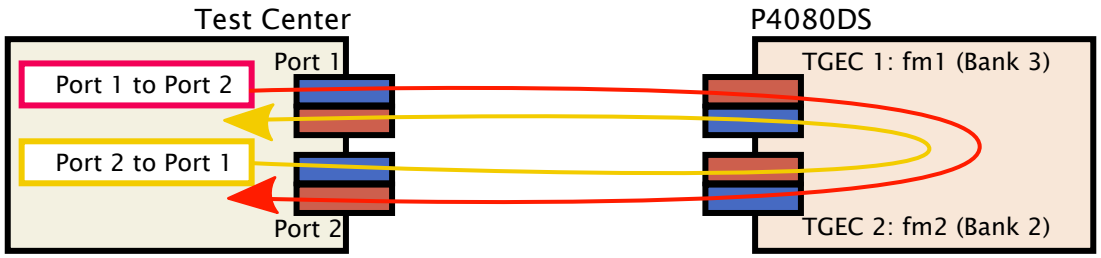
```
cat > do_arp.sh << EOF
#!/bin/sh

i=1
ifconfig fm2-gb0 192.168.1.1 netmask 255.255.255.0
ifconfig fm2-gb1 192.168.2.1 netmask 255.255.255.0
ifconfig fm2-gb2 192.168.3.1 netmask 255.255.255.0
ifconfig fm2-gb3 192.168.4.1 netmask 255.255.255.0
echo 1 > /proc/sys/net/ipv4/ip_forward

while expr \$i \< 17
do
ip=\`expr \$i + 1\`
hex=\`printf "%x" \$i\`
arp -s 192.168.1.\$ip 00:10:91:00:00:02
arp -s 192.168.2.\$ip 00:10:92:00:00:02
arp -s 192.168.3.\$ip 00:10:93:00:00:02
arp -s 192.168.4.\$ip 00:10:94:00:00:02
i=\`expr \$i + 1\`
```

```
done
EOF
#chmod +x do_arp.sh
#./do_arp.sh
#fmc -c config.xml -p policy_ipv4.xml -a
```

2. **2x10Gb Ethernet IPv4 Forward configuration:**

   1024 flows(bi-directional) between 2x10G Ethernet interfaces:

   - Spirent Testcenter configuration:
     - Port 1 -> Port 2 :
       - Src: 192.168.2.2 – 192.168.1.23
       - Dst: 192.168.3.2 – 192.168.2.24
     - Port 2 -> Port 1 :
       - Src: 192.168.3.2 – 192.168.1.24
       - Dst: 192.168.2.2 – 192.168.2.23



   - Linux configuration:

```
cat > do_arp.sh << EOF
#!/bin/sh

i=1
ifconfig fm1-10g 192.168.2.1 netmask 255.255.255.0
ifconfig fm2-10g 192.168.3.1 netmask 255.255.255.0
echo 1 > /proc/sys/net/ipv4/ip_forward

while expr \$i \< 23
do
ip=\`expr \$i + 1\`
hex=\`printf "%x" \$i\`
arp -s 192.168.3.\$ip 00:10:94:00:00:01
arp -s 192.168.2.\$ip 00:10:95:00:00:01
i=\`expr \$i + 1\`
done
arp -s 192.168.3.24 00:10:94:00:00:01
EOF
#chmod +x ./do_arp.sh
#./do_arp.sh
#fmc -c config.xml -p policy_ipv4.xml -a
```

Running the test

After Spirent Testcenter application is completely configured, the IPv4 automated forward throughput measurement can be started.

Appendix

In order to get better Linux IPFWD performance on P2040RDB/P3041DS/P4080DS/P5020DS/P5040DS which are FManv1, the PCD configuration must be applied before starting the Linux IPFWD performance test (by running the fmc tool as seen in the commented lines above).

Here is one example for P4080DS:

FMC configuration files for optimization of Linux IPv4 forward can be found in /etc/fmc/config in rootfs. In this directory, there are specific configuration and policy files for each supported RCW. Using the RCW, numerous platform parameters can be configured, including the SERDES protocols used by the P4080DS board. For the 4G traffic mode, the config.xml should have the 4 ports configuration entries:

```
<cfgdata>
   <config>
      <engine name="fm1">
         <port type="MAC" number="1" policy="fm_policy_9"/>
         <port type="MAC" number="2" policy="fm_policy_10"/>
         <port type="MAC" number="3" policy="fm_policy_11"/>
         <port type="MAC" number="4" policy="fm_policy_12"/>
      </engine>
   </config>
</cfgdata>
```

For the 20G traffic mode, change config.xml to have the 2x10G ports configuration entries:

```
<cfgdata>
   <config>
      <engine name="fm0">
         <port type="MAC" number="9" policy="fm_policy_7"/>
      </engine>
      <engine name="fm1">
         <port type="MAC" number="9" policy="fm_policy_15"/>
      </engine>
   </config>
</cfgdata>
```

For the other platforms, use the existing xml files from /etc/fmc/config.

12.2.1.2 IPv4 Forward - T1024RDB

12.2.1.2.1 Test Environment

The first objective is to baseline the Linux IPv4 Forward performance. Next identify any optimizations that have been discovered and ensure they are implemented on the QorIQ DPAA platform. Investigate other changes that may improve performance. This guide is intended to provide details for reproducing Linux IPv4 packet forwarding performance results for T1024RDB.

Platform Configurations

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/FMan	SerDes Protocol	Config. Name
T1024RDB	Rev C	Rev 1.0	• CPU - 1400 MHz • CCB - 400 MHz • DDR - 1600 MHz • FMan - 700 MHz	• 0x95 • 0x135	• RRX_PPP_95 • RNS_NPP_135

Network Simulator

Spirent TestCenter Performance Analysis System

- Spirent Testcenter version: 4.41
- Choose Bound Stream Block for Traffic generator
- Choose Wizard -> Benchmarking -> RFC2544 -> Throughput Test -> Sequencer sequencer for throughput test
- Testcenter configuration:
  - Test Duration: 60 Seconds
  - Resolution: 0.1%
  - Packet Loss Rate: 0.001 %
  - Packet Size: 84(ipv6:86), 408,1424(ipv6-aes:1424) bytes

Software Platform Identification

All software was built from SDK ISO. Please refer to following images:

Table 599. Images

Target Development System Software	Image file name/Description
FMan ucode	T1024 rev 1.0 - fsl_fman_ucode_t1024_r1.0_106_4_17.bin(Please refer to SDK document to use correct ucode version)
RCW	T1024RDB: RNS_NPP_135/rcw_135_1400MHz.rcw T1024RDB: RRX_PPP_95/rcw_95_1400MHz.rcw
Boot loader	u-boot-T1024RDB.bin
Linux Kernel	uImage-t1024rdb.bin
Device Tree	uImage-t1024rdb.dtb
Default File system	fsl-image-core-t1024rdb.ext2.gz.u-boot

- **Target Development System Software**
  - **Boot loader**
    - U-boot 2015.01 with default ddr configurations:
    - T1024RDB with one DDR controllers
      - Enable chip select interleaving mode: CS0+CS1
  - **FMAN ucode**

    fsl_fman_ucode_t1024_r1.0_106_4_17.bin.
  - **Operating system**

    Default kernel in ISO
  - **Additional root filesystem considerations**
    - For Linux: uImage-t1024rdb.bin and fsl-image-core-t1024rdb.ext2.gz.u-boot

12.2.1.2.2 Test Procedure

This section describes the test procedure for various configurations of IPv4 forwarding

Test configuration

Different platforms have different Ethernet interfaces, please refer to SDK documentation (Selecting Ethernet interfaces) to configure Ethernet interfaces for Linux IPV4 forward.

T1024RDB have two RCW:

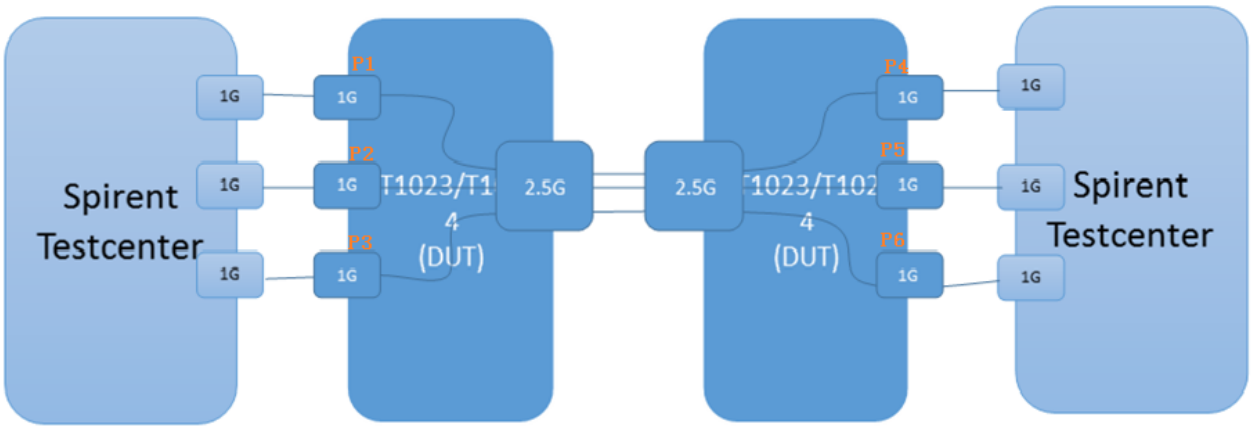
- RCW_135: 1 x 1Gb rgmii ports + 1 x 2.5G SGMII port + 2 x mini_PCIE ports
- RCW_95: 2 x 1Gb rgmii port + 1 x 10G XFI port + 1 x PCIE port + 2 x mini_PCIE ports

The 10G XFI port could speed down to 2.5G. So traffic which will go through DPAA on T1024RDB will use following Ethernet interfaces:

- RCW_135: fm1-mac3(2.5G)/fm1-mac4(RGMII)/eth*(mini_PCIE)/eth*(mini_PCIE)
- RCW_95: fm1-mac1(2.5G)/fm1-mac4(RGMII)/fm1-mac3(RGMII)/eth*(mini_PCIE)

show traffic flow which will go through to DPAA:

-



Note:

- Please use one TC to send traffic, placing two TC in the picture is aimed at showing ports' connection more clear.
- Ports Property:
  - For RCW_135:Port1 and Port4 are RGMII ports.others ports are all mini_PCIE;
  - For RCW_95:Port1/Port2/Port3/Port4 are RGMII ports.others ports are mini_PCIE;

1. 4.1.1 IPv4 Forward 6g flows(bi-directional) configuration:
   - Spirent Testcenter configuration for Linux IPv4 Forward:
     - Port 1 to Port 4 :
       - Src: 192.85.1.2 – 192.85.1.17 -> Dst: 192.85.4.2 –192.85.4.17
       - SMac: 00:10:94:00:00:01 -> DMac: <fm1-mac3(left) mac address>
       - Gateway: 192.85.1.1
     - Port 2 to Port 5 :
       - Src: 192.85.2.2 – 192.85.2.17 -> Dst: 192.85.5.2 – 192.85.5.9
       - SMac: 00:10:94:00:00:02 -> DMac: <fm1-mac3(left) mac address>
       - Gateway: 192.85.2.1
     - Port 3 to Port 6:

- Src: 192.85.3.2 – 192.85.3.9 -> Dst: 192.85.6.2 – 192.85.6.17
- SMac: 00:10:94:00:00:03 -> DMac: <fm1-mac3(left) mac address>
- Gateway: 192.85.3.1

- Port 4 to Port 1:
  - Src: 192.85.4.2 – 192.85.4.17 -> Dst: 192.85.1.2 – 192.85.1.17
  - SMac: 00:10:94:00:00:05 -> DMac: <fm1-mac3(right) mac address>
  - Gateway: 192.85.4.1
- Port 5 to Port 2 :
  - Src: 192.85.5.2 – 192.85.5.9 -> Dst: 192.85.2.2 – 192.85.2.17
  - SMac: 00:10:94:00:00:04 -> DMac: <fm1-mac3(right) mac address>
  - Gateway: 192.85.5.1
- Port 6 to Port 3:
  - Src: 192.85.6.2 – 192.85.6.17 -> Dst: 192.85.3.2 – 192.85.3.8
  - SMac: 00:10:94:00:10:01 -> DMac: <fm1-mac3(right) mac address>
  - Gateway: 192.85.6.1

Running the test

After Spirent Testcenter application is completely configured, the IPv4 automated forward throughput measurement can be started.

Appendix

1. No need to configure PCD when test Linux IPFwd performance.
2. For RCW_135, as there are only 3 1G ports, we need use all of them to send traffic, suggests boot from lan server to run fully-automatic test.
3. Limited by board design, there is only one 1G fman port when 2.5G SGMII works, we only use one fman port to test usdpaa and asf on T1024RDB:2.5G SGMII port.
4. Limited by board design, there is only one 1G fman port when 2.5G SGMII works, for linux IPFwd and IPSec test, we can use 10G PCIe card to test performance of the 2.5G SGMII port.

12.2.1.3 IPv4 Forward - T1040D4RDB

12.2.1.3.1 Test Environment

The first objective is to baseline the Linux IPv4 Forward performance. Next identify any optimizations that have been discovered and ensure they are implemented on the QorIQ DPAA platform. Investigate other changes that may improve performance. This guide is intended to provide details for reproducing Linux IPv4 packet forwarding performance results for T1040D4RDB.

Platform Configurations

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/FMan	SerDes Protocol	Config. Name
T1040D4RDB	Rev A	Rev 1.1	• CPU - 1400 MHz • CCB - 600 MHz • DDR - 1600 MHz • FMan - 600 MHz	0x66	RR_P_66

Network Simulator

Spirent TestCenter Performance Analysis System

- Spirent Testcenter version: 4.41
- Choose Bound Stream Block for Traffic generator
- Choose Wizard -> Benchmarking -> RFC2544 -> Throughput Test -> Sequencer sequencer for throughput test
- Testcenter configuration:
  - Test Duration: 60 Seconds
  - Resolution: 0.1%
  - Packet Loss Rate: 0.001 %
  - Packet Size: 64, 128, 256, 390, 512, 1024, 1280, 1518 bytes

Software Platform Identification

All software was built from SDK ISO. Please refer to following images:

Table 600. Images

Target Development System Software	Image file name/Description
FMan ucode	T1040 rev 1.1 - fsl_fman_ucode_t1040_r1.1_106_4_17.bin(Please refer to SDK document to use correct ucode version)
RCW	T1040D4RDB: RR_P_66/rcw_1400MHz.rcw
Boot loader	u-boot-T1040D4RDB.bin
Linux Kernel	uImage-t1040d4rdb.bin
Device Tree	uImage-t1040d4rdb.dtb
Default File system	fsl-image-core-t1040d4rdb.ext2.gz.u-boot
ASF Kernel	uImage.ppce5500-asf
ASF File system	fsl-image-asf-t1040d4rdb.ext2.gz.u-boot

- **Target Development System Software**
  - **Boot loader**
    - U-boot 2015.01 with default ddr configurations:

- T1040RDB with one DDR controllers
  - Enable chip select interleaving mode: CS0+CS1
- **FMAN ucode**

  fsl_fman_ucode_t1040_r1.1_106_4_17.bin.
- **Operating system**

  Default kernel in ISO
- **Additional root filesystem considerations**
  - For Linux: uImage-t1040d4rdb.bin and fsl-image-core-t1040d4rdb.ext2.gz.u-boot
  - For ASF: uImage.ppce5500-asf and fsl-image-asf-t1040d4rdb.ext2.gz.u-boot

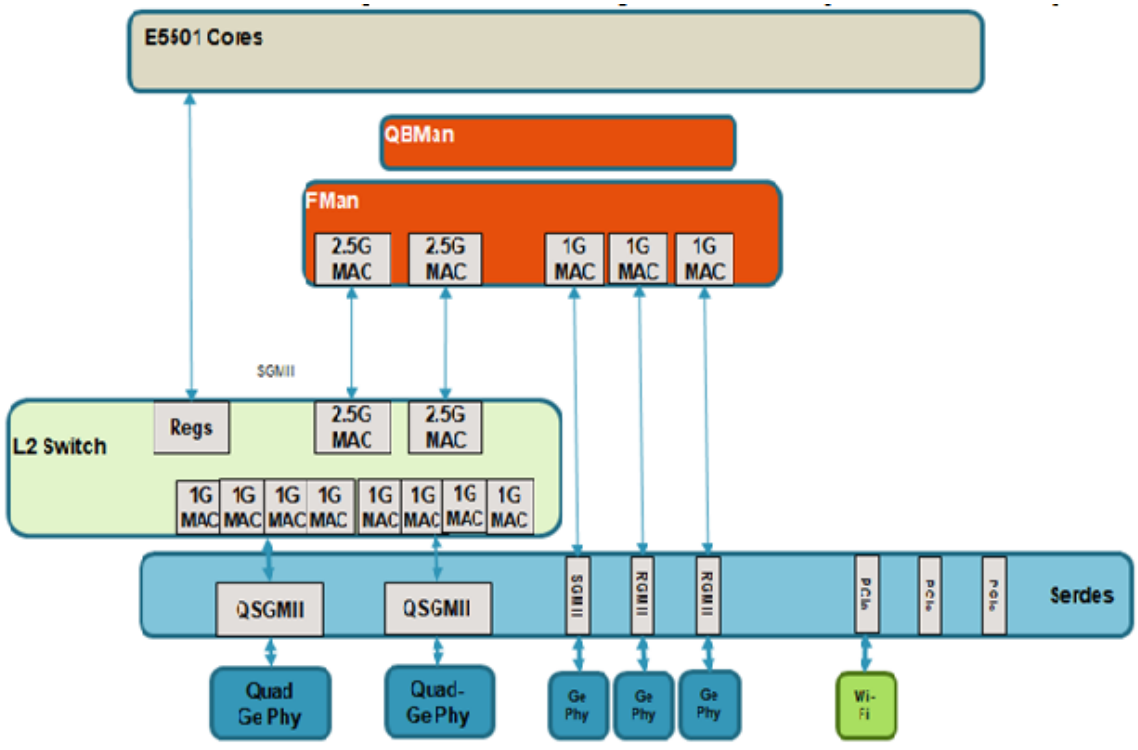
12.2.1.3.2 Test Procedure

This section describes the test procedure for various configurations of IPv4 forwarding

Test configuration

Different platform has different Ethernet interface, please refer to SDK document (Selecting Ethernet interfaces) to configure Ethernet interfaces for 8x1G Linux IPFWD.

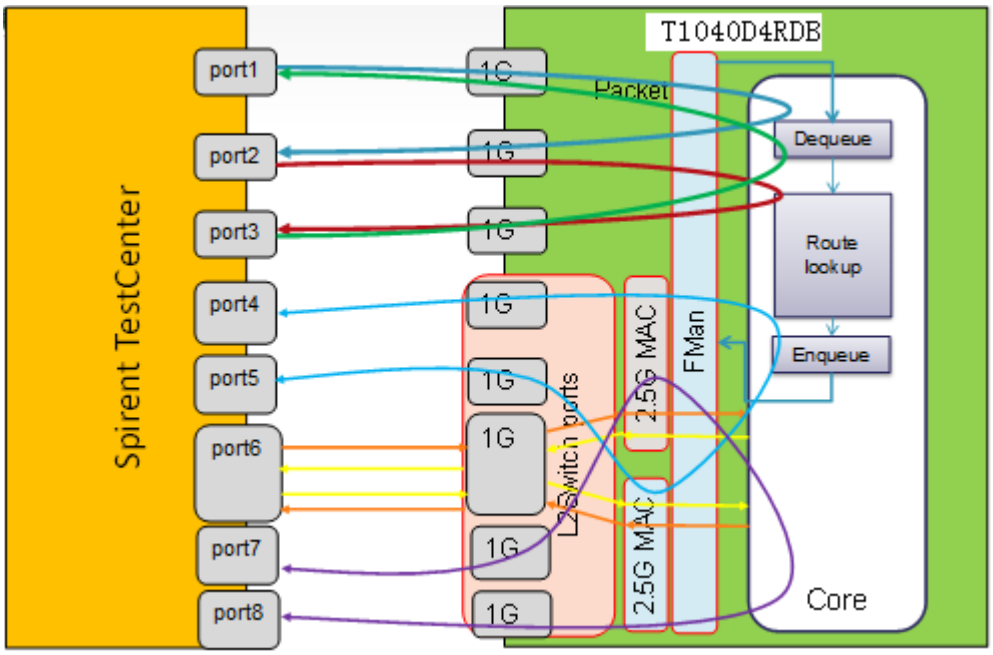
T1040D4RDB have 8 1Gb L2 switch ports which are external ports and 1 SGMII and 2 RGMII ports, like as followings:



Note: two internal 2.5G mac of l2switch which connected to Fman's two 2.5G mac.

So traffic which will go through DPAA on T1040D4RDB will use following Ethernet interfaces: fm1-gb0(2.5G)/fm1-gb1(2.5G)/ fm1-gb2(SGMII)/fm1-gb3(RGMII)/fm1-gb4(RGMII). And the maximum traffic what we can inject to DPAA is 8Gbps. Thus,

Linux/ASF/USDPAA IPv4 forward chooses five L2 switch ports and three SGMII/RGMII ports for benchmark test. The following diagram show traffic flow which will go through to DPAA:



1. **4.1.1 I8x1Gb Ethernet IPv4 Forward 1020 flows(bi-directional) configuration:**
   - Note: DMAC is mac address of fm1-gbx (0, 1, 2, 3, 4) of T1040RDB board. Please make change based on target board.
   - Spirent Testcenter configuration for Linux IPv4 Forward:
     - Port 1 to Port 5 < 1Gbps >:
       - Src: 192.85.1.2 – 192.85.1.18 -> Dst: 192.85.4.2 –192.85.4.5
       - SMac: 00:10:94:00:00:01 -> DMac: <fm1-gb0 mac address>
     - Port 2 to Port 4 < 1Gbps >:
       - Src: 192.85.1.20 – 192.85.1.36 -> Dst: 192.85.2.20 – 192.85.2.23
       - SMac: 00:10:94:00:00:02 -> DMac: <fm1-gb0 mac address>
     - Port 3 to Port 3 < 0.5Gbps >:
       - Src: 192.85.1.40 – 192.85.1.56 -> Dst: 192.85.2.40 – 192.85.2.43
       - SMac: 00:10:94:00:00:03 -> DMac: <fm1-gb0 mac address>
     - Port 3 to Port 3 < 0.5Gbps >:
       - Src: 192.85.2.40 – 192.85.2.56 -> Dst: 192.85.1.40 – 192.85.1.43
       - SMac: 00:10:94:00:00:06 -> DMac: <fm1-gb1 mac address>
     - Port 4 to Port 2 < 1Gbps >:
       - Src: 192.85.2.20 – 192.85.2.36 -> Dst: 192.85.1.20 – 192.85.1.23
       - SMac: 00:10:94:00:00:05 -> DMac: <fm1-gb1 mac address>
     - Port 5 to Port 1 < 1Gbps >:
       - Src: 192.85.2.2 – 192.85.2.18 -> Dst: 192.85.1.2 – 192.85.1.5
       - SMac: 00:10:94:00:00:04 -> DMac: <fm1-gb1 mac address>

- Port 6 to Port 7 < 1Gbps >:
  - Src: 192.85.3.2 – 192.85.3.18 -> Dst: 192.85.4.2 – 192.85.4.13
  - SMac: 00:10:94:00:10:01 -> DMac: <fm1-gb2 mac address>
- Port 7 to Port 8< 1Gbps >:
  - Src: 192.85.4.2 – 192.85.4.18 -> Dst: 192.85.5.2 – 192.85.5.13
  - SMac: 00:10:94:00:10:02 -> DMac: <fm1-gb3 mac address>
- Port 8 to Port 6 < 1Gbps >:
  - Src: 192.85.5.2 – 192.85.5.18 -> Dst: 192.85.3.2 – 192.85.3.15
  - SMac: 00:10:94:00:10:03 -> DMac: <fm1-gb4 mac address>

- Spirent Testcenter configuration for ASF IPv4 Forward:
  - Port 1 to Port 5 < 1Gbps >:
    - Src: 192.85.1.2 -> Dst: 192.85.2.2 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:00:01 -> DMac: <fm1-gb0 mac address>
  - Port 2 to Port 4 < 1Gbps >:
    - Src: 192.85.1.20 -> Dst: 192.85.2.20 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:00:02 -> DMac: <fm1-gb0 mac address>
  - Port 3 to Port 3 < 0.5Gbps >:
    - Src: 192.85.1.40 -> Dst: 192.85.2.40 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:00:03 -> DMac: <fm1-gb0 mac address>
  - Port 3 to Port 3 < 0.5Gbps >:
    - Src: 192.85.2.40 -> Dst: 192.85.1.40 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:00:06 -> DMac: <fm1-gb1 mac address>
  - Port 4 to Port 2 < 1Gbps >:
    - Src: 192.85.2.20 -> Dst: 192.85.1.20 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:00:05 -> DMac: <fm1-gb1 mac address>
  - Port 5 to Port 1 < 1Gbps >:
    - Src: 192.85.2.2 -> Dst: 192.85.1.2 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:00:04 -> DMac: <fm1-gb1 mac address>
  - Port 6 to Port 7 < 1Gbps >:
    - Src: 192.85.3.2 -> Dst: 192.85.4.2 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:10:01 -> DMac: <fm1-gb2 mac address>
  - Port 7 to Port 8< 1Gbps >:
    - Src: 192.85.4.2 -> Dst: 192.85.5.2 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:10:02 -> DMac: <fm1-gb3 mac address>
  - Port 8 to Port 6 < 1Gbps >:
    - Src: 192.85.5.2 -> Dst: 192.85.3.2 Src port:10000 - 10016 ->Dst port: 10000 – 10003
    - SMac: 00:10:94:00:10:03 -> DMac: <fm1-gb4 mac address>
- Configuration for Linux IPv4 Forward on T1040D4RDB:

- Configure L2switch settings
  - #killall netserver
  - #killall l2sw_bin
  - #l2sw_bin
  - mac add 00:10:94:00:00:01 0
  - mac add 00:10:94:00:00:02 1
  - mac add 00:10:94:00:00:03 2
  - mac add 00:10:94:00:00:06 2
  - mac add 00:10:94:00:00:05 3
  - mac add 00:10:94:00:00:04 4
  - mac add <fm1-gb3 mac address> 8
  - mac add <fm1-gb4 mac address> 9
- Configure network settings for fm1-gb0 – fm1-gb4
  - #killall netserver
  - #killall l2sw_bin
  - #l2sw_bin
  - mac add 00:10:94:00:00:01 0
  - mac add 00:10:94:00:00:02 1
  - mac add 00:10:94:00:00:03 2
  - mac add 00:10:94:00:00:06 2
  - mac add 00:10:94:00:00:05 3
  - mac add 00:10:94:00:00:04 4
  - mac add <fm1-gb3 mac address> 8
  - mac add <fm1-gb4 mac address> 9

```
cat >  ipv4_fwd_8g_8port.sh << EOF
                #!/bin/sh

eth0=fm1-gb0
eth1=fm1-gb1
eth2=fm1-gb2
eth3=fm1-gb3
eth4=fm1-gb4

ifconfig \$eth0 192.85.1.1/24 up
ifconfig \$eth1 192.85.2.1/24 up
ifconfig \$eth2 192.85.3.1/24 up
ifconfig \$eth3 192.85.4.1/24 up
ifconfig \$eth4 192.85.5.1/24 up

# for fm1-gb2, fm1-gb3, fm1-gb4
for ((i=2;i<=18;i++))
do
        arp -s 192.85.3.\$i 00:10:94:00:10:01
        arp -s 192.85.4.\$i 00:10:94:00:10:02
        arp -s 192.85.5.\$i 00:10:94:00:10:03
```

```
done

# for fm1-gb0 and fm1-gb1
for ((i=2;i<=18;i++))
do
        arp -s 192.85.1.\$i 00:10:94:00:00:01
        arp -s 192.85.1.\`expr \$i + 18\` 00:10:94:00:00:02
        arp -s 192.85.1.\`expr \$i + 38\` 00:10:94:00:00:03
        arp -s 192.85.2.\$i 00:10:94:00:00:04
        arp -s 192.85.2.\`expr \$i + 18\` 00:10:94:00:00:05
        arp -s 192.85.2.\`expr \$i + 38\` 00:10:94:00:00:06

done

echo 1 > /proc/sys/net/ipv4/ip_forward
EOF
```

Running the test

After Spirent Testcenter application is completely configured, the IPv4 automated forward throughput measurement can be started.

Appendix

1. **No need to configure PCD**

12.2.2 IPv4 Forward - eTSEC

12.2.2.1 Test Environment

Benchmarking Objectives

The purpose of this guide is to show how to setup a IPv4 Forwarding system on a NXP QorIQ processor based system and how to measure its performance.

NOTE

LS1021ATWR was used for the demo. Please refer to the board's manual for other boards.

After a baseline of performance and identifying optimizations these activities are possible:

1. Review the results with SDK development and any other groups or individuals who may have valuable input that could increase performance.
2. Post the results and this guide so that groups within NXP may share them with customers.
3. Compare these results with other NXP and competitors' products.
4. Provide assistance to any customer activities that could result in design wins.

Once established, the baseline can be used as a starting point for investigating other changes that may improve performance.

Test environment for eTSEC Platforms

A list of eTSEC platforms:

1. LS1021ATWR

For board switch settings, please see to the corresponding reference manual. Operating frequencies (CPU, CCB, DDR) must be set to the highest allowed values (check the product reference manual).

Test Equipment

The test equipment for this effort:

- Network Traffic generator and analyzer:
  1. SmartBits SMB-600B 2 slot chassis.
  2. 2 x LAN-3321A TeraMetric modules.
- SmartFlow application

Software Platform Identification

All software was built from SDK ISO.

Table 601. Target Development System Software

Resource	Description
Boot Loader	U-boot with NXP-specific patches was included SDK ISO. Configuration: • Enable cacheline interleaving mode: CS0+CS1 (if the DIMM support chip select interleaving) • Enable DDR controller interleaving mode: cache line (if there are two DDR controllers)
Operating system	Linux Source code with NXP-specific patches was included SDK ISO. This is the default kernel package for the SDK configuration.
Root filesystem considerations	The testing procedure makes use of the utilities **arp**, **ifconfig** and **ethtool**. These are part of the net-tools and iproute2 packages, which should be included in the root filesystem.

12.2.2.2 Test Procedure

Test procedure for IPv4 Forwarding benchmarking tests.

Test configuration

The configuration used is detailed in the figure below:

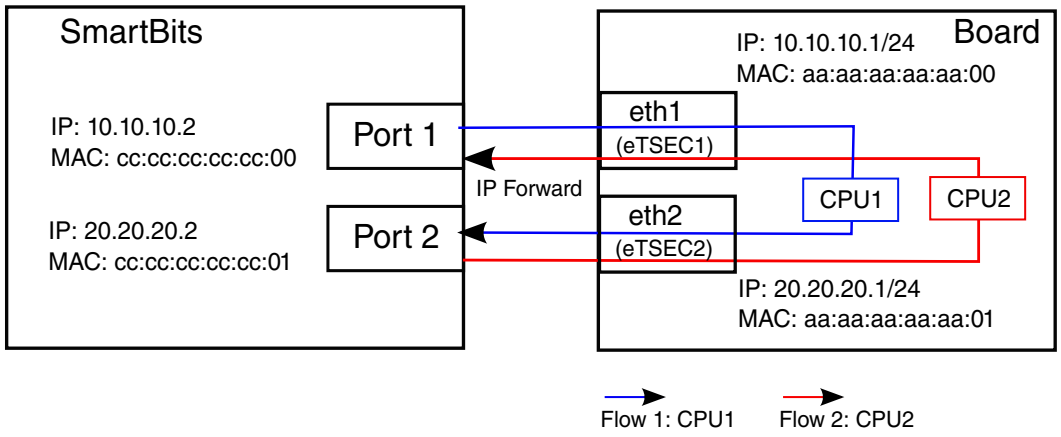


Figure 379. Configurations

Test Equipment Configuration

Test parameters:

Table 602. Test equipment parameters

Parameter	Description
Frame Size:	• 64 bytes • 128 bytes • 256 bytes • 390 bytes • 512 bytes • 1024 bytes • 1280 bytes • 1518 bytes
Test Duration	• 60 seconds
Resolution	• 0.1%
Acceptable Frame Loss	• 0.001%

Test Setup

Table 603. Target Development System Software

Task	Description

Make sure that operating frequencies (CPU, CCB, DDR) are set to the highest allowed values (check the corresponding platform reference manual);	• `CPU:  Freescale LayerScape LS1021E, Version: 2.0, (0x87081120)`
	`Clock Configuration:`
	`CPU0(ARMV7):1000 MHz,`
	`Bus:300 MHz, DDR:800 MHz (1600 MT/s data rate),`

Kernel configuration: The provided Linux kernel comes with a default configuration file). In order to obtain better performance, the optimizations shown at left must be enabled:	• `CONFIG_DEBUG_KERNEL` - disabled (default: built-in). • `CONFIG_NR_CPUS` - 2 (default: 8). • `CONFIG_USB_SUPPORT` - disabled (default: built-in).
Kernel boot parameters used:	• `root=/dev/ram rw console=ttyS0,115200 ramdisk_size=1000000 log_buf_len=128K` **NOTE** ramboot is preferred when performing benchmarking, since nfsboot might incur performance hit due to inherent network traffic

Table continues on the next page...

Table 603. Target Development System Software (continued)

Task	Description

Set up SMP interrupt affinities in order to route the flows to separate CPUs. For example:	• Make Flow1 affine to CPU1, that is, setup affinity of eth1_g0_rx and eth2_g0_tx interrupt handlers to 1, for example:
	• `#echo 1 > /proc/irq/36/smp_affinity`
	`#echo 1 > /proc/irq/31/smp_affinity`
	• Respectively make Flow2 affine to CPU2, meaning eth2_g0_rx and eth1_g0_tx have their affinity set to 2, for instance:
	• `#echo 2 > /proc/irq/32/smp_affinity`
	`#echo 2 > /proc/irq/35/smp_affinity`

Tune `Gianfar` driver parameters:	• Setting the Rx/Tx interrupt coalescing thresholds for packet count and microseconds:
	• `#ethtool -C eth1 rx-frames 32 rx-usecs 22`
	`#ethtool -C eth1 tx-frames 32 tx-usecs 22`
	`#ethtool -C eth2 rx-frames 32 rx-usecs 22`
	`#ethtool -C eth2 tx-frames 32 tx-usecs 22`
	• where ~11 usecs is the approx. time needed to transfer 16 frames of 64B payload each at line rate over a 1Gbit link

Setup the arp entries and enable IPv4 forwarding (see figure above):	• `#arp -s 10.10.10.2 cc:cc:cc:cc:cc:00 -i eth1`
	`#arp -s 20.20.20.2 cc:cc:cc:cc:cc:01 -i eth2`
	`#sysctl -w net.ipv4.ip_forward=1`

To further optimize the throughput, kill unneeded user space processes (if any). For example:	• `/etc/init.d/lighttpd stop`
	`/etc/init.d/crond stop`
	`/etc/init.d/sshd stop`
	`/etc/init.d/xinetd stop`
	`/etc/init.d/cups stop`
	`/etc/init.d/nfsserver stop`
	NOTE
	The exact commands depend on the features included in the root filesystem.

Results

The SmartFlow application reports the throughput results at the end of the test session.

12.2.3 IPSec Forward - DPAA

12.2.3.1 IPSec Forward - DPAA

12.2.3.1.1 Test Environment

Baseline the Linux IPSec performance

Objective

Identify any optimizations that have been discovered and ensure they are implemented on the QorIQ DPAA platform. Investigate other changes that may improve performance. This guide provides details for reproducing Linux IPSec forwarding performance results.

References

- Benchmarking Terminology for Network Interconnect Devices, IETF RFC 1242.
- Benchmarking Methodology for Network Interconnect Devices, IETF RFC 2544.
- IPv4 Forwarding Application-Level Benchmark Implementation Agreement Revision 1.0
- NPF IPv4 Unicast Forwarding API, Network Processing Forum.
- Internet IP Packet Size Distribution, Network Processing Forum contribution npf2001.078.00
- Annex to IPv4 Forwarding Application Level Benchmark Specification, Version 0.1, Network Processing Forum
- Transition Mechanisms for IPv6 Hosts and Routers; IETF RFC 2893.
- Connection of IPv6 Domains via IPv4 Clouds; IETF RFC 3056
- Benchmarking Terminology for LAN Switching Devices; IETF RFC 2285
- Benchmarking Methodology for LAN Switching Devices; IETF RFC 2289
- Internet Protocol, Version 6 (IPv6) Specification; IETF RFC 2460
- Route Table Study Group, Proposal for generating and using IP forwarding tables for NPF Benchmark Implementation Agreements, NPF2003.004
- Annex to IP Forwarding Application Level Benchmark Specification, Version 0.1, Network Processing Forum

Hardware Platform Identification

P2040RDB

- Board: P2040RDB, CPLD version: 2.2
- CPU0: P2040E, Version: 2.0
- Core: E500MC, Version: 3.2
- Clock Configuration:
  - CPU0: 1200 MHz, CPU1:1200 MHz, CPU2:1200 MHz, CPU3:1200 MHz,
  - CCB: 600 MHz,
  - DDR: 600 MHz (1200 MT/s data rate) (Asynchronous), LBC:75 MHz
  - FMAN1: 500 MHz
- 3 on-board SGMII ports and 2 on-board RGMII ports

P3041DS

- Board: P3041DS, Sys ID: 0x1c, Sys Ver: 0x11, FPGA Ver: 0x05
- CPU: P3041E, Version: 2.0
- Core: E500MC, Version: 2.2
- Clock Configuration:

  - CPU0: 1500 MHz, CPU1:1500 MHz, CPU2:1500 MHz, CPU3:1500 MHz
  - CCB: 750 MHz
  - DDR: 666.667 MHz (1333.333 MT/s data rate) (Asynchronous), LBC:93.750 MHz
  - FMAN1: 583.333 MHz
- 1 SGMII riser card with 3 Ethernet ports and 2 on-board RGMII ports
- 1 10G XAUI riser card
- PCIe e1000 NIC

P4080DS

- Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0c
- CPU: P4080E, Version: 3.0
- Core: E500MC, Version: 3.1
- Clock Configuration:
  - CPU0: 1499.985 MHz, CPU1:1499.985 MHz, CPU2:1499.985 MHz, CPU3:1499.985 MHz
  - CPU4: 1499.985 MHz, CPU5:1499.985 MHz, CPU6:1499.985 MHz, CPU7:1499.985 MHz
  - CCB: 799.992 MHz
  - DDR: 649.994 MHz (1299.987 MT/s data rate) (Asynchronous), LBC:99.999 MHz
  - FMAN1: 599.994 MHz
  - FMAN2: 599.994 MHz
- 1 SGMII riser card with 4 Ethernet ports
- 2 10G XAUI riser cards

P5020DS

- Board: P5020DS, Sys ID: 0x1c, Sys Ver: 0x01, FPGA Ver: 0x03
- CPU: P5020E, Version: 2.0
- Core: E5500, Version: 1.2
- Clock Configuration:
  - CPU0: 2000 MHz, CPU1:2000 MHz
  - CCB: 800 MHz
  - DDR: 666.667 MHz (1333.333 MT/s data rate) (Asynchronous), LBC:100 MHz
  - FMAN1: 600 MHz
- 1 SGMII riser card with 3 Ethernet ports and 2 on-board RGMII ports
- 1 10G XAUI riser cards
- PCIe e1000 NIC

P5040DS

- Board: P5040DS, Sys ID: 0x20, Sys Ver: 0x02, FPGA Ver: 0x02
- CPU: P5040E, Version: 2.1
- Core: E5500, Version: 1.2
- Clock Configuration:
  - CPU0: 2266.667 MHz, CPU1:2266.667 MHz, CPU2:2266.667 MHz, CPU3:2266.667 MHz

  - CCB: 800 MHz
  - DDR: 800 MHz (1600 MT/s data rate) (Asynchronous), LBC:100 MHz
  - FMAN1: 600 MHz
  - FMAN2: 600 MHz
- 2 SGMII riser card with 2 Ethernet ports on each and 2 on-board RGMII ports
- 2 10G XAUI riser cards
- PCIe e1000 NIC

B4860QDS

- Board: B4860QDS, Sys ID: 0x21, Sys Ver: 0x12
- CPU: B4860E, Version: 2.2
- Core: E6500, Version: 2.0
- **Clock Configuration:**
  - CPU0: 1600 MHz, CPU1:1600 MHz, CPU2:1600 MHz, CPU3:1600 MHz
  - CCB: 666.667 MHz
  - DDR: 933.333 MHz (1866.667 MT/s data rate), IFC:166.667 MHz
  - FMAN1: 666.667 MHz
- 2 on-board SGMII ports
- 2 10G XFI ports

T4240QDS

- Board: T4240QDS, Sys ID: 0x1e, Sys Ver: 0x22
- CPU: T4240E, Version: 2.0
- Core: E6500, Version: 2.0
- **Clock Configuration:**
  - CPU0: 1800 MHz, CPU1:1800 MHz, CPU2:1800 MHz, CPU3:.1800 MHz
  - CPU4: 1800 MHz, CPU5:1800 MHz, CPU6:1800 MHz, CPU7:1800 MHz
  - CPU8: 1800 MHz, CPU9:1800 MHz, CPU10:1800 MHz, CPU11:1800 MHz
  - CCB: 733.333 MHz
  - DDR: 933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC: 183.333 MHz
  - FMAN1: 733.333 MHz
  - FMAN2: 733.333 MHz
- 2 on-board RGMII ports
- 4 10G XAUI riser cards
- PCIe e1000 NIC

T4240RDB

- Board: T4240RDB, Board rev: 0x04 CPLD ver: 0x0401
- CPU: T4240E, Version: 2.0
- Core: E6500, Version: 2.0
- **Clock Configuration:**

  - CPU0: 1800 MHz, CPU1:1800 MHz, CPU2:1800 MHz, CPU3:1800 MHz
  - CPU4: 1800 MHz, CPU5:1800 MHz, CPU6:1800 MHz, CPU7:1800 MHz
  - CPU8: 1800 MHz, CPU9:1800 MHz, CPU10:1800 MHz, CPU11:1800 MHz
  - CCB: 733.333 MHz
  - DDR: 933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:183.333 MHz
  - FMAN1: 733.333 MHz
  - FMAN2: 733.333 MHz
- 8 on-board SGMII ports
- 4 10G XFI ports
- PCIe e1000 NIC

T2080QDS

- Board: T2080QDS, Sys ID: 0x28, Board Arch: V1, Board Version: A
- CPU: T2080E, Version: 1.1
- Core: E6500, Version: 2.0
- **Clock Configuration:**
  - CPU0: 1800 MHz, CPU1:1800 MHz, CPU2:1800 MHz, CPU3:1800 MHz
  - CCB: 700 MHz
  - DDR: 933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:175 MHz
  - FMAN1: 700 MHz
- 2 on-board RGMII ports
- 4 10G XFI ports
- PCIe e1000 NIC

T2080RDB

- Board: T2080RDB, Board rev: 0x01 CPLD ver: 0x03
- CPU: T2080E, Version: 1.1
- Core: E6500, Version: 2.0
- **Clock Configuration:**
  - CPU0: 1799.820 MHz, CPU1:1799.820 MHz, CPU2:1799.820 MHz, CPU3:1799.820 MHz
  - CCB: 599.940 MHz
  - DDR: 799.980 MHz (1599.960 MT/s data rate) (Asynchronous), IFC:149.985 MHz
  - FMAN1: 699.930 MHz
- 2 on-board RGMII ports
- 2 10G XFI ports2 10G BaseT ports
- PCIe e1000 NIC

T1023RDB

- Board: T1023RDB
- CPU: T1023E, Version: 1.0
- Core: E5500, Version: 2.1

- **Clock Configuration:**
  - CPU0: 1400 MHz, CPU1:1400 MHz
  - CCB: 400 MHz
  - DDR: 800 MHz (1600 MT/s data rate) (Asynchronous), IFC:100 MHz
  - FMAN1: 700 MHz
- 1 on-board RGMII port
- 1 on-board SGMII port
- 1 2.5G SGMII port
- PCIe e1000 NIC

T1042D4RDB

- Board: T1042D4RDB, Board rev: 0x01 CPLD ver: 0x08
- CPU: T1042E, Version: 1.1
- Core: E5500, Version: 2.1
- **Clock Configuration:**
  - CPU0: 1200 MHz, CPU1:1200 MHz, CPU2:1200 MHz, CPU3:1200 MHz
  - CCB: 500 MHz
  - DDR: 800 MHz (1600 MT/s data rate) (Asynchronous), IFC:125 MHz
  - FMAN1: 500 MHz
- 2 on-board RGMII ports
- PCIe e1000 NIC

Network Simulator

Spirent TestCenter Performance Analysis System

- Spirent Testcenter version: 4.41
- Choose Bound Stream Block for Traffic generator
- Choose Wizard -> Benchmarking -> RFC2544 from Spirent Testcenter to create command sequencer for throughput test
- Manual ZLT method:
  - Test Duration: 60 Seconds
  - Resolution: 0.1%
  - Packet Loss Rate: 0.001 %
  - Packet Size: 84(ipv6:86), 408,1424(ipv6-aes:1424) bytes

Target Development System Software

Software Platform Identification

All software was built from the QorIQ SDK ISO.

Boot loader

U-boot:

- Enable Cacheline interleaving mode: CS0+CS1
- Enable DDR Controller interleaving mode: cache line (If there are two DDR controllers),

RCW

- P2041RDB: RR_PH_0x19/rcw_5g_1200mhz.bin
- P3041DS: RR_HXAPNSP_0x36/rcw_15g_1500mhz.bin
- P4080DS: R_PPSXX_0xe/rcw_2sgmii_1500mhz_rev3.bin
- P5020DS: RR_HXAPNSP_0x36/rcw_15g_2000mhz.bin
- P5040DS: RR_XXSNSpP_0x02/rcw_26g_2267mhz.bin
- B4860QDS: N_RSSS_0x2A_0x8D/rcw_4sgmii_4srio_2xfi_1600mhz_1866ddr_rev2.bin
- T4240QDS: RR_XXXXPRPR_1_1_5_5/rcw_1_1_5_5_1666MHz_rev2.bin
- T4240RDB: SSFFPPH_27_55_1_9/rcw_27_55_1_9_1666MHz.bin
- T2080QDS: RR_PNNPPH_66_15/rcw_66_15_1800MHz_rev11.bin
- T2080RDB: RRFFXX_P_66_15/rcw_66_15_1800MHz_rev11.bin
- T1023RDB: RNSS_NPP_77/rcw_77_1400MHz.rcw
- T1042D4RDB: RR_P_86/rcw_1400MHz.bin

FMan ucode(Please refer to SDK document to use correct ucode version)

- P2041 rev 2.0 - fsl_fman_ucode_p2041_r2.0_106_1_18.bin
- P3041 rev 2.0 - fsl_fman_ucode_p3041_r2.0_106_1_18.bin
- P4080 rev 3.0 - fsl_fman_ucode_p4080_r3.0_106_2_18.bin
- P5020 rev 2.0 - fsl_fman_ucode_p5020_r2.0_106_1_18.bin
- P5040 rev 2.0 - fsl_fman_ucode_p5040_r2.0_106_1_18.bin
- B4860 rev 2.2 - fsl_fman_ucode_b4860_r2.0_106_4_18.bin
- T4240 rev 2.0 - fsl_fman_ucode_t4240_r2.0_106_4_18.bin
- T2080 rev 1.1 - fsl_fman_ucode_t2080_r1.1_106_4_18.bin
- T1023 rev 1.0 - fsl_fman_ucode_t1024_r1.0_106_4_18.bin
- T1042 rev 1.1 - fsl_fman_ucode_t1040_r1.1_106_4_18.bin

Operating System

Default kernel in ISO

NOTE: For **T2080QDS** and **T2080RDB** performance measurements, the following kernel parameters must be used:

- CONFIG_FSL_DPAA_ETH_MAX_BUF_COUNT=512
- CONFIG_FSL_DPAA_ETH_REFILL_THRESHOLD=384

Additional root filesystem considerations

The tests procedure for IPv4 forwarding makes use of the "arp", "ifconfig" and “ip” utilities. These are part of the net-tools and iproute2 packages, which the root filesystem stored in the hard disk should have.

12.2.3.1.2 Test Procedure

Test procedure for various configurations of IPv4 Sec & Forwarding. Let’s refer to P3041DS board for example.

Test Configuration

P3041 has five 1Gb Ethernet interfaces and 1 10Gb Ethernet interface. There are fm1-gb0, fm1-gb1, fm1-gb2, fm1-gb3, fm1-gb4, fm1-10g.

Linux Ipv4 SEC Forward Configuration

- 128 tunnels (bi-directional) configuration:
  - Spirent TestCenter configuration:
    - Port 1 to Port 2: Src: 192.85.1.2 – 192.85.1.9 -> Dst: 192.86.1.2 – 192.86.1.9
    - Port 2 to Port 1: Src: 192.86.1.2 – 192.86.1.9 -> Src: 192.85.1.2 – 192.85.1.9

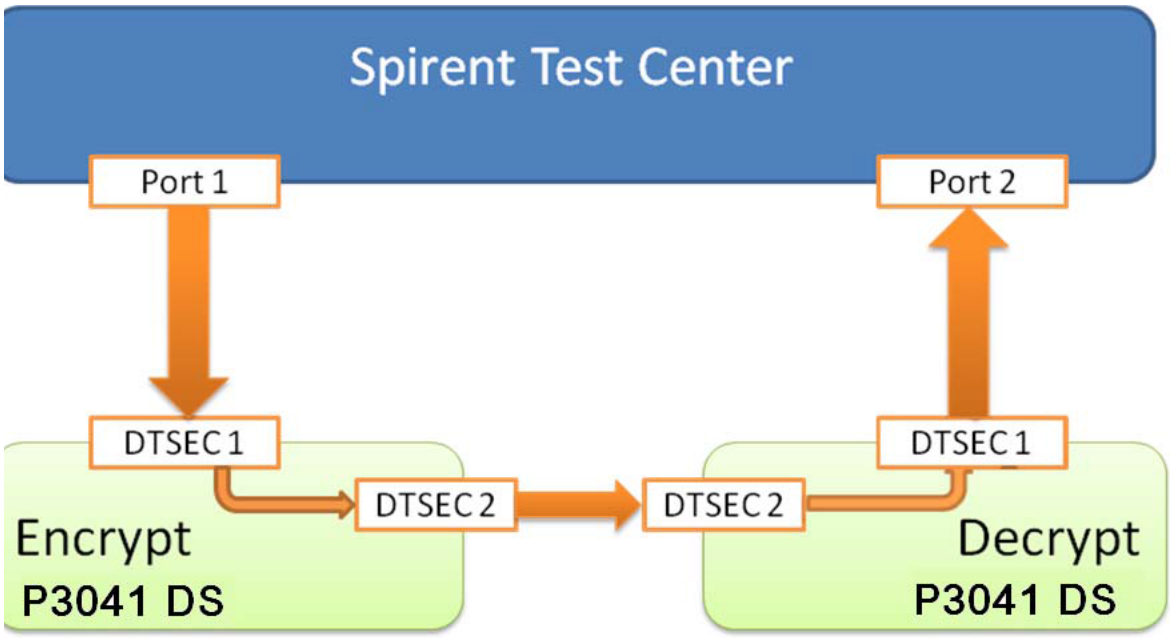


Figure 380. IPSecfwd Configuration

- Linux configuration:
  - The Linux configuration file is listed below.
  - On left gateway(Encrypt) board, run: bash iproute_128tunnels_p3041.sh left
  - On right gateway(Decrypt) board run: iproute_128tunnels_p3041.sh right

NOTE

The default left gateway board is board_left, and the right is board_right, you can modify the script "iproute_128tunnels_p3041.sh" what you want.

NOTE

For 2x10G linux ipsec, only need to replace above 1G interface to 10G interface.

Running the Test

After Sprient Testcenter application has been completely configured, start to send traffic for measuring IPv4 IPsec throughput.

Linux Configuration File

```
#!/bin/bash
eth0=fm1-gb0
eth1=fm1-gb1

make_esp_tunnel() {

# ESP SAs do$dir2g encryption us$dir2g 192 bit long keys (168 + 24 parity)
# and hmac-sha1 authentication us$dir2g 160 bit long keys

echo "add $1 $2 esp 0x$3 -m tunnel
    -E $4  0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831
    -A hmac-sha1 0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;" | setkey -c

echo "add $2 $1 esp 0x`expr $3 + 100` -m tunnel
    -E $4  0xf6ddb555acfd9d77b03ea3843f2653255afe8eb5573965df
    -A hmac-sha1 0xea6856479330dc9c17b8f6c37e2a895363d83f21;" | setkey -c

}

make_esp_policy() {

# Security policies1
if [ $1 == left ]
then
    dir1=out
    dir2=in
    echo "spdadd $2 $3 any -P $dir1 ipsec
           esp/tunnel/$4-$5/require;" | setkey -c
    echo "spdadd $3 $2 any -P $dir2 ipsec
           esp/tunnel/$5-$4/require;" | setkey -c
else
    dir1=in
    dir2=out
    echo "spdadd $2 $3 any -P $dir1 ipsec
           esp/tunnel/$4-$5/require;" | setkey -c
    echo "spdadd $3 $2 any -P $dir2 ipsec
           esp/tunnel/$5-$4/require;" | setkey -c
fi

}

# Flush the SAD and SPD
setkey -F
setkey -FP

# set ip address
left_addr_ip=192.85.1.1
right_addr_ip=192.86.1.1
left_src_mac=00:10:94:00:00:01
right_src_mac=00:10:94:00:00:02
proto="aes-cbc"
base1=200
base2=200
echo 1 > /proc/sys/net/ipv4/ip_forward

case $1 in
        left)
                ifconfig $eth0 $left_addr_ip
```

```
                i=2
                for((j=2;j<10;j++))
                do
                    arp -s 192.85.1.$k $left_src_mac -i $eth0
                    for((k=2;k<10;k++))
                    do
                    if [ $base2 == 256 ]
                    then
                    base2=`expr $base2 - 256`
                        base1=`expr $base1 + 1`
                    fi
                    ip addr add 200.$base1.$base2.10/24 dev $eth1
                        make_esp_policy $1 192.85.1.$j 192.86.1.$k 200.$base1.$base2.10
200.$base1.$base2.20
                        make_esp_tunnel 200.$base1.$base2.10 200.$base1.$base2.20 `expr 200 + $i`
$proto
                        ((base2++))
                        ((i++))
                    done
                done
ifconfig $eth1 up
route add default dev $eth1

            ;;
        right)
                ifconfig $eth0 $right_addr_ip
                i=2
                for((j=2;j<10;j++))
                do
                    arp -s 192.86.1.$k $left_src_mac -i $eth0
                    for((k=2;k<10;k++))
                    do
                    if [ $base2 == 256 ]
                    then
                    base2=`expr $base2 - 256`
                        base1=`expr $base1 + 1`
                    fi
                    ip addr add 200.$base1.$base2.20/24 dev $eth1
                        make_esp_policy $1 192.85.1.$j 192.86.1.$k 200.$base1.$base2.10
200.$base1.$base2.20
                        make_esp_tunnel 200.$base1.$base2.10 200.$base1.$base2.20 `expr 200 + $i`
$proto
                        ((base2++))
                        ((i++))
                    done
                done
        ifconfig $eth1 up
        route add default dev $eth1

            ;;
esac
```

FMC configuration

In order to achieve the greatest performance, one needs to ensure that PCD files are applied. In order to do that, after applying the tunnel configuration above, one needs to execute a command similar to the following on both the *left* as well as the *right* board:

```
root@left :~# fmc -c /etc/fmc/config/private/<board>/<RCW name>/config.xml -p /etc/fmc/
config/private/<board>/<RCW name>/policy_ipv4.xml -a
```

Replace in the command above the board with the full name of the board i.e. T2080QDS and the RCW name with the name of the RCW file used on the board. For instance, for a T2080QDS, which is configured to use RR_PNNPPH_66_15, the command above would translate as below:

```
root@left :~# fmc -c /etc/fmc/config/private/t2080qds/RR_PNNPPH_66_15/config.xml -
p /etc/fmc/config/private/t2080qds/RR_PNNPPH_66_15/policy_ipv4.xml -a
```

For more information regarding the way the DPAA1 Ethernet driver and the FMC configuration file(s) interact, please consult the following topic: "FMC, FMD, and the ethernet Driver"

12.2.3.2 IPSec Forward - T1024RDB

12.2.3.2.1 Test Environment

The first objective is to baseline the Linux IPv4 Forward performance. Next identify any optimizations that have been discovered and ensure they are implemented on the QorIQ DPAA platform. Investigate other changes that may improve performance. This guide is intended to provide details for reproducing Linux IPv4 packet forwarding performance results for T1024RDB.

Platform Configurations

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/FMan	SerDes Protocol	Config. Name
T1024RDB	Rev C	Rev 1.0	• CPU - 1400 MHz • CCB - 400 MHz • DDR - 1600 MHz • FMan - 700 MHz	• 0x95 • 0x135	• RRX_PPP_95 • RNS_NPP_135

Network Simulator

Spirent TestCenter Performance Analysis System

- Spirent Testcenter version: 4.41
- Choose Bound Stream Block for Traffic generator
- Choose Wizard -> Benchmarking -> RFC2544 -> Throughput Test -> Sequencer sequencer for throughput test
- Testcenter configuration:
  - Test Duration: 60 Seconds
  - Resolution: 0.1%
  - Packet Loss Rate: 0.001 %
  - Packet Size: 84(ipv6:86), 408,1424(ipv6-aes:1424) bytes

Software Platform Identification

All software was built from SDK ISO. Please refer to following images:

Table 604. Images

Target Development System Software	Image file name/Description
FMan ucode	T1024 rev 1.0 - fsl_fman_ucode_t1024_r1.0_106_4_17.bin(Please refer to SDK document to use correct ucode version)
RCW	T1024RDB: RNS_NPP_135/rcw_135_1400MHz.rcw T1024RDB: RRX_PPP_95/rcw_95_1400MHz.rcw
Boot loader	u-boot-T1024RDB.bin
Linux Kernel	uImage-t1024rdb.bin
Device Tree	uImage-t1024rdb.dtb
Default File system	fsl-image-core-t1024rdb.ext2.gz.u-boot

- **Target Development System Software**
  - **Boot loader**
    - U-boot 2015.01 with default ddr configurations:
    - T1024RDB with one DDR controllers
      - Enable chip select interleaving mode: CS0+CS1
  - **FMAN ucode**

    fsl_fman_ucode_t1024_r1.0_106_4_17.bin.
  - **Operating system**

    Default kernel in ISO
  - **Additional root filesystem considerations**
    - For Linux: uImage-t1024rdb.bin and fsl-image-core-t1024rdb.ext2.gz.u-boot

12.2.3.2.2 Test Procedure

This section describes the test procedure for various configurations of IPv4 IPSec forwarding

Test configuration

Different platforms have different Ethernet interfaces, please refer to SDK documentation (Selecting Ethernet interfaces) to configure Ethernet interfaces for Linux IPSec forward.

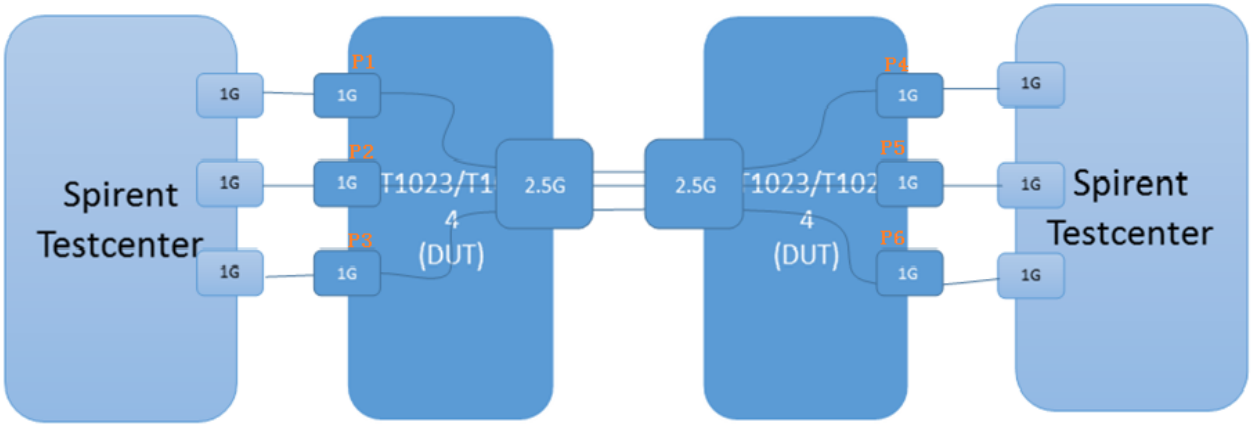
T1024RDB have two RCW:

- RCW_135: 1 x 1Gb rgmii ports + 1 x 2.5G SGMII port + 2 x mini_PCIE ports
- RCW_95: 2 x 1Gb rgmii port + 1 x 10G XFI port + 1 x PCIE port + + 2 x mini_PCIE ports

The 10G XFI port could speed down to 2.5G. So traffic which will go through DPAA on T1024RDB will use following Ethernet interfaces:

- RCW_135: fm1-mac3(2.5G)/fm1-mac4(RGMII)/eth*(mini_PCIE)/eth*(mini_PCIE)
- RCW_95: fm1-mac1(2.5G)/fm1-mac4(RGMII)/fm1-mac3(RGMII)/eth*(mini_PCIE)

show traffic flow which will go through to DPAA:



Note:

- Please use one TC to send traffic, placing two TC in the picture is aimed at showing ports' connection more clear.
- Ports Property:
  - For RCW_135:Port1 and Port4 are RGMII ports.others ports are all mini_PCIE;
  - For RCW_95:Port1/Port2/Port3/Port4 are RGMII ports.others ports are mini_PCIE;

1. 4.1.1 IPv4 SEC Forward 6g 128 flows(bi-directional) configuration:(take RCW_135 as an example)
   - Spirent Testcenter configuration for Linux IPv4 SEC Forward:
     - Port 1 to Port 4 :
       - Src: 192.85.1.2 – 192.85.1.9 -> Dst: 192.86.1.2 –192.85.4.5
       - SMac: 00:10:94:00:00:01 -> DMac: <fm1-mac3(left) mac address>
       - Gateway: 192.85.1.1
     - Port 2 to Port 5 :
       - Src: 192.87.1.2 – 192.87.1.5 -> Dst: 192.88.1.2 – 192.85.1.5
       - SMac: 00:10:94:00:00:03 -> DMac: <fm1-mac3(left) mac address>
       - Gateway: 192.87.1.1
     - Port 3 to Port 6:
       - Src: 192.89.1.2 – 192.89.1.5 -> Dst: 192.90.1.2 – 192.90.1.5
       - SMac: 00:10:94:00:00:05 -> DMac: <fm1-mac3(left) mac address>
       - Gateway: 192.89.1.1
     - Port 4 to Port 1:
       - Src: 192.86.1.2 – 192.86.1.5 -> Dst: 192.85.1.2 – 192.85.1.9
       - SMac: 00:10:94:00:00:02 -> DMac: <fm1-mac3(right) mac address>
       - Gateway: 192.86.1.1
     - Port 5 to Port 2 :
       - Src: 192.88.1.2 – 192.88.1.5 -> Dst: 192.87.1.2 – 192.87.1.5
       - SMac: 00:10:94:00:00:04 -> DMac: <fm1-mac3(right) mac address>
       - Gateway: 192.88.1.1
     - Port 6 to Port 3:

- Src: 192.90.1.2 – 192.90.1.5 -> Dst: 192.89.1.2 – 192.89.1.5
- SMac: 00:10:94:00:00:06 -> DMac: <fm1-mac3(right) mac address>
- Gateway: 192.90.1.1

- tunnel ip for Linux IPv4 SEC Forward:
  - tunnel ip of 2.5G SGMII port on left board :200.200.100.10
  - tunnel ip of 2.5G SGMII port on right board :200.200.100.20

Running the test

After Spirent Testcenter application is completely configured, the IPv4 SEC automated forward throughput measurement can be started.

Appendix

1. No need to configure PCD when test Linux IPFwd performance.
2. For RCW_135, as there are only 3 1G ports, we need use all of them to send traffic, suggests boot from lan server to run fully-automatic test.
3. Limited by board design, there is only one 1G fman port when 2.5G SGMII works, we only use one fman port to test usdpaa and asf on T1024RDB:2.5G SGMII port.
4. Limited by board design, there is only one 1G fman port when 2.5G SGMII works, for linux IPFwd and IPSec test, we can use 10G PCIe card to test performance of the 2.5G SGMII port.

12.2.3.3 IPSec Forward - T1040D4RDB

12.2.3.3.1 Test Environment

Benchmarking Objectives

The purpose of this guide is to show how to setup an IPSec Forwarding system on a NXP QorIQ processor based system and how to measure its performance.

Platform Configurations: T1040D4RDB

- Board: t1040D4RDB, board rev: 0x01, CPLD ver: 0x06, vBank: 4
- CPU0: t1040E, Version: 1.0
- Core: e5500, Version: 2.0
- Clock Configuration:
  - CPU0:1400 MHz, CPU1:1400 MHz, CPU2:1400 MHz, CPU3:1400 MHz,
  - CCB:600 MHz,
  - DDR:800 MHz (1600 MT/s data rate) (Asynchronous), IFC:150 MHz
  - QE: 300MHz
  - FMAN1: 600 MHz
  - QMAN: 300MHz
  - PME: 300 MHz
- 1 on board SGMII port and 2 on board RGMII ports
- 8 Gigabit switch ports are external ports and are connected to external Phys

Network Simulator: Spirent TestCenter Performance Analysis System

- Spirent Testcenter version: 4.41
- Choose Bound Stream Block for Traffic generator

- Choose Wizard -> Benchmarking-> RFC2544-> Throughput Test-> Sequencer
- Testcenter configuration:
  - Test Duration: 60 Seconds
  - Resolution: 0.1%
  - Packet Loss Rate: 0.001 %
  - Packet Size: 84(ipv6:86), 408,1424(ipv6-aes:1424) bytes

Software Platform Identification

All software was built from SDK ISO.

Host development System Software Target Development System Software **Boot loader**

- U-boot 2015. 01 with default ddr configurations:
- T1040D4RDB with one DDR controllers
  - Enable chip select interleaving mode: CS0+CS1

RCW

Serdes: RR_P_66

Rcw: rcw_1400MHz.bin

FMAN ucode

fsl_fman_ucode_t1040_r1.0_106_4_15.bin(Please refer to SDK document to use correct ucode version)

Operating system

Default kernel in ISO

uImage and Root filesystem

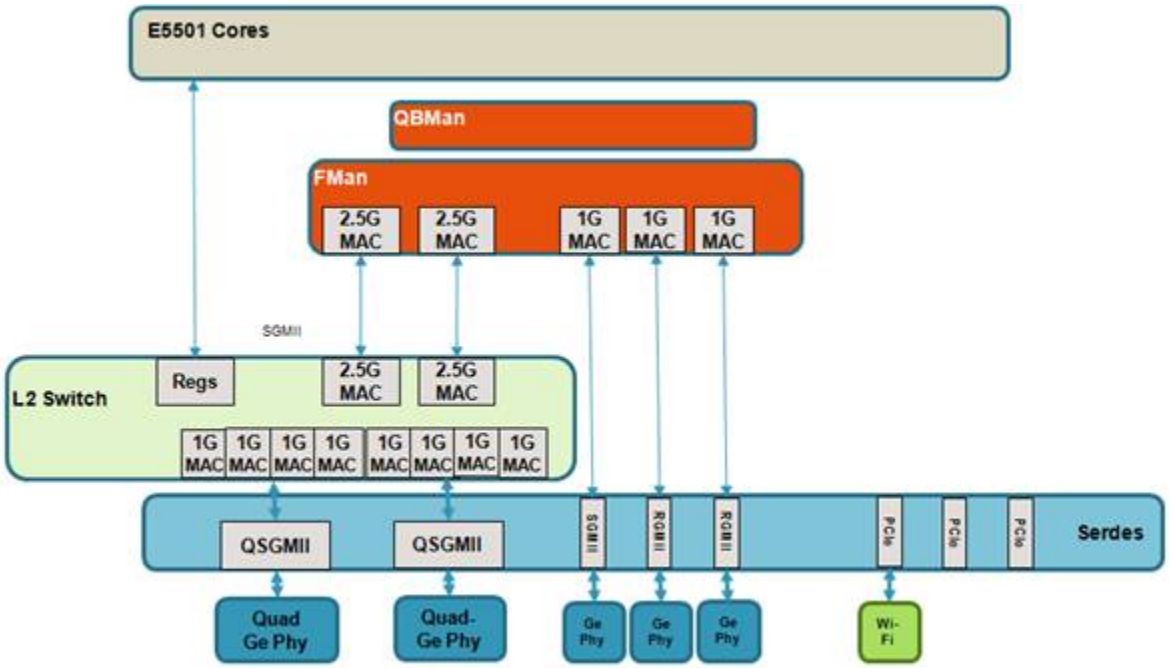
- For Linux: uImage-t1040d4rdb.bin and fsl-image-core-t1040d4rdb.ext2.gz.u-boot
- For ASF: uImage.ppce5500-asf and fsl-image-asf-t1040d4rdb.ext2.gz.u-boot

12.2.3.3.2 Test Procedure

This section describes the test procedure for various configurations of IPv4 IPSec forwarding

Test Configuration

T1040D4RDB have8 1Gb L2 switch ports which are external ports and 1 SGMII and 2 RGMII ports, like as followings:

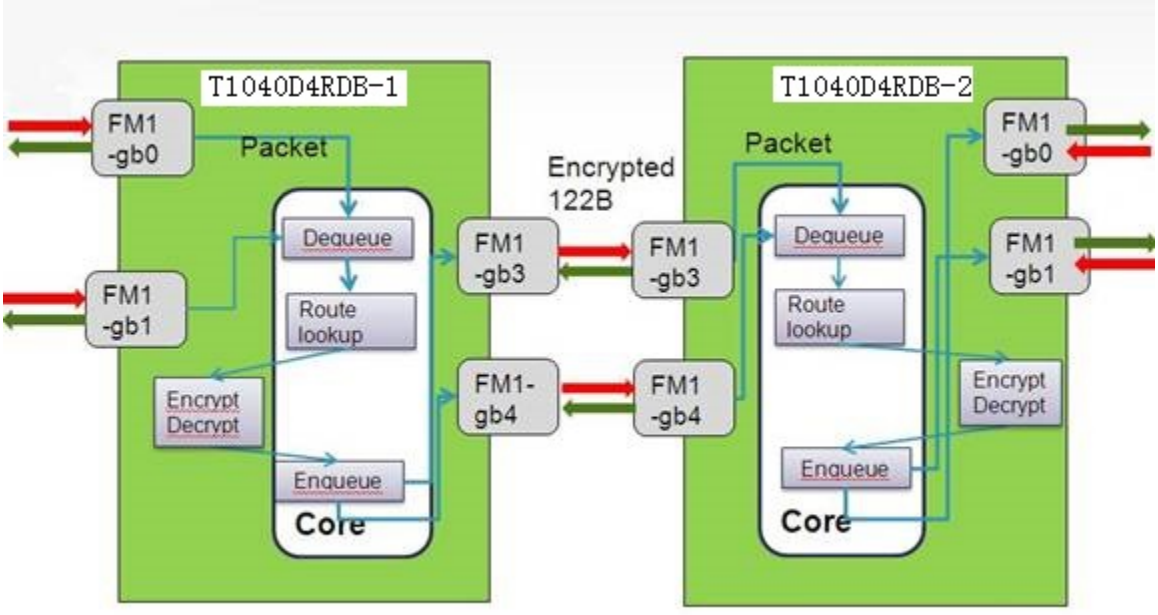


Note: two internal 2.5G mac of l2switch which connected to Fman's two 2.5G mac.

So traffic which will go through DPAA on T1040D4RDB will use following Ethernet interfaces for each board:

fm1-gb0(2.5G)/fm1-gb1(2.5G)/fm1-gb3(RGMII)/fm1-gb4(RGMII).

And the maximum traffic what we can inject to DPAA is 8Gbps. Thus, Linux/ASF/USDPAA IPv4 IPSec forward chooses four L2 switch ports and three SGMII/RGMII ports for benchmark test. The following diagram show traffic flow which will go through to DPAA:



4x1Gb Ethernet IPv4 IPSec Forward 128 flows(bi-directional) configuration:

Note: DMAC is mac address of fm1-gbx (0, 1, 3, 4) of T1040D4RDB board. Please make change based on target board.

- **Spirent Testcenter configuration for Linux IPv4 IPSec Forward:**
- Port 1 to T1040RDB-1 fm1-gb0 <1Gbps>:
  - Src: 192.85.1.2 – 192.85.1.9 -> Dst: 192.86.1.2 – 192.86.1.5
  - SMac: 00:10:94:00:00:01 -> DMac***: <fm1-gb0 mac address>***
- Port 2 to T1040RDB-1 fm1-gb1 <1Gbps>:
  - Src: 192.87.1.2 – 192.87.1.5 -> Src: 192.88.1.2 – 192.88.1.9
  - SMac: 00:10:94:00:00:03 -> DMac: ***<fm1-gb1 mac address>***
- Port 1 to T1040RDB-2 fm1-gb0 <1Gbps>:
  - Src: 192.86.1.2 – 192.85.1.5 -> Dst: 192.85.1.2 – 192.85.1.9
  - SMac: 00:10:94:00:00:02 -> DMac***: <fm1-gb0 mac address>***
- Port 2 to T1040RDB-2 fm1-gb1 <1Gbps>:
  - Src: 192.88.1.2 – 192.88.1.9 -> Src: 192.87.1.2 – 192.87.1.5
  - SMac: 00:10:94:00:00:04 -> DMac: ***<fm1-gb1 mac address>***
- **Spirent Testcenter configuration for ASF IPv4 IPSec Forward:**
- Port 1 to T1040RDB-1 fm1-gb0 <1Gbps>:
  - Src: 192.85.1.2 – 192.85.1.9 -> Dst: 192.86.1.2 – 192.86.1.5Src port: 10000 -> Dst port: 10000
  - SMac: 00:10:94:00:00:01 -> DMac***: <fm1-gb0 mac address>***
- Port 2 to T1040RDB-1 fm1-gb1 <1Gbps>:
  - Src: 192.87.1.2 – 192.87.1.5 -> Src: 192.88.1.2 – 192.88.1.9 Src port: 10000 -> Dst port: 10000
  - SMac: 00:10:94:00:00:03 -> DMac: ***<fm1-gb1 mac address>***
- Port 1 to T1040RDB-2 fm1-gb0 <1Gbps>:
  - Src: 192.86.1.2 – 192.85.1.5 -> Dst: 192.85.1.2 – 192.85.1.9 Src port: 10000 -> Dst port: 10000
  - SMac: 00:10:94:00:00:02 -> DMac***: <fm1-gb0 mac address>***
- Port 2 to T1040RDB-2 fm1-gb1 <1Gbps>:
  - Src: 192.88.1.2 – 192.88.1.9 -> Src: 192.87.1.2 – 192.87.1.5 Src port: 10000 -> Dst port: 10000
  - SMac: 00:10:94:00:00:04 -> DMac: ***<fm1-gb1 mac address>***
- **Configuration for Linux IPv4 Forward on T1040D4RDB:**
- Configure fman
- Configure L2switch settings
- #killall netserver

#killall l2sw_bin

#l2sw_bin

- On T1040D4RDB-1:

mac add 00:10:94:00:00:01 0

mac add 00:10:94:00:00:03 1

mac add ***<fm1-gb3 mac address>*** 8

mac add ***<fm1-gb4 mac address>*** 9

- On T1040D4RDB-2:

mac add 00:10:94:00:00:02 0

mac add 00:10:94:00:00:04 1

mac add ***<fm1-gb3 mac address>*** 8

mac add ***<fm1-gb4 mac address>*** 9

- Disable the PHY polling to low CPU usage

poll off

- Verify L2switch configuration

mac dump

- exited the l2sw_bin using “ctrl+c”
- Configure network settings for fm1-gb0 – fm1-gb4

```
#!/bin/sh
eth0=fm1-gb0
eth1=fm1-gb1
eth2=fm1-gb3
eth3=fm1-gb4

make_esp_tunnel() {

# ESP SAs do$dir2g encryption us$dir2g 192 bit long keys (168 + 24 parity)
# and hmac-sha1 authentication us$dir2g 160 bit long keys

echo "add $1 $2 esp 0x$3 -m tunnel
    -E $4  0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831
    -A hmac-sha1 0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;" | setkey -c

#echo "add $2 $1 esp 0x$(($3 + 100)) -m tunnel
#    -E $4  0xf6ddb555acfd9d77b03ea3843f2653255afe8eb5573965df
#    -A hmac-sha1 0xea6856479330dc9c17b8f6c37e2a895363d83f21;" | setkey -c

echo "add $2 $1 esp 0x$(($3 + 100)) -m tunnel
    -E $4  0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831
    -A hmac-sha1 0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;" | setkey -c
}

make_esp_policy() {

# Security policies1
if [ $1 == left ]
then
    dir1=out
    dir2=in
    echo "spdadd $2 [any] $3 [any] any -P $dir1 ipsec
           esp/tunnel/$4-$5/require;" | setkey -c
    echo "spdadd $3 [any] $2 [any] any -P $dir2 ipsec
           esp/tunnel/$5-$4/require;" | setkey -c
else
    dir1=in
    dir2=out
    echo "spdadd $2 [any] $3 [any] any -P $dir1 ipsec
           esp/tunnel/$4-$5/require;" | setkey -c
```

```
    echo "spdadd $3 [any] $2 [any] any -P $dir2 ipsec
          esp/tunnel/$5-$4/require;" | setkey -c
fi

}

# Flush the SAD and SPD
setkey -F
setkey -FP

# set ip address
left_addr_ip1=192.85.1.1 # for fm1-gb0 on left board
left_addr_ip2=192.87.1.1 # for fm1-gb1 on left board
right_addr_ip1=192.86.1.1 # for fm1-gb0 on right board
right_addr_ip2=192.88.1.1 # for fm1-gb1 on right board

left_src_mac1=00:10:94:00:00:01  # for STC port1 which connect to sw_p1 on left board
left_src_mac2=00:10:94:00:00:03  # for STC port3 which connect to sw_p2 on left board
right_src_mac1=00:10:94:00:00:02 # for STC port2 which connect to sw_p1 on right board
right_src_mac2=00:10:94:00:00:04 # for STC port4 which connect to sw_p2 on right board

proto="aes-cbc" # set protocol
base1=100 # ip address base for fm1-gb3 <--> fm1-gb3
base2=200 # ip address base for fm1-gb4 <--> fm1-gb4

echo 1 > /proc/sys/net/ipv4/ip_forward

case $1 in
        left)
                # set ip address on fm1-gb0/1
                ifconfig $eth0 $left_addr_ip1 netmask 255.255.0.0 up
                ifconfig $eth1 $left_addr_ip2 netmask 255.255.0.0 up
                ifconfig $eth2 up
                ifconfig $eth3 up

                # set route on fm1-gb3/4
                route add -net 192.86.0.0 netmask 255.255.0.0 dev $eth2
                route add -net 192.88.0.0 netmask 255.255.0.0 dev $eth3

                k=2
                for i in `seq 2 9`
                do
                    arp -s 192.85.1.$i $left_src_mac1 -i $eth0
                    for j in `seq 2 5`
                    do
                    while (($k < 6))
                    do
                        arp -s 192.87.1.$k $left_src_mac2 -i $eth1
                        k=$((k + 1))
                    done
                    ip addr add 200.200.$base1.10/24 dev $eth2
                    echo ===base1 is $base1===
                    ip addr add 200.200.$base2.30/24 dev $eth3
                    echo ===base2 is $base2===

                    make_esp_policy $1 192.85.1.$i 192.86.1.$j 200.200.$base1.10
200.200.$base1.20
                    make_esp_policy $1 192.87.1.$j 192.88.1.$i 200.200.$base2.30
200.200.$base2.40
                    make_esp_tunnel 200.200.$base1.10 200.200.$base1.20 $base1 $proto
```

```
                    base1=$(($base1 + 1))
                    make_esp_tunnel 200.200.$base2.30 200.200.$base2.40 $(($base2 + 100)) $proto
                    base2=$(($base2 + 1))
                    done
                done

            ;;
        right)
                # set ip address on fm1-gb0/1
                ifconfig $eth0 $right_addr_ip1 netmask 255.255.0.0 up
                ifconfig $eth1 $right_addr_ip2 netmask 255.255.0.0 up
                ifconfig $eth2 up
                ifconfig $eth3 up

                # set route on fm1-gb3/4
                route add -net 192.85.0.0 netmask 255.255.0.0 dev $eth2
                route add -net 192.87.0.0 netmask 255.255.0.0 dev $eth3

                k=2
                for i in `seq 2 9`
                do
                    arp -s 192.88.1.$i $right_src_mac2 -i $eth1
                    for j in `seq 2 5`
                    do
                    while (($k < 6))
                    do
                        arp -s 192.86.1.$k $right_src_mac1 -i $eth0
                        k=$((k + 1))
                    done
                    ip addr add 200.200.$base1.20/24 dev $eth2
                    echo ===base1 is $base1===
                    ip addr add 200.200.$base2.40/24 dev $eth3
                    echo ===base2 is $base2===

                    make_esp_policy $1 192.85.1.$i 192.86.1.$j 200.200.$base1.10
200.200.$base1.20
                    make_esp_policy $1 192.87.1.$j 192.88.1.$i 200.200.$base2.30
200.200.$base2.40
                    make_esp_tunnel 200.200.$base1.10 200.200.$base1.20 $base1 $proto
                    base1=$(($base1 + 1))
                    make_esp_tunnel 200.200.$base2.30 200.200.$base2.40 $(($base2 + 100)) $proto
                    base2=$(($base2 + 1))
                    done
                done
            ;;
esac
```

- Copy above file to board
- On T1040D4RDB-1:

bash linux_ipsec_4g.sh left

- On T1040D4RDB-2:

bash linux_ipsec_4g.sh right

- **Configuration for ASF IPv4 forward on T1040D4RDB:**
- Configure ASF modules

#cd /usr/driver/asf/min

#insmod asf.ko

#insmod asfctrl.ko

insmod asfipsec.ko

insmod asfctrl_ipsec.ko

#echo 9000 > /sys/asfctrl/ffp/asfctrl_ffp_udp_tmout

#echo 0 > /proc/sys/net/core/xfrm_larval_drop

#cd /usr/driver/asf/scripts/fmc

#fmc -c asf-cfg-perf-2041.xml -p asf-fman-perf-policy.xml -s Soft_FragParser.xml –a

#echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout

#echo 9000 > /proc/sys/net/netfilter/nf_conntrack_udp_timeout_stream

- Configure L2switch settings
- • Refer to **Configure L2switch for Linux IPv4 Forward**
- Configure network settings for fm1-gb0 – fm1-gb4
- Refer to **Configure network settings for Linux IPv4 Forward**

Running the test

After Sprient Testcenter application configure completely, start pump traffic to measure IPv4 forward throughput.

Appendix

- Check L2switch status:

#l2sw_bin

mac dump

check

port stats all show

- type “?” to get more information in l2sw_bin

12.2.4 IPSec Forward - eTSEC

How to setup a board to baseline eTSEC IPSec performance for QorIQ processor development boards.

12.2.4.1 Test Environment

Baseline non-DPAA platform performance.

Benchmarking Objective

The purpose of this guide is to show how to setup a Linux IPsec system on a NXP QorIQ processor based system and how to measure its performance.

NOTE

LS1021ATWR was used for the demo. Please refer to the board's manual for other boards.

Hardware Platform Identification

Non-DPAA platforms:

- LS1021ATWR

For board switch settings, please see the corresponding reference manual. Operating frequencies (CPU, CCB, DDR) must be set to the highest allowed values (check the manual).

Network Simulator

Spirent Test Center Performance Analysis System

Spirent Testcenter v4.41.

Target Development System Software

Software Platform Identification

All software was built from the QorIQ SDK ISO.

Boot Loader

U-boot - configuration:

- Enable cacheline interleaving mode: CS0+CS1
- Enable DDR controller interleaving mode: cache line (if there are two DDR controllers)

Operating system

The BSP distribution includes source code for Linux vanilla kernel 4.1.8 and NXP-specific kernel patches. This is the default kernel package for the SDK configuration.

Root filesystem considerations

The IPsec testing procedure makes use of the “arp”, “ifconfig” and “ip” utilities. These are part of the net-tools and iproute2 packages, which should be included in the root filesystem.

12.2.4.2 Test Procedure

The test procedure for various configurations of IPv4 Sec & Forwarding. The LS1021ATWR platform is used as an example.

Test Configuration

The configuration used is detailed in Figure 1.

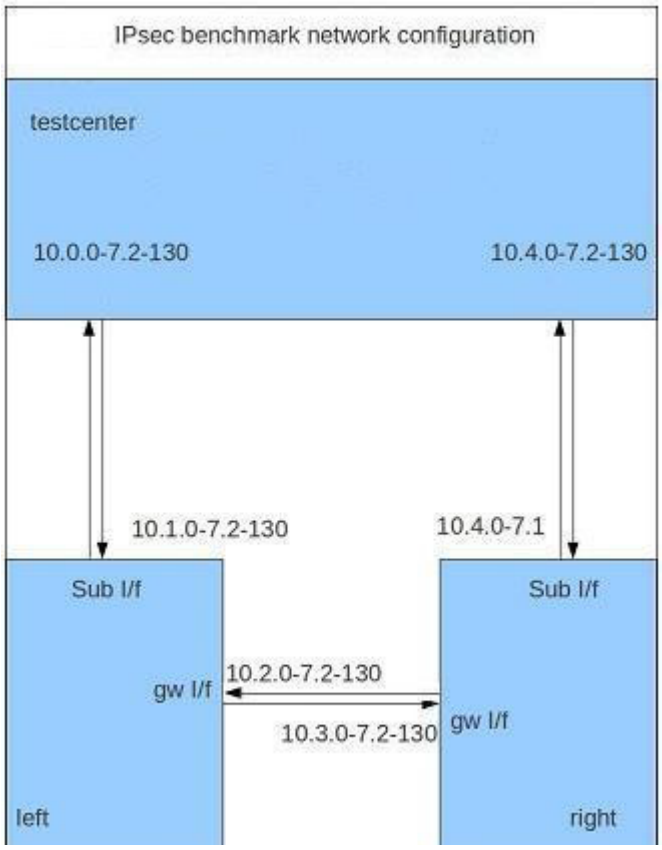


Figure 381. Multitunnel IPsec configuration

The number of flows is configurable. Please refer to the provided script (section "IPsec configuration script").

NOTE

In further sections, it is assumed that eth2 is used as subnet i/f, eth1 as gateway i/f and eth0 as boot i/f. If this is not the case, be careful to make all necessary changes (to the IPsec setup script, SMP affinities, etc.).

Spirent Test Center Configuration

- RFC2544 command sequencer (“Tools -> Wizards -> Benchmarking -> RFC2544”) and Test Center Reporter
- Performance-related parameters:

```
- Test Duration: 60 Seconds
- Resolution: 0.1%
- Packet Loss Rate: 0.001 %
- Frame Size: 82, 408, 1442 bytes
```

NOTE

The IPsec configuration script (section 4.1) sets the MTU to 2000 so that the 1518 bytes frames won’t be fragmented and reassembled.

Setup

- make sure that operating frequencies (CPU, CCB, DDR) are set to the highest allowed values (check the corresponding platform reference manual); for LS1021ATWR that would be (extracted from u-boot log):

```
CPU:   Freescale LayerScape LS1021E, Version: 2.0, (0x87081120)
Clock Configuration:
       CPU0(ARMV7):1000 MHz,
       Bus:300  MHz, DDR:800  MHz (1600 MT/s data rate)
```

- kernel configuration provided Linux kernel comes with a default configuration file,in order to obtain better performance, the following changes have to done:

```
CONFIG_NR_CPUS 2 (default: 4)
CONFIG_USB_SUPPORT disabled (default: builtin)
CONFIG_MMC disabled (default: builtin)
CONFIG_SOUND disabled (default: builtin)
```

- kernel boot parameters used:

```
root=/dev/ram rw console=ttyS0,115200 ramdisk_size=1000000 log_buf_len=128K
```

NOTE

Ramboot is preferred when performing benchmarking, since nfsboot might incur performance hit due to inherent network traffic

- kill all unneeded user space processes, for example:

```
/etc/init.d/sshd stop
/etc/init.d/xinetd stop
pid=`ps -ef | grep udhcpc | awk '{print $2}'` && kill -9 $pid
```

NOTE

The exact commands depend on the features included in the root filesystem.

- set hostname (needed so that we can use a single script for setting up IPsec on both boards)

```
o left board: hostname elmer
o right board: hostname donald
```

- set up IPsec by running the script provided in section "IPsec configuration script"
- set up SMP affinities for caam (due to the balanced job ring allocation mechanism, the settings might vary; below setup assumes there is an equal number of active sessions per job ring - such that Job Ring 0 is the first being picked - and that IPsec security associations are created alternatively, i.e. one SA for left-to-right direction, followed by one SA for right-to-left direction and so on)

```
o cat /proc/interrupts | grep jr
 74:          0          0       GIC 135 Level     1710000.jr
 75:          0          0       GIC 136 Level     1720000.jr
 76:          0          0       GIC 137 Level     1730000.jr
 77:          0          0       GIC 138 Level     1740000.jr

echo 1 > /proc/irq/74/smp_affinity
echo 1 > /proc/irq/76/smp_affinity
echo 2 > /proc/irq/75/smp_affinity
echo 2 > /proc/irq/77/smp_affinity
```

- set up SMP affinities for gianfar:

```
o cat /proc/interrupts | grep eth
 56:          0          0       GIC 176 Level     eth0_g0_tx
 57:          0          0       GIC 177 Level     eth0_g0_rx
 58:          0          0       GIC 178 Level     eth0_g0_er
 59:          0          0       GIC 179 Level     eth0_g1_tx
 60:          0          0       GIC 180 Level     eth0_g1_rx
 61:          0          0       GIC 181 Level     eth0_g1_er
 62:          0          0       GIC 182 Level     eth1_g0_tx
 63:          0          0       GIC 184 Level     eth1_g0_rx
 64:          0          0       GIC 185 Level     eth1_g0_er
 65:          0          0       GIC 186 Level     eth1_g1_tx
 66:          0          0       GIC 187 Level     eth1_g1_rx
 67:          0          0       GIC 188 Level     eth1_g1_er
 68:          0          0       GIC 189 Level     eth2_g0_tx
 69:         66          0       GIC 190 Level     eth2_g0_rx
 70:          0          0       GIC 191 Level     eth2_g0_er
 71:          0          0       GIC 192 Level     eth2_g1_tx
 72:          0          0       GIC 193 Level     eth2_g1_rx
 73:          0          0       GIC 194 Level     eth2_g1_er
o left board
echo 1 > /proc/irq/62/smp_affinity
echo 1 > /proc/irq/69/smp_affinity
echo 2 > /proc/irq/63/smp_affinity
echo 2 > /proc/irq/68/smp_affinity
o right board
echo 1 > /proc/irq/63/smp_affinity
echo 1 > /proc/irq/68/smp_affinity
echo 2 > /proc/irq/62/smp_affinity
echo 2 > /proc/irq/69/smp_affinity
```

NOTE

The SMP affinities setup is not identical on left and right board. The idea is to process left -> right flows on CPU0 and right -> left flows on CPU1.

- tune gianfar settings

```
ethtool -C eth1 rx-frames 12 rx-usecs 32
ethtool -C eth1 tx-frames 12 tx-usecs 32
ethtool -G eth1 rx 128 tx 128
ethtool -C eth2 rx-frames 12 rx-usecs 32
ethtool -C eth2 tx-frames 12 tx-usecs 32
ethtool -G eth2 rx 128 tx 128
```

- other optimizations

```
sysctl -w net.ipv4.conf.default.rp_filter=0
sysctl -w net.ipv4.conf.all.rp_filter=0
ifconfig eth0 down
```

Results

If performance is measured by using the TestCenter RFC 2544 Command Sequencer, results will be displayed in Spirent TestCenter Results Reporter.

IPSec Configuration Script

The script below creates a configurable number of IPsec tunnels (up to 1024), depending on the values of i_max and j_max. Currently, the script uses 3DES-CBC-HMAC-SHA1 as AEAD mode. This can be changed by editing the auth and enc parameters in mkstate() function. Also note that the network interfaces (left_sub_if, left_gw_if, right_sub_if, right_gw_if) and their MAC addresses must be modified according to your HW setup. MTU is set to 2000 bytes (instead of default 1500) so that 1500 bytes frames won't be fragmented (and reassembled at the other end).

```
#!/bin/bash
#ESP
function mkstate() {
    ip xfrm state add \
    src $2 dst $3 \
    proto esp spi $6 mode $5 \
    auth "hmac(sha1)" $8 \
    enc "cbc(des3_ede)" $7 \
    sel src 0.0.0.0/0 dst 0.0.0.0/0
}
function mkpolicy() {
    ip xfrm policy add \
    dir $6 src $1 dst $4 \
    tmpl src $2 dst $3 \
    proto esp mode $5
}
#
# usage: mktunnel src-netspec src-gw dst-gw dst-netspec
# e.g. mktunnel 192.168.2.0/24 192.168.3.1 192.168.3.2 192.168.4.2 <tunnel/transport>
#
function mktunnel() {
    base_lr=`echo $1-$2-$3-$4 | md5sum`
    base_rl=`echo $4-$3-$2-$1 | md5sum`

    # try and generate a %8x spi and gen keys for the 'left'.
    # seen complaints from leading zeros in an SPI, so cut them out, too.
    spi_lr=`echo $base_lr | tr -d 0 | tr -d a-f | cut -c -8`
    auth_key_lr=`echo $base_lr | cut -c -20`
    cipher_key_lr=`echo $base_lr | cut -c -24`
    mkstate $1 $2 $3 $4 $5 $spi_lr $cipher_key_lr $auth_key_lr
    # now the 'right'
    spi_rl=`echo $base_rl | tr -d 0 | tr -d a-f | cut -c -8`

    auth_key_rl=`echo $base_rl | cut -c -20`
    cipher_key_rl=`echo $base_rl | cut -c -24`
    mkstate $4 $3 $2 $1 $5 $spi_rl $cipher_key_rl $auth_key_rl
    ip addr show | awk '/inet/ {print $2}' | grep $2
    if [ $? == 0 ]
    then
        echo this host is the left
        mkpolicy $1 $2 $3 $4 $5 out
        mkpolicy $4 $3 $2 $1 $5 in
        mkpolicy $4 $3 $2 $1 $5 fwd
    else
        echo this host is the right
        mkpolicy $1 $2 $3 $4 $5 in
        mkpolicy $1 $2 $3 $4 $5 fwd
        mkpolicy $4 $3 $2 $1 $5 out
    fi
}

echo 1 > /proc/sys/net/ipv4/ip_forward
```

```
ip xfrm state flush
ip xfrm policy flush

case `hostname` in
    left|elmer) # with donald, elmer is left
        left_sub_if=eth2
        left_sub_net=1
        left_gw_if=eth1
        left_gw_net=2
        right_gw_net=3
        right_sub_net=4
        right_gw_mac=00:04:9f:01:1e:63
        tc1_rx_mac=00:04:9f:ff:15:01
        ifconfig $left_sub_if mtu 2000 up
        ifconfig $left_gw_if mtu 2000 up
        i=0; i_max=1 # i_max == 7 (1024 tunnels) exhausts the arp tables - how to reconfigure
them?
        while [ "$i" != "$i_max" ] ; do
            echo tunnelling 10.y.$i.x
            j=2; j_max=130
            while [ "$j" != "$j_max" ] ; do
                ip addr add 10.$left_sub_net.$i.$j dev $left_sub_if
                ip addr add 10.$left_gw_net.$i.$j dev $left_gw_if
                ip route add 10.0.$i.$j/32 via 10.$left_sub_net.$i.$j
                ip route add 10.$right_gw_net.$i.$j/32 via 10.$left_gw_net.$i.$j
                ip route add 10.$right_sub_net.$i.$j/32 via 10.$left_gw_net.$i.$j
                arp -i $left_gw_if -s 10.$right_gw_net.$i.$j $right_gw_mac
                arp -i $left_sub_if -s 10.0.$i.$j $tc1_rx_mac
                mktunnel 10.0.$i.$j/32 10.$left_gw_net.$i.$j 10.$right_gw_net.$i.$j
10.$right_sub_net.$i.$j/32 tunnel
                ((j+=1))
            done
            ((i+=1))
        done
    exit
    ;;

    right|donald) # with elmer, donald is right
        right_sub_if=eth2
        right_gw_if=eth1
        left_sub_net=1
        left_gw_net=2
        right_gw_net=3
        right_sub_net=4
        left_gw_mac=00:04:9f:00:15:01
        tc2_rx_mac=00:04:9f:ff:15:03
        ifconfig $right_sub_if mtu 2000 up
        ifconfig $right_gw_if mtu 2000 up
        i=0; i_max=1 # i_max == 7 (1024 tunnels) exhausts the arp tables - how to reconfigure
them?
        while [ "$i" != "$i_max" ] ; do
            ip addr add 10.$right_sub_net.$i.1 dev $right_sub_if
            j=2; j_max=130
            while [ "$j" != "$j_max" ] ; do
                ip addr add 10.$right_gw_net.$i.$j dev $right_gw_if
                ip route add 10.$left_gw_net.$i.$j/32 via 10.$right_gw_net.$i.$j
                ip route add 10.$right_sub_net.$i.$j/32 via 10.$right_sub_net.$i.1
                ip route add 10.0.$i.$j/32 via 10.$right_gw_net.$i.$j
                arp -i $right_sub_if -s 10.5.$i.$j $tc2_rx_mac
                arp -i $right_gw_if -s 10.$left_gw_net.$i.$j $left_gw_mac
```

```
                mktunnel 10.0.$i.$j/32 10.$left_gw_net.$i.$j 10.$right_gw_net.$i.$j
10.$right_sub_net.$i.$j/32 tunnel
                ((j+=1))
            done
            ((i+=1))
        done
    exit
    ;;
esac
```

12.2.5 Netperf

Provides details for reproducing NetPerf performance results. The objectives are to baseline Netperf performance and identify optimizations and implement these improvements on the QorIQ DPAA platform.

12.2.5.1 Test Environment

Benchmarking Objectives

The purpose of this guide is to show how to setup a Netperf system on a NXP QorIQ processor based system and how to measure its performance.

NOTE

P4080DS was used for the demo. Please refer to the board's manual for other boards.

Platform Configurations

Table 605. Development Platform Clocking

Platform	Core/CPU (MHz)	CCB (MHz)	DDR (MHz)	FMan (MHz)
P2040RDB	1200	600	600	500
P3041DS	1500	750	667	583
P4080DS	1500	800	650	600
P5020DS	2000	800	667	600
P5040DS	2267	800	800	600
B4860QDS	1600	667	933	667
T4240QDS	1800	733	933	733
T2080QDS	1800	600	933	700
T2080RDB	1800	600	933	700
T1024RDB	1400	400	800	700
T1023RDB	1200	400	800	600
T1040D4RDB	1400	600	800	700

Hardware Platform Identification

Table 606. Hardware Platform Identification

--	P2040	P3041	P4080	P5020	P5040	B4860	T4240	T2080	T1040D4	T1024	T1023
Board	P2040RDB	P3041DS	P4080DS	P5020DS	P5040DS	B4860QDS	T4240QDS	T2080QDS	T1040D4RDB	T1024RDB	T1023RDB
1G Ethernet	3 on-board SGMII ports and 2 on-board RGMII ports	1 SGMII riser card with 3 Ethernet ports and 2 on-board RGMII ports	1 SGMII riser card with 4 Ethernet ports	1 SGMII riser card with 3 Ethernet ports and 2 on-board RGMII ports	2 SGMII riser card with 2 Ethernet ports on each and 2 on-board RGMII port	2 on-board SGMII ports	2 on-board RGMII ports	--	2 on-board RGMII ports	2 on-board RGMII ports	1 on-board RGMII port and 1 on-board SGMII port
XAUI/XFI/10G BaseT	--	1 XAUI riser card	2 XAUI riser cards	1 XAUI riser card	2 XAUI riser cards	2 XAUI riser cards	4 XAUI riser cards	2 XFI cards	--	1 10G BaseT	--
2.5G SGMII	--	--	--	--	--	--	--	--	--	1 2.5G SGMII port	1 2.5G SGMII port
PCIe	--	e1000 NIC	--	e1000 NIC	e1000 NIC	--	e1000 NIC	--	--	--	--

Table 607. Software Platform Identification

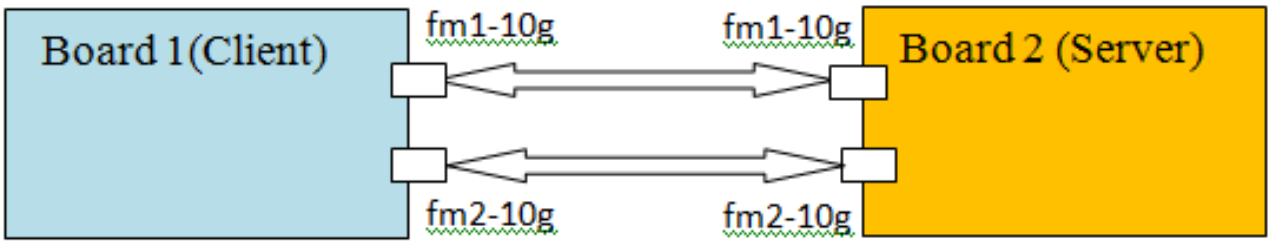
System Software	Description
Host Development System Software	Yocto
Target Development System: Boot Loader	U-boot 2016.01 with default ddr configurations
Target Development System: OS	• Kernel built based on default kernel configuration and enable CONFIG_DPAA_ETH_OPTIMIZE_FOR_TERM option • Default ramdisk rootfs coming with SDK v2.0 ISO
Application	The tests procedure for Netperf makes use of the "netserver", "netperf" and “ifconfig” utilities. These are part of the netperf and net-tools packages respectively, which are included the root file system.

12.2.5.2 Test Procedure

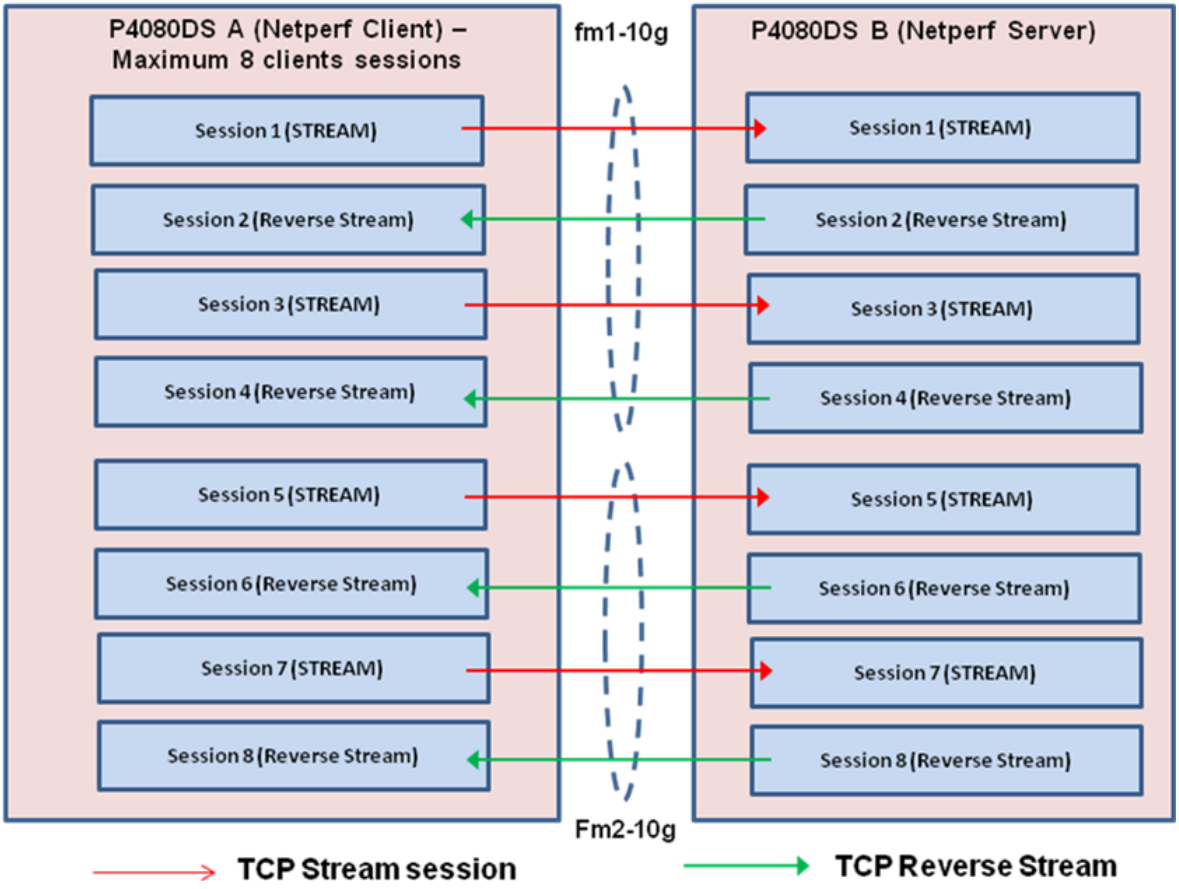
The P4080 was the first NXP SoC to incorporate the DPAA. As such, it is used in this examples. However, test methodology is intended to apply in the same manner to all DPAA-based SoCs.

Test Procedure: Test Bed Description

The test environment comprises two systems running Linux to be connected back-to-back. One is running Netperf server and the other is running Netperf client. The Netperf client is P4080DS and the Netperf server is another P4080DS or a Linux PC. The figure below shows two P4080DS connected back-to-back:



The software setup for the P4080DS is shown in the figure below:



Running Test and Result Collection: P4080DS or Linux PC Netperf Server Example

P4080DS or Linux PC works as Netperf server (netserver). Steps are as followings:

1. Set IP address

```
# ifconfig fm1-10g:1 1.0.0.1
# ifconfig fm1-10g:2 2.0.0.1
```

```
# ifconfig fm1-10g:3 3.0.0.1
# ifconfig fm1-10g:4 4.0.0.1
# ifconfig fm2-10g:5 5.0.0.1
# ifconfig fm2-10g:6 6.0.0.1
# ifconfig fm2-10g:7 7.0.0.1
# ifconfig fm2-10g:8 8.0.0.1
```

2. Configure fmc PCD files: (not required for Linux PC)

```
# fmc -c cfg_P4080DS_20g.xml -p policy_P4080DS.xml -a
```

3. Start netserver:

```
# netserver -L 1.0.0.1 0 -p 30001
# netserver -L 2.0.0.1 0 -p 30002
# netserver -L 3.0.0.1 0 -p 30003
# netserver -L 4.0.0.1 0 -p 30004
# netserver -L 5.0.0.1 0 -p 30005
# netserver -L 6.0.0.1 0 -p 30006
# netserver -L 7.0.0.1 0 -p 30007
# netserver -L 8.0.0.1 0 -p 30008
```

Running Test and Result Collection: P4080DS Netperf Client Mode Example

Another P4080DS works as Netperf client (netperf). Steps are as followings:

1. Set IP address

```
# ifconfig fm1-10g:1 1.0.0.2
# ifconfig fm1-10g:2 2.0.0.2
# ifconfig fm1-10g:3 3.0.0.2
# ifconfig fm1-10g:4 4.0.0.2
# ifconfig fm2-10g:5 5.0.0.2
# ifconfig fm2-10g:6 6.0.0.2
# ifconfig fm2-10g:7 7.0.0.2
# ifconfig fm2-10g:8 8.0.0.2
```

2. Configure fmc PCD files:

```
# fmc -c cfg_P4080DS_20g.xml -p policy_P4080DS.xml -a
```

3. Start netperf on client:

```
# netperf -Cc -H 1.0.0.1 -l 60 -p 30001 -t TCP_STREAM -T 1,1 -P 0 -B "outbound" -- -m 16384 &
# netperf -Cc -H 2.0.0.1 -l 60 -p 30002 -t TCP_MAERTS -T 2,2 -P 0 -B "inbound" -- -m 16384 &
# netperf -Cc -H 3.0.0.1 -l 60 -p 30003 -t TCP_STREAM -T 3,3 -P 0 -B "outbound" -- -m 16384 &
# netperf -Cc -H 4.0.0.1 -l 60 -p 30004 -t TCP_MAERTS -T 4,4 -P 0 -B "inbound" -- -m 16384 &
# netperf -Cc -H 5.0.0.1 -l 60 -p 30005 -t TCP_STREAM -T 5,5 -P 0 -B "outbound" -- -m 16384 &
# netperf -Cc -H 6.0.0.1 -l 60 -p 30006 -t TCP_MAERTS -T 6,6 -P 0 -B "inbound" -- -m 16384 &
# netperf -Cc -H 7.0.0.1 -l 60 -p 30007 -t TCP_STREAM -T 7,7 -P 0 -B "outbound" -- -m 16384 &
# netperf -Cc -H 8.0.0.1 -l 60 -p 30008 -t TCP_MAERTS -T 0,0 -P 0 -B "inbound" -- -m 16384 &
```

NOTE: replace the value ‘64’ of ‘-m’ option with below values to test other send message sizes:
64/128/256/512/1024/1440/1518/4172/9576/16384/32768/65535

To test default send message size, just remove the ‘-m 64’ option.

12.2.6 DPAA NAT and Firewall

How to setup the board to reproduce NAT and Firewall benchmark results for the QorIQ SDK boards.

12.2.6.1 Hardware

Benchmarking Objectives

The purpose of this guide is to show how to setup a NAT and Firewall system on a NXP QorIQ processor based system and how to measure its performance.

NOTE

P4080DS was used for the demo. Please refer to the board's manual for other boards.

The relevant QorIQ development systems covered

- P2040RDB
- P3041DS
- P4080DS (focus of this example)
- P5020DS
- P5040DS
- B4860QDS
- T4240QDS
- T4240RDB
- T2080QDS
- T2080RDB
- T1023RDB
- T1024RDB
- T1040RDB
- T1042RDB

Hardware Platform Identification

1. **P2040RDB**
   - Board: P2040RDB, CPLD version: 2.2
   - CPU: P2040E, Version: 2.0
   - Core: E500MC, Version: 3.2
   - Clock Configuration:
     - CPU0: 1200 MHz, CPU1:1200 MHz, CPU2:1200 MHz, CPU3:1200 MHz,
     - CCB: 600 MHz,
     - DDR: 600 MHz (1200 MT/s data rate) (Asynchronous), LBC:75 MHz
     - FMAN1: 500 MHz
   - 3 on-board SGMII ports and 2 on-board RGMII ports
2. **P3041DS**
   - Board: P3041DS, Sys ID: 0x1c, Sys Ver: 0x11, FPGA Ver: 0x05

- CPU: P3041E, Version: 2.0
- Core: E500MC, Version: 2.2
- Clock Configuration:
  - CPU0: 1500 MHz, CPU1:1500 MHz, CPU2:1500 MHz, CPU3:1500 MHz
  - CCB: 750 MHz
  - DDR: 666.667 MHz (1333.333 MT/s data rate) (Asynchronous), LBC:93.750 MHz
  - FMAN1: 583.333 MHz
- 1SGMII riser card with 3 Ethernet ports and 2 on-board RGMII ports
- 1 10G XAUI riser card
- PCIe e1000 NIC

3. **P4080DS**
   - Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0c
   - CPU: P4080E, Version: 3.0
   - Core: E500MC, Version: 3.1
   - Clock Configuration:
     - CPU0: 1499.985 MHz, CPU1:1499.985 MHz, CPU2:1499.985 MHz, CPU3:1499.985 MHz
     - CPU4: 1499.985 MHz, CPU5:1499.985 MHz, CPU6:1499.985 MHz, CPU7:1499.985 MHz
     - CCB: 799.992 MHz
     - DDR: 649.994 MHz (1299.987 MT/s data rate) (Asynchronous), LBC:99.999 MHz
     - FMAN1: 599.994 MHz
     - FMAN2: 599.994 MHz
   - 1 SGMII riser card with 4 Ethernet ports
   - 2 10G XAUI riser cards
4. **P5020DS**
   - Board: P5020DS, Sys ID: 0x1c, Sys Ver: 0x01, FPGA Ver: 0x03
   - CPU: P5020E, Version: 2.0
   - Core: E5500, Version: 1.2
   - Clock Configuration:
     - CPU0: 2000 MHz, CPU1:2000 MHz
     - CCB: 800 MHz
     - DDR: 666.667 MHz (1333.333 MT/s data rate) (Asynchronous), LBC:100 MHz
     - FMAN1: 600 MHz
   - 1 SGMII riser card with 3 Ethernet ports and 2 on-board RGMII ports
   - 1 10G XAUI riser card
   - PCIe e1000 NIC
5. **P5040DS**
   - Board: P5040DS, Sys ID: 0x20, Sys Ver: 0x02, FPGA Ver: 0x02
   - CPU: P5040E, Version: 2.1

- Core: E5500, Version: 1.2
- Clock Configuration:
  - CPU0: 2266.667 MHz, CPU1:2266.667 MHz, CPU2:2266.667 MHz, CPU3:2266.667 MHz
  - CCB: 800 MHz
  - DDR: 800 MHz (1600 MT/s data rate) (Asynchronous), LBC:100 MHz
  - FMAN1: 600 MHz
  - FMAN2: 600 MHz
- 2 SGMII riser cards with 2 Ethernet ports on each and 2 on-board RGMII ports
- 2 10G XAUI riser cards
- PCIe e1000 NIC

6. **B4860QDS**
   - Board: B4860QDS, Sys ID: 0x21, Sys Ver: 0x12
   - CPU: B4860E, Version: 2.2
   - Core: E6500, Version: 2.0
   - Clock configuration:
     - CPU0: 1600 MHz, CPU1:1600 MHz, CPU2:1600 MHz, CPU3:1600 MHz
     - CCB: 666.667 MHz
     - DDR: 933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:166.667 MHz
     - FMAN1: 666.667 MHz
   - 2 on-board SGMII ports
   - 2 10G XFI ports
7. **T4240QDS**
   - Board: T4240QDS, Sys ID: 0x1e, Sys Ver: 0x22
   - CPU: T4240E, Version: 2.0
   - Core: E6500, Version: 2.0
   - Clock configuration:
     - CPU0: 1666.667 MHz, CPU1:1666.667 MHz, CPU2:1666.667 MHz, CPU3:1666.667 MHz
     - CPU4: 1666.667 MHz, CPU5:1666.667 MHz, CPU6:1666.667 MHz, CPU7:1666.667 MHz
     - CPU8: 1666.667 MHz, CPU9:1666.667 MHz, CPU10:1666.667 MHz, CPU11:1666.667 MHz
     - CCB: 733.333 MHz
     - DDR: 933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:183.333 MHz
     - FMAN1: 733.333 MHz
     - FMAN2: 733.333 MHz
   - 2 on-board RGMII ports
   - 4 10G XAUI riser cards
   - PCIe e1000 NIC
8. **T4240RDB**
   - Board: T4240RDB, Board rev: 0x04 CPLD ver: 0x0401

- CPU: T4240E, Version: 2.0
- Core: E6500, Version: 2.0
- Clock configuration:
  - CPU0: 1666.667 MHz, CPU1:1666.667 MHz, CPU2:1666.667 MHz, CPU3:1666.667 MHz
  - CPU4: 1666.667 MHz, CPU5:1666.667 MHz, CPU6:1666.667 MHz, CPU7:1666.667 MHz
  - CPU8: 1666.667 MHz, CPU9:1666.667 MHz, CPU10:1666.667 MHz, CPU11:1666.667 MHz
  - CCB: 733.333 MHz
  - DDR: 933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:183.333 MHz
  - FMAN1: 733.333 MHz
  - FMAN2: 733.333 MHz
- 8 on-board SGMII ports
- 4 10G XFI ports
- PCIe e1000 NIC

9. **T2080QDS**
   - Board: T2080QDS, Sys ID: 0x28, Board Arch: V1, Board Version: A
   - CPU: T2080E, Version: 1.1
   - Core: E6500, Version: 2.0
   - Clock configuration:
     - CPU0: 1800 MHz, CPU1:1800 MHz, CPU2:1800 MHz, CPU3:1800 MHz
     - CCB: 700 MHz
     - DDR: 933.333 MHz (1866.667 MT/s data rate) (Asynchronous), IFC:175 MHz
     - FMAN1: 700 MHz
   - 2 on-board RGMII ports
   - 4 10G XFI ports
   - PCIe e1000 NIC
10. **T2080RDB**
    - Board: T2080RDB, Board rev: 0x01 CPLD ver: 0x03
    - CPU: T2080E, Version: 1.1
    - Core: E6500, Version: 2.0
    - Clock configuration:
      - CPU0: 1799.820 MHz, CPU1:1799.820 MHz, CPU2:1799.820 MHz, CPU3:1799.820 MHz
      - CCB: 599.940 MHz
      - DDR: 799.980 MHz (1599.960 MT/s data rate) (Asynchronous), IFC:149.985 MHz
      - FMAN1: 699.930 MHz
    - 2 on-board RGMII ports
    - 2 10G XFI ports
    - 2 10G BaseT ports
    - PCIe e1000 NIC

11. **T1023RDB**
    - Board: T1023RDB
    - CPU: T1023E, Version: 1.0
    - Core: E5500, Version: 2.1
    - Clock configuration:
      - CPU0: 1400 MHz, CPU1:1400 MHz
      - CCB: 400 MHz
      - DDR: 800 MHz (1600 MT/s data rate) (Asynchronous), IFC:100 MHz
      - FMAN1: 700 MHz
    - 1 on-board RGMII ports
    - 1 on-board SGMII ports
    - 1 2.5G SGMII port
    - PCIe e1000 NIC
12. **T1024RDB**
    - Board: T1024RDB, Board rev: 0x01 CPLD ver: 0x02
    - CPU: T1024E, Version: 1.0
    - Core: E5500, Version: 2.1
    - Clock configuration:
      - CPU0: 1400 MHz, CPU1:1400 MHz
      - CCB: 400 MHz
      - DDR: 800 MHz (1600 MT/s data rate) (Asynchronous), IFC:100 MHz
      - FMAN1: 700 MHz
    - 2 on-board RGMII ports
    - 1 10G BaseT port or 1 2.5G SGMII port
    - PCIe e1000 NIC
13. **T1040RDB**
    - Board: T1040RDB, Board rev: 0x01 CPLD ver: 0x06
    - CPU: T1040E, Version: 1.1
    - Core: E5500, Version: 2.1
    - Clock configuration:
      - CPU0: 1400 MHz, CPU1:1400 MHz, CPU2:1400 MHz, CPU3:1400 MHz
      - CCB: 600 MHz
      - DDR: 800 MHz (1600 MT/s data rate) (Asynchronous), IFC:150 MHz
      - FMAN1: 600 MHz
    - 2 on-board RGMII ports and 1 on-board SGMII ports
    - 8 QSGMII ports for L2-Switch
    - PCIe e1000 NIC
14. **T1042RDB**

- Board: T1042RDB, Board rev: 0x01 CPLD ver: 0x08
- CPU: T1042E, Version: 1.1
- Core: E5500, Version: 2.1
- Clock configuration:
  - CPU0: 1200 MHz, CPU1:1200 MHz, CPU2:1200 MHz, CPU3:1200 MHz
  - CCB: 500 MHz
  - DDR: 800 MHz (1600 MT/s data rate) (Asynchronous), IFC:125 MHz
  - FMAN1: 500 MHz
- 2 on-board RGMII ports
- PCIe e1000 NIC

Network Simulator

- Spirent Testcenter version: 4.41
- Choose Bound Stream Block for Traffic generator
- Choose Wizard -> Custom Test -> Throughput from Spirent Testcenter to create command sequencer for throughput test
- Testcenter configuration:
  - Test Duration: 60 Seconds
  - Resolution: 0.1%
  - Packet Loss Rate: 0.001 %
  - Packet Size: 64, 128, 256, 512, 1024, 1280, 1518 bytes

Target Development System Software

Boot loader

Default u-boot can be used for this benchmark test.

RCW

- P2040RDB: RR_PH_0x19/rcw_5g_1200mhz.bin
- P3041DS: RR_HXAPNSP_0x36/rcw_15g_1500mhz.bin
- P4080DS: R_PPSXX_0xe/rcw_2sgmii_1500mhz_rev3.bin
- P5020DS: RR_HXAPNSP_0x36/rcw_15g_2000mhz.bin
- P5040DS: RR_XXSNSpP_0x02/rcw_26g_2267mhz.bin
- B4860QDS: N_RSSS_0x2A_0x8D/rcw_4sgmii_4srio_2xfi_1600mhz_1866ddr_rev2.bin
- T4240QDS: RR_XXXXPRPR_1_1_5_5/rcw_1_1_5_5_1666MHz_rev2.bin
- T4240RDB: SSFFPPH_27_55_1_9/rcw_27_55_1_9_1666MHz.bin
- T2080QDS: RR_PNNPPH_66_15/rcw_66_15_1800MHz_rev11.bin
- T2080RDB: RRFFXX_P_66_15/rcw_66_15_1800MHz_rev11.bin
- T1023RDB: RNSS_NPP_77/rcw_77_1400MHz.rcw
- T1024RDB: RRX_PPP_95/rcw_95_1400MHz.bin
- T1040RDB: RR_P_66/rcw_1400MHz.bin
- T1042RDB: RR_P_86/rcw_1400MHz.bin

FMAN ucode

The SDK ISO contains the default ucode versions for all the supported platforms. Please refer to the “Linux Kernel and Device Drivers” section from the “Components” documentation chapter.

Linux Kernel

The default kernel in ISO doesn't enable NAT/Firewall feature, so it need manually configure kernel. The steps for building the kernel is listed below.

Additional root filesystem considerations

The tests procedure for NAT/Firewall makes use of the "arp", "ifconfig" and “ip” utilities. These are part of the net-tools and iproute2 packages, which the root filesystem stored in the hard disk should have.

12.2.6.2 Kernel

Kernel Configuration to add support for NAT and Firewall.

Kernel Configuration to add support for NAT and Firewall

Kernel Configure Tree View Options

```
[*]Networking support --->
  [*] Networking options --->
      [*] Network packet filtering framework (Netfilter)  --->
      [ ]   Network packet filtering debugging
      [*]   Advanced netfilter configuration
      Core Netfilter Configuration  --->
            -*- Netfilter NFACCT over NFNETLINK interface
            -*- Netfilter NFQUEUE over NFNETLINK interface
            -*- Netfilter LOG over NFNETLINK interface
            <*> Netfilter connection tracking support
            -*-   Connection mark tracking support
            [*]   Connection tracking zones
            [*]   Connection tracking events
            [*]   Connection tracking timeout
            [*]   Connection tracking timestamping
            <*>   DCCP protocol connection tracking support (EXPERIMENTAL)
            <*>   SCTP protocol connection tracking support (EXPERIMENTAL)
            <*>   UDP-Lite protocol connection tracking support
            <*>   Amanda backup protocol support
            <*>   FTP protocol support
            <*>   H.323 protocol support
            <*>   IRC protocol support
            <*>   NetBIOS name service protocol support
            < >   SNMP service protocol support
            <*>   PPtP protocol support
            <*>   SANE protocol support (EXPERIMENTAL)
            <*>   SIP protocol support
            <*>   TFTP protocol support
            <*>   Connection tracking netlink interface
            <*>   Connection tracking timeout tuning via Netlink
            <*>   Connection tracking helpers in user-space via Netlink
            [*]   NFQUEUE integration with Connection Tracking
            <*> Transparent proxying support (EXPERIMENTAL)
            -*- Netfilter Xtables support (required for ip_tables)
                  *** Xtables combined modules ***
            -*-   nfmark target and match support
            -*-   ctmark target and match support
                  *** Xtables targets ***
```

```
        <*>   AUDIT target support
        <*>   CHECKSUM target support
        <*>   "CLASSIFY" target support
        <*>   "CONNMARK" target support
        -*-   "CT" target support
        <*>   "DSCP" and "TOS" target support
        -*-   "HL" hoplimit target support
        <*>   "HMARK" target support
        <*>   IDLETIMER target support
        <*>   LOG target support
        <*>   "MARK" target support
        -*-   "NETMAP" target support
        <*>   "NFLOG" target support
        <*>   "NFQUEUE" target Support
        <*>   "NOTRACK" target support
        -*-   "RATEEST" target support
        -*-   REDIRECT target support
        <*>   "TEE" - packet cloning to alternate destination
        <*>   "TPROXY" target support (EXPERIMENTAL)
        <*>   "TRACE" target support
        <*>   "TCPMSS" target support
        <*>   "TCPOPTSTRIP" target support (EXPERIMENTAL)
              *** Xtables matches ***
        <*>   "addrtype" address type match support
        <*>   "cluster" match support
        <*>   "comment" match support
        <*>   "connbytes" per-connection counter match support
        <*>   "connlimit" match support"
        <*>   "connmark" connection mark match support
        <*>   "conntrack" connection tracking match support
        < >   "cpu" match support
        <*>   "dccp" protocol match support
        < >   "devgroup" match support
        <*>   "dscp" and "tos" match support
        -*-   "ecn" match support
        <*>   "esp" match support
        <*>   "hashlimit" match support
        <*>   "helper" match support
        -*-   "hl" hoplimit/TTL match support
        <*>   "iprange" address range match support
        <*>   "length" match support
        <*>   "limit" match support
        <*>   "mac" address match support
        <*>   "mark" match support
        <*>   "multiport" Multiple port match support
        <*>   "osf" Passive OS fingerprint match
        <*>   "owner" match support
        <*>   IPsec "policy" match support
        <*>   "pkttype" packet type match support
        <*>   "quota" match support
        <*>   "rateest" match support
        <*>   "realm" match support
        <*>   "recent" match support
        <*>   "sctp" protocol match support (EXPERIMENTAL)
        <*>   "socket" match support (EXPERIMENTAL)
        <*>   "state" match support
        <*>   "statistic" match support
        <*>   "string" match support
        <*>   "tcpmss" match support
        <*>   "time" match support
```

```
            <*>   "u32" match support
          < >   IP set support (NEW)  --->
          < >   IP virtual server support (NEW)  --->
      IP: Netfilter Configuration  --->
            <*> IPv4 connection tracking support (required for NAT)
            <*> IP Userspace queueing via NETLINK (OBSOLETE)
            <*> IP tables support (required for filtering/masq/NAT)
            <*>   "ah" match support
            <*>   "ecn" match support
            < >   "rpfilter" reverse path filter match support
            <*>   "ttl" match support
            <*>   Packet filtering
            <*>     REJECT target support
            <*>   LOG target support
            <*>   ULOG target support
            <*>   IPv4 NAT
            <*>     MASQUERADE target support
            <*>     NETMAP target support
            <*>     REDIRECT target support
            <*>     Basic SNMP-ALG support (NEW)
            <*>   Packet mangling
            <*>     CLUSTERIP target support (EXPERIMENTAL)
            <*>     ECN target support
            <*>     "TTL" target support
            <*>   raw table support (required for NOTRACK/TRACE)
            <*> ARP tables support
            <*>   ARP packet filtering
            <*>   ARP payload mangling
```

Prepare ramdisk

Select the following packages when build ramdisk:

- `[*] busybox(busybox.config)`
- `[*] ethtool`
- `[*] net-tools`
- `[*] iptables`

Make sure BusyBox Configuration includes following changes under Networking Utilities:

- `[*]arping`
- `[*]ftpget`
- `[*]ftpput`
- `[*]hostname`
- `[*]ifconfig`
- `[*]Enable status reporting output (+7k)`

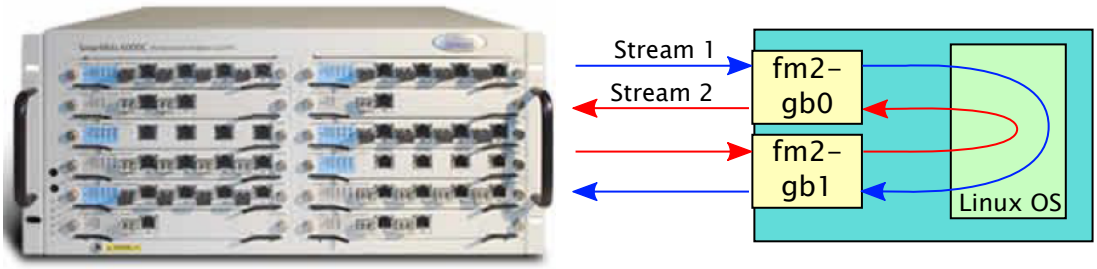
- `[*]Enable option "hw" (ether only)`
- `[*]ifupdown`
- `[*]arping`
- `[*]Use ip applet`
- `[*]tftp`
- `[*]Enable "get" command`
- `[*]Enable "put" command`
- `[*]ifupdown`

12.2.6.3 Test Setup

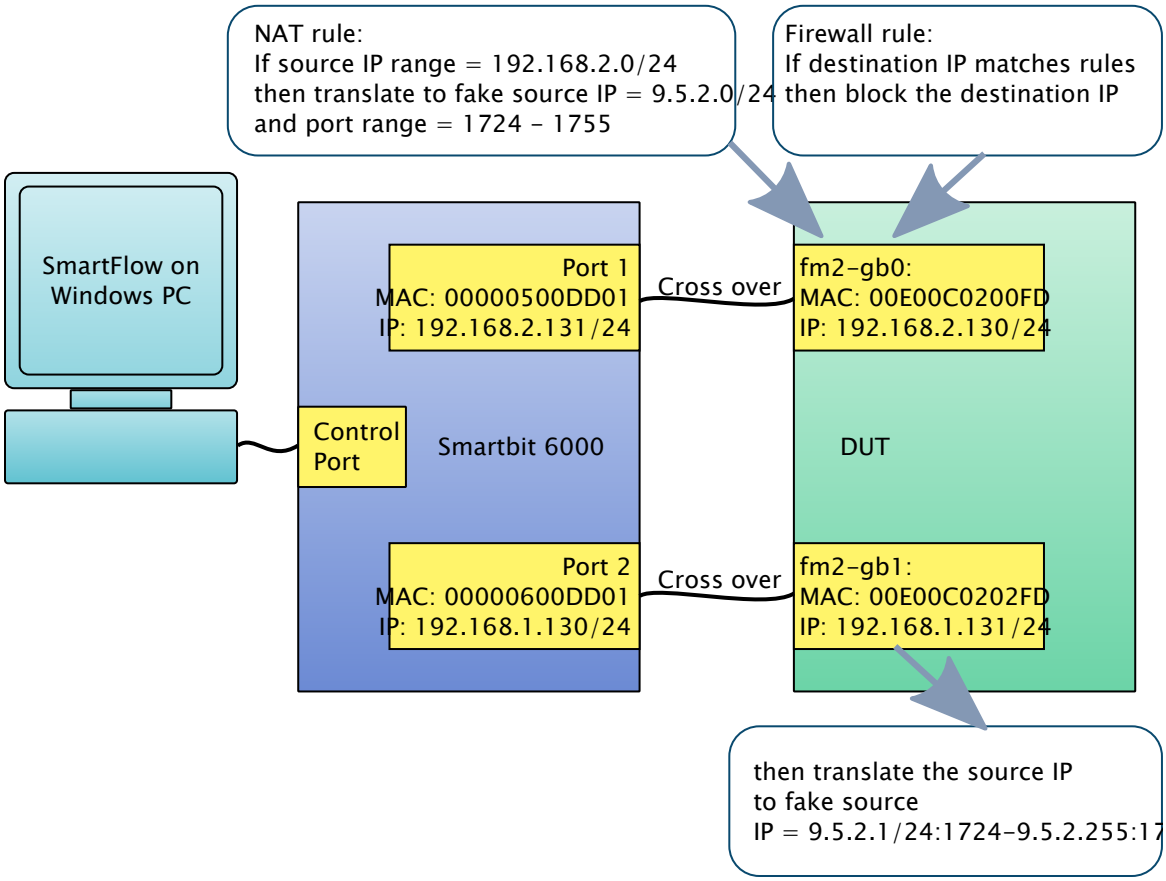
Running the DPAA performance benchmark test

Typical Test Setup

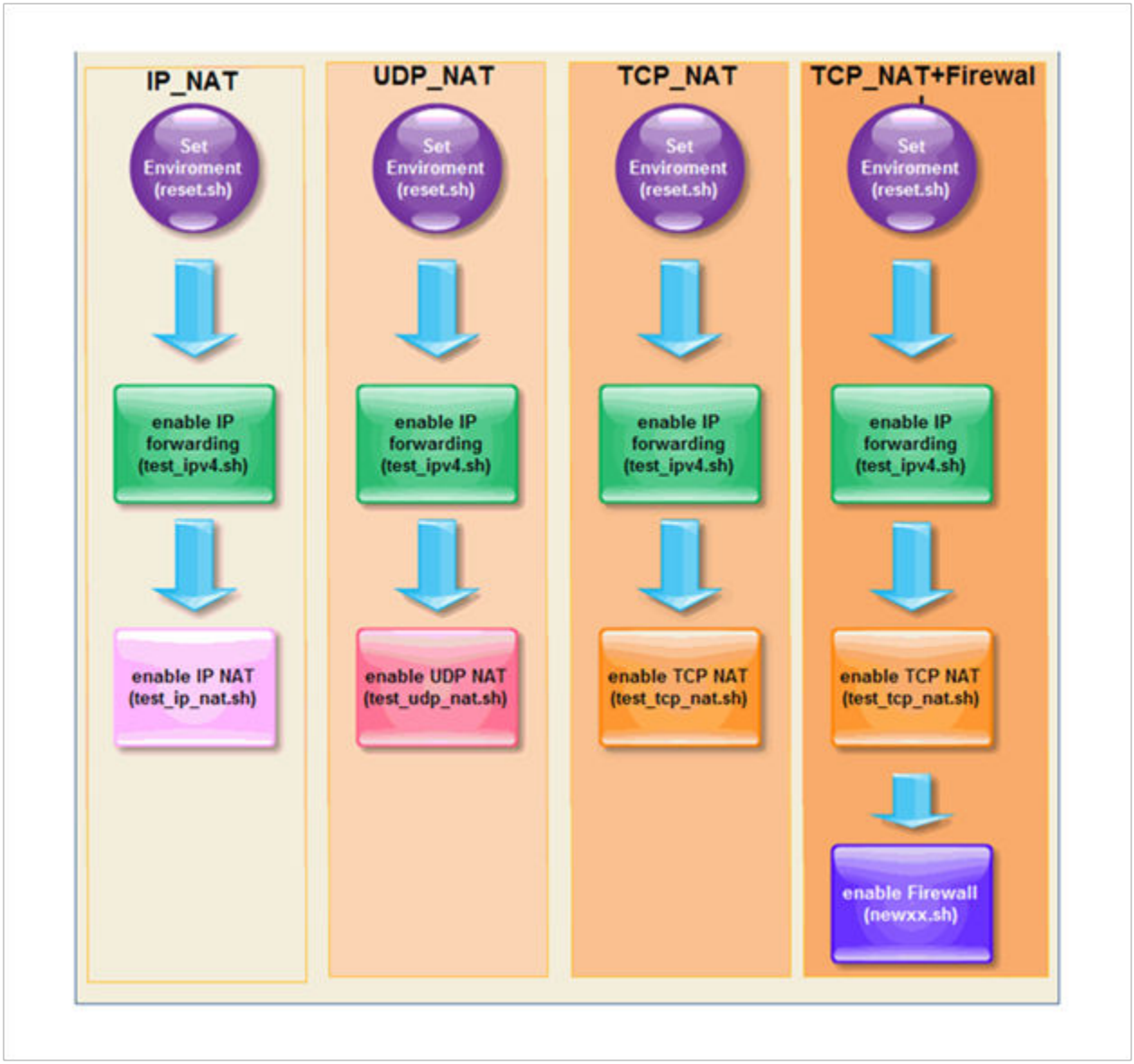
- Uni-directional (1 flow) = Stream 1
- Bi-directional (2 flow) = Stream 1 + Stream 2



NAT and Firewall Test Setup



Test Procedure



- Use reset.sh to set the environment which is included Source IP range, Fake IP range, income network device, output network device, then flush all rules in those tables and set the default policy for the tables.
- Use `test_ipv4.sh` to set QorIQ_USDK board IP address, do smartbit MAC and IP binding, and then enable IP Forwarding.
- Use `test_xxx_NAT.sh` to enable NAT translate.
- Use `newxxx.sh` to enable Firewall. `New1.sh` means 1 rule, `new128.sh` means 128 rules, `new1024.sh` means 1024 rules.

Test Tips

We can use the following commands to check if the setting is correct.:

```
iptable -L -t nat     #check the rule for the nat

tcpdump -i fm2-gb0    #dump the output interface to check the source ip addr was changed according
the nat rule.
```

```
iptable -L            #check the filter rules we created.
```

12.2.6.4 Test Demos

Example Test Demo and scripts to setup the board and to reproduce NAT and Firewall benchmark results.

Test Demos

Example Test Demo and scripts to setup the board and to reproduce NAT and Firewall benchmark results are provided below.

IPV4 Forwarding + TCP NAT:

```
!!!!! WARNING !!!!!!!
The default password for the root account is: root
please change this password using the 'passwd' command
and then edit this message (/etc/issue) to remove this message

p4080 login: root
Password:
[root@p4080 root]#
[root@p4080 root]# ifconfig fm2-gb0 192.168.2.130 netmask 255.255.255.0 up
[root@p4080 root]# ifconfig fm2-gb1 192.168.1.131 netmask 255.255.255.0 up
[root@p4080 root]# ifconfig
fm2-gb0 Link encap:Ethernet HWaddr 00:E0:0C:00:AA:05
        inet addr:192.168.2.130 Bcast:192.168.2.255 Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 Kb) TX bytes:0 (0.0 b)
        Memory:fe5e0000-fe5e0fff

fm2-gb1 Link encap:Ethernet HWaddr 00:E0:0C:00:AA:06
        inet addr:192.168.1.131 Bcast:192.168.1.255 Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 Kb) TX bytes:0 (0.0 b)
        Memory:fe5e0000-fe5e0fff

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        UP LOOPBACK RUNNING MTU:16436 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

[root@p4080 root]#
[root@p4080 root]# arp -s 192.168.2.131 00:00:05:00:00:01
[root@p4080 root]# arp -s 192.168.1.130 00:00:06:00:00:01
[root@p4080 root]# echo 1 > /proc/sys/net/ipv4/ip_forward
[root@p4080 root]# iptables -F INPUT
[root@p4080 root]# iptables -F OUTPUT
[root@p4080 root]# iptables -F FORWARD
[root@p4080 root]# iptables -t nat -F PREROUTING
[root@p4080 root]# iptables -t nat -F POSTROUTING
[root@p4080 root]# iptables -P INPUT ACCEPT
```

```
[root@p4080 root]# iptables -P OUTPUT ACCEPT
[root@p4080 root]# iptables -P FORWARD ACCEPT
[root@p4080 root]# iptables -t nat -A POSTROUTING -p tcp -o fm2-gb1 -m iprange --src-range \
192.168.2.1-192.168.2.255 -j SNAT --to-source 9.5.2.1-9.5.2.255:1724-1755
[root@p4080 root]# iptables -L
Chain INPUT (policy ACCEPT)
target prot opt source destination

Chain FORWARD (policy ACCEPT)
target prot opt source destination

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
[root@p4080 root]# iptables -L -t nat

Chain PREROUTING (policy ACCEPT)
target prot opt source destination

Chain POSTROUTING (policy ACCEPT)
target prot opt source   destination
SNAT   tcp  --  anywhere anywhere            source IP range 192.168.2.1-192.168.2.255 to:
9.5.2.1-9.5.2.255:1724-1755
Chain OUTPUT (policy ACCEPT)
target prot opt source destination
[root@p4080 root]#
```

IPV4 Forwarding +TCP NAT+Firewall (1 rule):

```
p4080 login: root
Password:
[root@p4080 root]#
[root@p4080 root]# ifconfig fm2-gb0 192.168.2.130 netmask 255.255.255.0 up
[root@p4080 root]# ifconfig fm2-gb1 192.168.1.131 netmask 255.255.255.0 up
[root@p4080 root]# ifconfig
fm2-gb0 Link encap:Ethernet HWaddr 00:E0:0C:00:AA:05
        inet addr:192.168.2.130 Bcast:192.168.2.255 Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 Kb) TX bytes:0 (0.0 b)
        Memory:fe5e0000-fe5e0fff

fm2-gb1 Link encap:Ethernet HWaddr 00:E0:0C:00:AA:06
        inet addr:192.168.1.131 Bcast:192.168.1.255 Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 Kb) TX bytes:0 (0.0 b)
        Memory:fe5e0000-fe5e0fff

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        UP LOOPBACK RUNNING MTU:16436 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
```

```
lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        UP LOOPBACK RUNNING MTU:16436 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

[root@p4080 root]#
[root@p4080 root]# arp -s 192.168.2.131 00:00:05:00:00:01
[root@p4080 root]# arp -s 192.168.1.130 00:00:06:00:00:01
[root@p4080 root]# echo 1 > /proc/sys/net/ipv4/ip_forward
[root@p4080 root]# iptables -F INPUT
[root@p4080 root]# iptables -F OUTPUT
[root@p4080 root]# iptables -F FORWARD
[root@p4080 root]# iptables -t nat -F PREROUTING
[root@p4080 root]# iptables -t nat -F POSTROUTING
[root@p4080 root]# iptables -P INPUT ACCEPT
[root@p4080 root]# iptables -P OUTPUT ACCEPT
[root@p4080 root]# iptables -P FORWARD ACCEPT
[root@p4080 root]# iptables -t nat -A POSTROUTING -p tcp -o fm2-gb1 -m iprange --src-range \
192.168.2.1-192.168.2.255 -j SNAT --to-source 9.5.2.1-9.5.2.255:1724-1755
[root@p4080 root]# iptables -L
Chain INPUT (policy ACCEPT)
target prot opt source destination

Chain FORWARD (policy ACCEPT)
target prot opt source destination

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
[root@p4080 root]# iptables -L -t nat
Chain PREROUTING (policy ACCEPT)
target prot opt source destination
Chain POSTROUTING (policy ACCEPT)
target prot opt source   destination
SNAT   tcp  --  anywhere anywhere            source IP range 192.168.2.1-192.168.2.255 to:
9.5.2.1-9.5.2.255:1724-1755

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
[root@p4080 root]#
[root@p4080 root]# iptables -t filter -A FORWARD -i fm2-gb0 \
-d 255.128.0.0/9 -j REJECT
[root@p4080 root]# iptables -L
Chain INPUT (policy ACCEPT)
target prot opt source destination

Chain FORWARD (policy ACCEPT)
target prot opt source   destination
REJECT all  --  anywhere 255.128.0.0/9 reject-with icmp-port-unreachable

Chain OUTPUT (policy ACCEPT)
target prot opt source destination

[root@p4080 root]# iptables -L -t nat
Chain PREROUTING (policy ACCEPT)
target prot opt source destination

Chain POSTROUTING (policy ACCEPT)
```

```
target prot opt source   destination
SNAT   tcp  --  anywhere anywhere            source IP range 192.168.2.1-192.168.2.255 to:
9.5.2.1-9.5.2.255:1724-1755

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
[root@p4080 root]#
```

Test Scripts

Reset/sh

```
EXT_IFACE="fm2-gb1"
INT_IFACE="fm2-gb0"
EXT_IFACE_2="fm2-gb0"
INT_IFACE_2="fm2-gb1"
PORT_RANGE="1724-1755"
INT_IP="192.168.2.130"
EXT_IP="192.168.1.131"
INT_IP_2="192.168.3.132"
EXT_IP_2="192.168.4.133"
SRC_IP_RANGE_1="192.168.1.1-192.168.1.255"
SRC_IP_SUBNET_1="192.168.1.0/24"
FAKE_SRC_IP_RANGE_1=9.5.1.1-9.5.1.255
SRC_IP_RANGE_2=192.168.2.1-192.168.2.255
SRC_IP_SUBNET_2=192.168.4.0/24
FAKE_SRC_IP_RANGE_2=9.5.2.1-9.5.2.255
SRC_IP_RANGE_3=192.168.3.1-192.168.3.255
SRC_IP_SUBNET_3=192.168.3.0/24
FAKE_SRC_IP_RANGE_3=9.5.3.1-9.5.3.255
SRC_IP_RANGE_4=192.168.6.1-192.168.6.255
SRC_IP_SUBNET_4=192.168.6.0/24
FAKE_SRC_IP_RANGE_4=9.5.4.1-9.5.4.255
SRC_IP_RANGE_5="192.168.2.1-192.168.2.255"
SRC_IP_SUBNET_5="192.168.2.0/24"
FAKE_SRC_IP_RANGE_5=9.5.5.1-9.5.5.255
IPT="/usr/sbin/iptables"
if [ ! -e /usr/local/lib ]; then
mkdir /usr/local/lib
ln -s /lib/iptables /usr/local/lib/iptables
fi
$IPT -F INPUT
$IPT -F OUTPUT
$IPT -F FORWARD
$IPT -t nat -F PREROUTING
$IPT -t nat -F POSTROUTING
# policy: conservative setting drop everything unless allowed
# here we have a BAD policy set-up
# we do so only for the sake of catching all packets.
$IPT -P INPUT ACCEPT
$IPT -P OUTPUT ACCEPT
$IPT -P FORWARD ACCEPT
```

Test_ipv4.sh

```
ifconfig fm2-gb0 192.168.2.130 netmask 255.255.255.0 up
ifconfig fm2-gb1 192.168.1.131 netmask 255.255.255.0 up
ifconfig fm2-gb2 down
ifconfig fm2-gb3 down
```

```
ifconfig
arp -s 192.168.2.131 00:00:05:00:00:01
arp -s 192.168.1.130 00:00:06:00:00:01
echo 1 > /proc/sys/net/ipv4/ip_forward
cat /proc/sys/net/ipv4/ip_forward
```

Test_ip_nat.sh

```
#
# NAT (inbound)
#
# It seems we have problem to set this one up because NF conntrack is stateful
#
# NAT (outbound) using a range of IP and no ports
#
$IPT -t nat -A POSTROUTING -p ip -o $EXT_IFACE -m iprange --src-range \
$SRC_IP_RANGE_2 -j SNAT --to-source $FAKE_SRC_IP_RANGE_2
#$IPT -t nat -A POSTROUTING -p ip -o $INT_IFACE -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p ip -o $EXT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p ip -o $INT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
# DO NOT edit beyond this line except for firewall operation
#. new32.sh
```

Test_UDP_NAT.sh

```
#
# NAT (inbound)
#
# It seems we have problem to set this one up because NF conntrack is stateful
#
# NAT (outbound) using a range of IP and no ports
#
$IPT -t nat -A POSTROUTING -p udp -o $EXT_IFACE -m iprange --src-range \
$SRC_IP_RANGE_2 -j SNAT --to-source $FAKE_SRC_IP_RANGE_2:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p udp -o $INT_IFACE -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p udp -o $EXT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p udp -o $INT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
# DO NOT edit beyond this line except for firewall operation
#. new32.sh
```

Test_UDP_NAT_2flow.sh

```
#
# NAT (inbound)
#
# It seems we have problem to set this one up because NF conntrack is stateful
#
# NAT (outbound) using a range of IP and no ports
#
$IPT -t nat -A POSTROUTING -p udp -o $EXT_IFACE -m iprange --src-range \
$SRC_IP_RANGE_2 -j SNAT --to-source $FAKE_SRC_IP_RANGE_2:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p udp -o $INT_IFACE -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
```

```
$IPT -t nat -A POSTROUTING -p udp -o $EXT_IFACE_2 -m iprange --src-range \
$SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p udp -o $INT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
# DO NOT edit beyond this line except for firewall operation
#. new32.sh
```

Test_TCP_NAT.sh

```
#
# NAT (inbound)
#
# It seems we have problem to set this one up because NF conntrack is stateful
#
# NAT (outbound) using a range of IP and no ports
#
$IPT -t nat -A POSTROUTING -p tcp -o $EXT_IFACE -m iprange --src-range \
$SRC_IP_RANGE_2 -j SNAT --to-source $FAKE_SRC_IP_RANGE_2:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p tcp -o $INT_IFACE -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p tcp -o $EXT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p tcp -o $INT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
# DO NOT edit beyond this line except for firewall operation
#. new32.sh
```

Test_TCP_NAT_2flow.sh

```
#
# NAT (inbound)
#
# It seems we have problem to set this one up because NF conntrack is stateful
#
# NAT (outbound) using a range of IP and no ports
#
$IPT -t nat -A POSTROUTING -p tcp -o $EXT_IFACE -m iprange --src-range \
$SRC_IP_RANGE_2 -j SNAT --to-source $FAKE_SRC_IP_RANGE_2:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p tcp -o $INT_IFACE -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
$IPT -t nat -A POSTROUTING -p tcp -o $EXT_IFACE_2 -m iprange --src-range \
$SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p tcp -o $INT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
# DO NOT edit beyond this line except for firewall operation
#. new32.sh
```

New1.sh

```
$IPT -t filter -A FORWARD -i $INT_IFACE -d 255.128.0.0/9 -j REJECT
```

12.2.7 eTSEC NAT and Firewall

How to setup the board to reproduce NAT and Firewall benchmark results for the QorIQ USDK boards.

12.2.7.1 Hardware

Benchmarking Objectives

The purpose of this guide is to show how to setup NAT and Firewall on a NXP QorIQ processor based system and how to measure its performance.

NOTE

LS1021ATWR was used for the demo. Please refer to the board's manual for other boards.

The relevant QorIQ development systems covered

- LS1021ATWR

Hardware Platform Identification

Table 608. LS1021ATWR SoC Clock Settings

Core/CPU (MHz)	CCB (MHz)	DDR (MHz)
1000	300	800

Flashing U-boot on LS1021ATWR

- `tftpboot 0x81000000 u-boot.bin`
- `protect off 0x64100000 +$filesize`
- `erase 0x64100000 +$filesize;`
- `cp.b 1000000  0x64100000 $filesize;`
- `protect on 0x64100000 +$filesize;`
- `cmp.b  1000000 0x64100000 $filesize`

Flashing RCW on LS1021ATWR

- `tftpboot 0x81000000 u-boot.bin`
- `protect off 0x64000000 +$filesize`
- `erase 0x64000000 +$filesize;`
- `cp.b 1000000  0x64000000 $filesize;`
- `protect on 0x64000000 +$filesize;`
- `cmp.b  1000000 0x64000000 $filesize`

Environmental Variables and boot up kernel

- ```
  setenv ipaddr 10.192.208.162
  ```
- ```
  setenv serverip 10.193.20.104
  ```
- ```
  setenv gatewayip 10.192.208.254
  ```
- ```
  setenv bootargs root=/dev/ram rw ramdisk_size=1000000 log_buf_len=128K console=ttyS0,115200
  ```
- ```
  tftp 82000000 uImage
  ```
- ```
  tftp 88000000 rootfs.ext2.gz.uboot
  ```
- ```
  tftp 8f000000 uImage.dtb
  ```
- ```
  bootm 82000000 88000000 8f000000
  ```

12.2.7.2 Kernel

The kernel is prepared and configured to run the eTSEC performance benchmark on QorIQ boards.

Preparing the Kernel on LS1021ATWR: Config Kernel

Kernel Configure Tree View Options, this is only a demo, please enable the kernel NAT/Firewall options as more as possible according to the kernel version.

```
[*] Networking support  --->
    Networking options  --->
        [*] Network packet filtering framework (Netfilter)  --->
            [ ]   Network packet filtering debugging (NEW)
            [*]   Advanced netfilter configuration (NEW)
            Core Netfilter Configuration  --->
            -*- Netfilter NFACCT over NFNETLINK interface
            -*- Netfilter NFQUEUE over NFNETLINK interface
            -*- Netfilter LOG over NFNETLINK interface
            <*> Netfilter connection tracking support
            -*-   Connection mark tracking support
                  [*]   Connection tracking zones
                  [*]   Connection tracking events
                  [*]   Connection tracking timeout
                  [*]   Connection tracking timestamping
                  <*>   DCCP protocol connection tracking support (EXPERIMENTAL)
                  <*>   SCTP protocol connection tracking support (EXPERIMENTAL)
                  <*>   UDP-Lite protocol connection tracking support
                  <*>   Amanda backup protocol support
                  <*>   FTP protocol support
                  <*>   H.323 protocol support
                  <*>   IRC protocol support
                  <*>   NetBIOS name service protocol support
                  <*>   SNMP service protocol support
                  <*>   PPtP protocol support
                  <*>   SANE protocol support (EXPERIMENTAL)
                  <*>   SIP protocol support
                  <*>   TFTP protocol support
```

```
                <*>   Connection tracking netlink interface
                <*>   Connection tracking timeout tuning via Netlink
                <*>   Connection tracking helpers in user-space via Netlink
                [*]   NFQUEUE integration with Connection Tracking
                <*> Transparent proxying support (EXPERIMENTAL)
                -*- Netfilter Xtables support (required for ip_tables)
                    *** Xtables combined modules ***
                -*-   nfmark target and match support
                -*-   ctmark target and match support
                       *** Xtables targets ***
                <*>   AUDIT target support
                <*>   CHECKSUM target support
                <*>   "CLASSIFY" target support
                <*>   "CONNMARK" target support
                -*-   "CT" target support
                <*>   "DSCP" and "TOS" target support
                -*-   "HL" hoplimit target support
                <*>   "HMARK" target support
                <*>   IDLETIMER target support
                <*>   LOG target support
                <*>   "MARK" target support
                -*-   "NETMAP" target support
                <*>   "NFLOG" target support
                <*>   "NFQUEUE" target Support
                <*>   "NOTRACK" target support
                -*-   "RATEEST" target support
                -*-   REDIRECT target support
                <*>   "TEE" - packet cloning to alternate destination
                <*>   "TPROXY" target support (EXPERIMENTAL)
                <*>   "TRACE" target support
                <*>   "TCPMSS" target support
                <*>   "TCPOPTSTRIP" target support (EXPERIMENTAL)
                       *** Xtables matches ***
                <*>   "addrtype" address type match support
                <*>   "cluster" match support
                <*>   "comment" match support
                <*>   "connbytes" per-connection counter match support
                <*>   "connlimit" match support"
                <*>   "connmark" connection mark match support
                <*>   "conntrack" connection tracking match support
                <*>   "cpu" match support
                <*>   "dccp" protocol match support
                <*>   "devgroup" match support
                <*>   "dscp" and "tos" match support
                -*-   "ecn" match support
                <*>   "esp" match support
                <*>   "hashlimit" match support
                <*>   "helper" match support
                -*-   "hl" hoplimit/TTL match support
                <*>   "iprange" address range match support
                <*>   "length" match support
                <*>   "limit" match support
                <*>   "mac" address match support
                <*>   "mark" match support
                <*>   "multiport" Multiple port match support
                <*>   "osf" Passive OS fingerprint match
                <*>   "owner" match support
                <*>   IPsec "policy" match support
                <*>   "pkttype" packet type match support
                <*>   "quota" match support
```

```
                <*>   "rateest" match support
                <*>   "realm" match support
                <*>   "recent" match support
                <*>   "sctp" protocol match support (EXPERIMENTAL)
                <*>   "socket" match support (EXPERIMENTAL)
                <*>   "state" match support
                <*>   "statistic" match support
                <*>   "string" match support
                <*>   "tcpmss" match support
                <*>   "time" match support
                <*>   "u32" match support
            < >   IP set support (NEW)  --->
            < >   IP virtual server support (NEW)  --->
          IP: Netfilter Configuration  --->
                <*> IPv4 connection tracking support (required for NAT)
                <*> IP Userspace queueing via NETLINK (OBSOLETE)
                <*> IP tables support (required for filtering/masq/NAT)
                <*>   "ah" match support
                <*>   "ecn" match support
                <*>   "rpfilter" reverse path filter match support
                <*>   "ttl" match support
                <*>   Packet filtering
                <*>     REJECT target support
                <*>   LOG target support
                <*>   ULOG target support
                <*>   IPv4 NAT
                <*>     MASQUERADE target support
                <*>     NETMAP target support
                <*>     REDIRECT target support
                <*>     Basic SNMP-ALG support (NEW)
                <*>   Packet mangling
                <*>     CLUSTERIP target support (EXPERIMENTAL)
                <*>     ECN target support
                <*>     "TTL" target support
                <*>   raw table support (required for NOTRACK/TRACE)
                <*> ARP tables support
                <*>   ARP packet filtering
                <*>   ARP payload mangling
```

```
Gianfar Ethernet (CONFIG_GIANFAR):
Location:
    -> Device Drivers
       -> Network device support (NETDEVICES [=y])
         -> Ethernet driver support (ETHERNET [=y])
           -> Freescale devices (NET_VENDOR_FREESCALE [=y])
```

Preparing the Kernel on LS1021ATWR: Kernel config Tips

Use the default kernel config then enable almost all NAT/Firewall features as the NAT/Firewall config.

Prepare ramdisk on LS1021ATWR

Select the following packages when build ramdisk:

-
```
[*] busybox(busybox.config)
```

-
```
[*] ethtool
```

- `[*] net-tools`
- `[*] iptables`

Make sure BusyBox Configuration includes following changes under Networking Utilities:

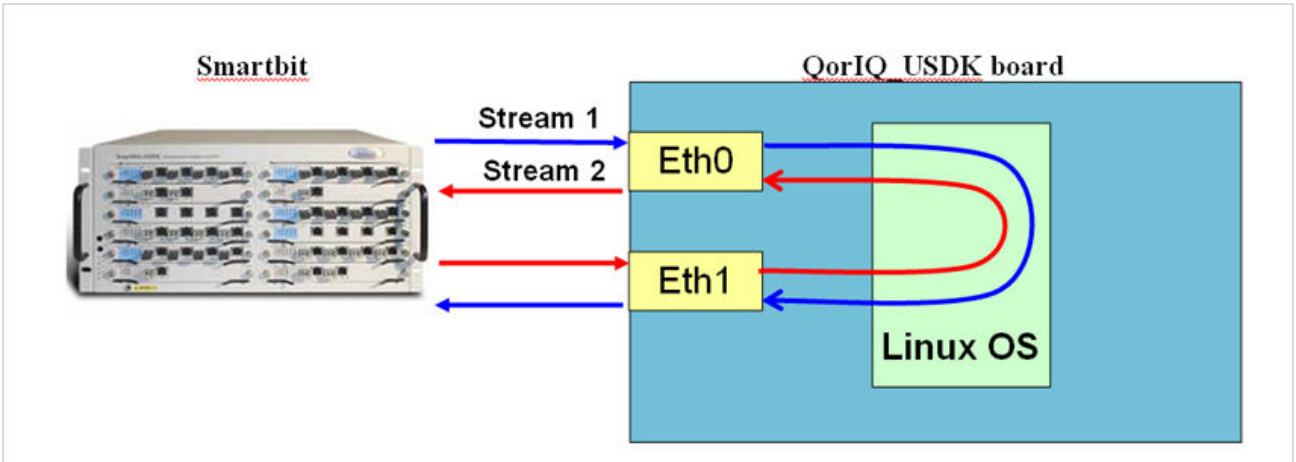
- `[*]arping`
- `[*]ftpget`
- `[*]ftpput`
- `[*]hostname`
- `[*]ifconfig`
- `[*]Enable status reporting output (+7k)`
- `[*]Enable option "hw" (ether only)`
- `[*]ifupdown`
- `[*]arping`
- `[*]Use ip applet`
- `[*]tftp`
- `[*]Enable "get" command`
- `[*]Enable "put" command`
- `[*]ifupdown`

12.2.7.3 Test Setup

Running the eTSEC performance benchmark test

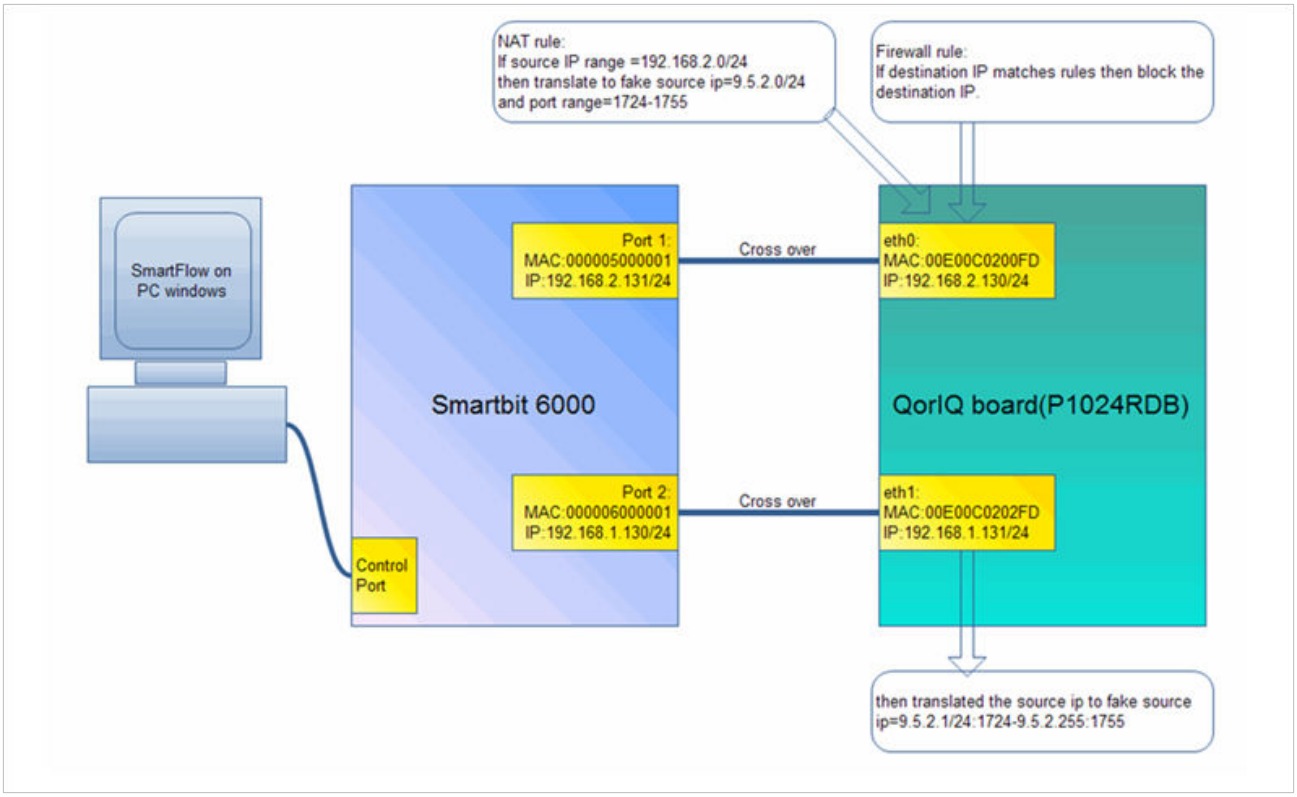
Typical Test Setup on LS1021ATWR

- Uni-directional (1 flow) = Stream 1
- Bi-directional (2 flow) = Stream 1 + Stream 2
-

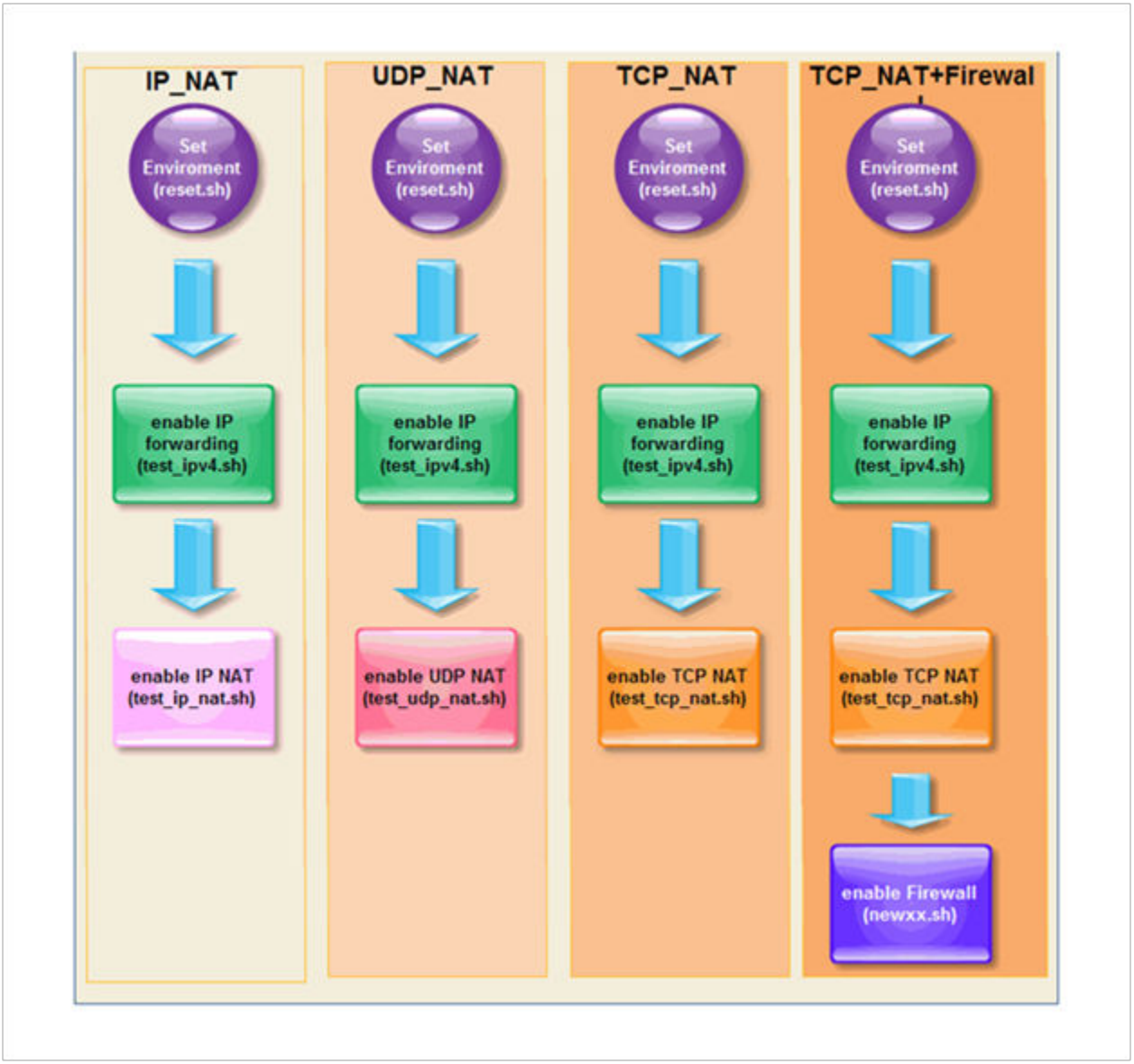


1. Connect two Ethernet cross cables between the target board and Smartbits Ethernet ports
2. Connect one Ethernet cross cable between a Windows PC and Smartbits management port
3. Run smartflow from Windows PC and connect to Smartbits, the Smartbits' IP is 10.193.20.212.
4. Set IP and MAC to adapt both Smartbits and the board.

LS1021ATWR NAT and Firewall Test Setup



LS1021ATWR Test Procedure



- Use reset.sh to set the environment which is included Source IP range, Fake IP range, income network device, output network device, then flush all rules in those tables and set the default policy for the tables.
- Use `test_ipv4.sh` to set QorIQ_SDK board IP address, do smartbit MAC and IP binding, and then enable IP Forwarding.
- Use `test_xxx_NAT.sh` to enable NAT translate.
- Use `newxxx.sh` to enable Firewall. `New1.sh` means 1 rule, new128 means 128 rules, new1024.sh means 1024 rules.

LS1021ATWR Test Tips

To improve performance, tune Rx and Tx interrupt coalescing values to reduce the interrupt rate. A more suitable (compared to the default values) set of parameters for the packet forwarding use case is:

- ```
  #ethtool -C eth0 rx-frames 32 rx-usecs 22 tx-frames 32 tx-usecs 22
  ```
- ```
  #ethtool -C eth1 rx-frames 32 rx-usecs 22 tx-frames 32 tx-usecs 22
  ```

These commands are used to tune the RX interrupt rate per RX/TX DMA channel. Also, it is recommended to set the Rx/ Tx interrupt affinities in such a way as to ensure Flow/CPU affinity, for improved performance.

12.2.7.4 Test Demos

Example Test Demo and scripts to setup the board to reproduce NAT and Firewall benchmark results.

LS1021ATWR Test Demos

Examples of test demo code are provided below.

1. IPV4 Forwarding + TCP NAT:

```
!!!!! WARNING !!!!!!!

The default password for the root account is: root
please change this password using the 'passwd' command
and then edit this message (/etc/issue) to remove this message

ls1021atwr login: root
Password:
[root@ls1021atwr root]#
[root@ls1021atwr root]# ifconfig eth0 192.168.2.130 netmask 255.255.255.0 up
[root@ls1021atwr root]# ifconfig eth1 192.168.1.131 netmask 255.255.255.0 up
txbd[0]: addr, vaddr = 0x2ab08000,0xeab08000
txbd[1]: addr, vaddr = 0x2ab08400,0xeab08400
txbd[2]: addr, vaddr = 0x2ab08800,0xeab08800
txbd[3]: addr, vaddr = 0x2ab08c00,0xeab08c00
txbd[4]: addr, vaddr = 0x2ab09000,0xeab09000
txbd[5]: addr, vaddr = 0x2ab09400,0xeab09400
txbd[6]: addr, vaddr = 0x2ab09800,0xeab09800
txbd[7]: addr, vaddr = 0x2ab09c00,0xeab09c00
rxbd[0]: addr,vaddr=0x2ab0a000,0xeab0a000
rxbd[1]: addr,vaddr=0x2ab0a400,0xeab0a400
rxbd[2]: addr,vaddr=0x2ab0a800,0xeab0a800
rxbd[3]: addr,vaddr=0x2ab0ac00,0xeab0ac00
rxbd[4]: addr,vaddr=0x2ab0b000,0xeab0b000
rxbd[5]: addr,vaddr=0x2ab0b400,0xeab0b400
rxbd[6]: addr,vaddr=0x2ab0b800,0xeab0b800
rxbd[7]: addr,vaddr=0x2ab0bc00,0xeab0bc00
[root@ls1021atwr root]# PHY: mdio@ffe24000:02 - Link is Up - 1000/Full
[root@ls1021atwr root]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:E0:0E:02:00:FD
          inet addr:192.168.2.130  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3204 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:344509 (336.4 Kb)  TX bytes:0 (0.0 b)
          Base address:0xe000

eth1      Link encap:Ethernet  HWaddr 00:04:9F:01:21:C3
          inet addr:192.168.1.131  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:897 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:94254 (92.0 Kb)  TX bytes:0 (0.0 b)
          Base address:0x6000
```

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

[root@ls1021atwr root]#
[root@ls1021atwr root]# arp -s 192.168.2.131 00:00:05:00:00:01
[root@ls1021atwr root]# arp -s 192.168.1.130 00:00:06:00:00:01
[root@ls1021atwr root]# echo 1 > /proc/sys/net/ipv4/ip_forward
[root@ls1021atwr root]# iptables -F INPUT
[root@ls1021atwr root]# iptables -F OUTPUT
[root@ls1021atwr root]# iptables -F FORWARD
[root@ls1021atwr root]# iptables -t nat -F PREROUTING
[root@ls1021atwr root]# iptables -t nat -F POSTROUTING
[root@ls1021atwr root]# iptables -P INPUT ACCEPT
[root@ls1021atwr root]# iptables -P OUTPUT ACCEPT
[root@ls1021atwr root]# iptables -P FORWARD ACCEPT
[root@ls1021atwr root]# iptables -t nat -A POSTROUTING -p tcp -o eth1 -m iprange --src-range \
 192.168.2.1-192.168.2.255 -j SNAT --to-source 9.5.2.1-9.5.2.255:1724-1755
[root@ls1021atwr root]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
[root@ls1021atwr root]# iptables -L -t nat
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT       tcp  --  anywhere            anywhere           source IP range 192.168.2.1-192.168.2.255
to:9.5.2.1-9.5.2.255:1724-1755

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
[root@ls1021atwr root]# uname -a
Linux ls1021atwr 2.6.32.13-01402-g9e1d6fe-dirty #6 SMP Thu Jul 29 16:33:36 CST 2010 ppc GNU/Linux
[root@ls1021atwr root]# ethtool  -C eth0 rx-frames 32 rx-usecs 22
[root@ls1021atwr root]# ethtool  -C eth0 tx-frames 32 tx-usecs 22
[root@ls1021atwr root]# ethtool  -C eth1 rx-frames 32 rx-usecs 22
[root@ls1021atwr root]# ethtool  -C eth1 tx-frames 32 tx-usecs 22
[root@ls1021atwr root]#
```

2. IPV4 Forwarding +TCP NAT+Firewall (1 rule):

```
ls1021atwr login: root
Password:
[root@ls1021atwr root]# ifconfig eth0 192.168.2.130 netmask 255.255.255.0 up
[root@ls1021atwr root]# ifconfig eth1 192.168.1.131 netmask 255.255.255.0 up
txbd[0]: addr, vaddr = 0x2ab08000,0xeab08000
txbd[1]: addr, vaddr = 0x2ab08400,0xeab08400
txbd[2]: addr, vaddr = 0x2ab08800,0xeab08800
```

```
txbd[3]: addr, vaddr = 0x2ab08c00,0xeab08c00
txbd[4]: addr, vaddr = 0x2ab09000,0xeab09000
txbd[5]: addr, vaddr = 0x2ab09400,0xeab09400
txbd[6]: addr, vaddr = 0x2ab09800,0xeab09800
txbd[7]: addr, vaddr = 0x2ab09c00,0xeab09c00
rxbd[0]: addr,vaddr=0x2ab0a000,0xeab0a000
rxbd[1]: addr,vaddr=0x2ab0a400,0xeab0a400
rxbd[2]: addr,vaddr=0x2ab0a800,0xeab0a800
rxbd[3]: addr,vaddr=0x2ab0ac00,0xeab0ac00
rxbd[4]: addr,vaddr=0x2ab0b000,0xeab0b000
rxbd[5]: addr,vaddr=0x2ab0b400,0xeab0b400
rxbd[6]: addr,vaddr=0x2ab0b800,0xeab0b800
rxbd[7]: addr,vaddr=0x2ab0bc00,0xeab0bc00
[root@ls1021atwr root]# PHY: mdio@ffe24000:02 - Link is Up - 1000/Full
[root@ls1021atwr root]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:E0:0E:02:00:FD
          inet addr:192.168.2.130  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3204 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:344509 (336.4 Kb)  TX bytes:0 (0.0 b)
          Base address:0xe000

eth1      Link encap:Ethernet  HWaddr 00:04:9F:01:21:C3
          inet addr:192.168.1.131  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:897 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:94254 (92.0 Kb)  TX bytes:0 (0.0 b)
          Base address:0x6000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

[root@ls1021atwr root]#
[root@ls1021atwr root]# arp -s 192.168.2.131 00:00:05:00:00:01
[root@ls1021atwr root]# arp -s 192.168.1.130 00:00:06:00:00:01
[root@ls1021atwr root]# echo 1 > /proc/sys/net/ipv4/ip_forward
[root@ls1021atwr root]# iptables -F INPUT
[root@ls1021atwr root]# iptables -F OUTPUT
[root@ls1021atwr root]# iptables -F FORWARD
[root@ls1021atwr root]# iptables -t nat -F PREROUTING
[root@ls1021atwr root]# iptables -t nat -F POSTROUTING
[root@ls1021atwr root]# iptables -P INPUT ACCEPT
[root@ls1021atwr root]# iptables -P OUTPUT ACCEPT
[root@ls1021atwr root]# iptables -P FORWARD ACCEPT
[root@ls1021atwr root]# iptables -t nat -A POSTROUTING -p tcp -o eth1 -m iprange --src-range \
 192.168.2.1-192.168.2.255 -j SNAT --to-source 9.5.2.1-9.5.2.255:1724-1755
[root@ls1021atwr root]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
```

```
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
[root@ls1021atwr root]# iptables -L -t nat
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT      tcp  --  anywhere            anywhere           source IP range 192.168.2.1-192.168.2.255
to:9.5.2.1-9.5.2.255:1724-1755

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
[root@ls1021atwr root]# uname -a
Linux ls1021atwr 2.6.32.13-01402-g9e1d6fe-dirty #6 SMP Thu Jul 29 16:33:36 CST 2010 ppc GNU/Linux
[root@ls1021atwr root]# ethtool  -C eth0 rx-frames 32 rx-usecs 22
[root@ls1021atwr root]# ethtool  -C eth0 tx-frames 32 tx-usecs 22
[root@ls1021atwr root]# ethtool  -C eth1 rx-frames 32 rx-usecs 22
[root@ls1021atwr root]# ethtool  -C eth1 tx-frames 32 tx-usecs 22
[root@ls1021atwr root]#

[root@ls1021atwr root]#
[root@ls1021atwr root]# iptables -t filter -A FORWARD -i eth0 \
 -d 255.128.0.0/9 -j REJECT
[root@ls1021atwr root]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
REJECT     all  --  anywhere             255.128.0.0/9       reject-with icmp-port-unreachable

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
[root@ls1021atwr root]# iptables -L -t nat
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT      tcp  --  anywhere            anywhere           source IP range 192.168.2.1-192.168.2.255
to:9.5.2.1-9.5.2.255:1724-1755

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

[root@ls1021atwr root]#
```

LS1021AWTR Test Scripts

Six examples of test scripts are shown below:

1. Reset/sh

```
EXT_IFACE="eth1"
INT_IFACE="eth0"
EXT_IFACE_2="eth0"
INT_IFACE_2="eth1"
```

```
PORT_RANGE="1724-1755"
INT_IP="192.168.2.130"
EXT_IP="192.168.1.131"
INT_IP_2="192.168.3.132"
EXT_IP_2="192.168.4.133"
SRC_IP_RANGE_1="192.168.1.1-192.168.1.255"
SRC_IP_SUBNET_1="192.168.1.0/24"
FAKE_SRC_IP_RANGE_1=9.5.1.1-9.5.1.255
SRC_IP_RANGE_2=192.168.2.1-192.168.2.255
SRC_IP_SUBNET_2=192.168.4.0/24
FAKE_SRC_IP_RANGE_2=9.5.2.1-9.5.2.255
SRC_IP_RANGE_3=192.168.3.1-192.168.3.255
SRC_IP_SUBNET_3=192.168.3.0/24
FAKE_SRC_IP_RANGE_3=9.5.3.1-9.5.3.255
SRC_IP_RANGE_4=192.168.6.1-192.168.6.255
SRC_IP_SUBNET_4=192.168.6.0/24
FAKE_SRC_IP_RANGE_4=9.5.4.1-9.5.4.255
SRC_IP_RANGE_5="192.168.2.1-192.168.2.255"
SRC_IP_SUBNET_5="192.168.2.0/24"
FAKE_SRC_IP_RANGE_5=9.5.5.1-9.5.5.255
IPT="/usr/sbin/iptables"
if [ ! -e /usr/local/lib ]; then
mkdir /usr/local/lib
ln -s /lib/iptables /usr/local/lib/iptables
fi
$IPT -F INPUT
$IPT -F OUTPUT
$IPT -F FORWARD
$IPT -t nat -F PREROUTING
$IPT -t nat -F POSTROUTING
# policy: conservative setting drop everything unless allowed
# here we have a BAD policy set-up
# we do so only for the sake of catching all packets.
$IPT -P INPUT ACCEPT
$IPT -P OUTPUT ACCEPT
$IPT -P FORWARD ACCEPT
```

2. Test_ipv4.sh

```
ifconfig eth0 192.168.2.130 netmask 255.255.255.0 up
ifconfig eth1 192.168.1.131 netmask 255.255.255.0 up
ifconfig eth2 down
ifconfig eth3 down
ifconfig
arp -s 192.168.2.131 00:00:05:00:00:01
arp -s 192.168.1.130 00:00:06:00:00:01
echo 1 > /proc/sys/net/ipv4/ip_forward
cat /proc/sys/net/ipv4/ip_forward
```

3. Test_ip_nat.sh

```
#
# NAT (inbound)
#
# It seems we have problem to set this one up because NF conntrack is stateful
#
# NAT (outbound) using a range of IP and no ports
#
$IPT -t nat -A POSTROUTING -p ip -o $EXT_IFACE -m iprange --src-range \
```

```
 $SRC_IP_RANGE_2 -j SNAT --to-source $FAKE_SRC_IP_RANGE_2
#$IPT -t nat -A POSTROUTING -p ip -o $INT_IFACE -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p ip -o $EXT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p ip -o $INT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
# DO NOT edit beyond this line except for firewall operation
#. new32.sh
```

4. Test_UDP_NAT.sh

```
#
# NAT (inbound)
#
# It seems we have problem to set this one up because NF conntrack is stateful
#
# NAT (outbound) using a range of IP and no ports
#
$IPT -t nat -A POSTROUTING -p udp -o $EXT_IFACE -m iprange --src-range \
 $SRC_IP_RANGE_2 -j SNAT --to-source $FAKE_SRC_IP_RANGE_2:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p udp -o $INT_IFACE -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p udp -o $EXT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p udp -o $INT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
# DO NOT edit beyond this line except for firewall operation
#. new32.sh
```

5. Test_TCP_NAT.sh

```
#
# NAT (inbound)
#
# It seems we have problem to set this one up because NF conntrack is stateful
#
# NAT (outbound) using a range of IP and no ports
#
$IPT -t nat -A POSTROUTING -p tcp -o $EXT_IFACE -m iprange --src-range \
 $SRC_IP_RANGE_2 -j SNAT --to-source $FAKE_SRC_IP_RANGE_2:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p tcp -o $INT_IFACE -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p tcp -o $EXT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
#$IPT -t nat -A POSTROUTING -p tcp -o $INT_IFACE_2 -m iprange --src-range \
# $SRC_IP_RANGE_1 -j SNAT --to-source $FAKE_SRC_IP_RANGE_1:$PORT_RANGE
# DO NOT edit beyond this line except for firewall operation
#. new32.sh
```

6. New1.sh

```
$IPT -t filter -A FORWARD -i $INT_IFACE -d 255.128.0.0/9 -j REJECT
```

12.2.8 NAS - DPAA

How to setup a NAS - DPAA system on NXP QorIQ system, and how to measure its performance.

12.2.8.1 Benchmarking Objectives

The purpose of this guide is to show how to setup a NAS system on a NXP QorIQ DPAA system and how to measure its performance.

NOTE

P2040RDB /P2041RDB was used for the demo. Please refer to the board's manual for other DPAA boards.

12.2.8.2 Test Environment

NAS Client

The suggested NAS client is described in table 1. The client has high throughput performance to make sure the testbed would not limit the NAS performance. Any degradation to the suggested NAS client may lead to degradation of overall NAS performance.

Table 609. NAS client configuration

Item	Description
CPU	Minimum requires Intel® i5 CPU 750@2.67GHz
RAM	4G DDR3 RAM
Hard Drives	Western Digital® WD3000HLFS 10000RPM,Two drives configured in RAID 0 array, higher performance SSD a plus
Ethernet	Standalone Intel PRO/1000 PT Desktop Adapter (PCIE) for 1 Gbps bandwidth, or Intel PRO/1000 PT Quad Port Server Adapter for 4 Gbps bandwidth, 10 Gbps NICs are required for more than 8 HDD raid benchmarking
OS	Window® 7 (32bit) or Windows Vista®
Anti-virus software	off
Firewall software	off
Connection to NAS server	Direct connection through Ethernet cable without switches

NAS server

In this document, the NAS benchmark will be done on the P2041RDB and P2040RDB. A P2041RDB board can be configured to a P2040 board. For details, please refer to user manual of the board.

The SATA disk being used is Western Digital WD3000HLFS, 10000RPM. If lower speed SATA is used, it will impact the overall NAS performance.

P2041/P2040 has only 2 onboard SATA connector, which is not enough to compose RAID5 array. Therefore, 2 ports PCIe to SATA adaptor will be used for RAID5 tests.

See Table 2 for details.

Table 610. NAS server configuration

Item	Description
Silicon and board	P2041RDB P2040RDB

Table continues on the next page...

Table 610. NAS server configuration (continued)

Item	Description
RAM	4G DDR3 RAM
Hard Drives	Western Digital WD3000HLFS 10000 RPM, higher performance SSD a plus Single drive or four drives configured in RAID 5 array, for others RAID6/0/10 NAS, please refer to NXP RAID benchmark guide
Ethernet	Onboard dTSECs or TGECs
PCIe-SATA adaptor	At least 2 x Silicon Image 3132, 2 x LSI-9221 cards, a plus
MTU setting	MTU=1500, 9014
OS	Linux BSP, SDK 2.0 kernel version 4.1.8 or later

Measurement

- File copy under Windows7
- The tests are done using the robocopy command built into Windows 7 (and available as a download as part of the free Windows Server® 2003 Resource Kit Tools).
- The benchmarks consist of Write and Read transfers of a single file which size is 8 GB (double size of the NAS server memory).
- A result of (XXX) MB/s means (YYYYYYYYY) Bytes/1024/1024 instead of (YYYYYYYYY) Bytes/1000/1000.
- The CPU utilization should be recorded when the throughput is being recorded.

Important tips for the best performance

U-boot part

- Make sure 2 PCIe slots enable and configured as PCIe x 4 to boost RAID5 and SATA performance

Linux kernel part

1. Use the corenet32_smp_defconfig to configure your kernel and make sure to enable:
   - FSL_DPAA_ETH_OPTIMIZE_FOR_TERM
   - FSL_DPAA_ETH_SG_SUPPORT
   - FSL_FM_RX_EXTRA_HEADROOM=0
   - FMAN_P3040_P4080_P5020=[y]
   - EXT4_FS
2. Make sure to disable:
   - FSL_DPAA_OFFLINE_PORTS
   - FSL_DPAA_1588
   - NETFILTER

Others

1. Enable NIC IPv4 checksum offload in both NAS server and NAS client
2. Enable NIC TCP RX/TX checksum offload in both NAS server and NAS client
3. Ensure PCD effective in NAS server, and disable unused network interface in the PCD config file
4. Ensure every NIC works at proper 1 Gbps or 10 Gbps Full Duplex mode as expected

5. While benchmarking performance in multi-client mode, ensure every file which will be copied will deploy to different NAS client disks.
6. Use SSD to instead of normal machine disk if possible
7. The performance of local SATA/RAID5 should be guaranteed. To enable the SEC to offload the XOR for RAID5, refer to the local SATA/RAID5 benchmark guide.
8. Use multi-client mode to avoid a network interface bottle-neck. To calculate of number of networking interfaces, use the following formula:

   (Maximum throughput of local SATA/RAID5/RAID6 reading) / (Line speed of every networking interface) = Number of networking interfaces

   For example:

   (750 MB/s) / (110 MB/s) = 6.8 means that 7 network interfaces of 1 Gbps and 7 clients are needed or one 10 Gbps networking interface is needed.

12.2.8.3 Test Procedure

This section describes the test procedure for various configurations of a DPAA NAS

12.2.8.3.1 Single-core single-disk single-client

Hardware setup

Connect a single disk to a Silicon Image PCIe-SATA adaptor (Figure 3.1.1) on the P2041RDB board then benchmark the NAS performance of single-disk/single-client.

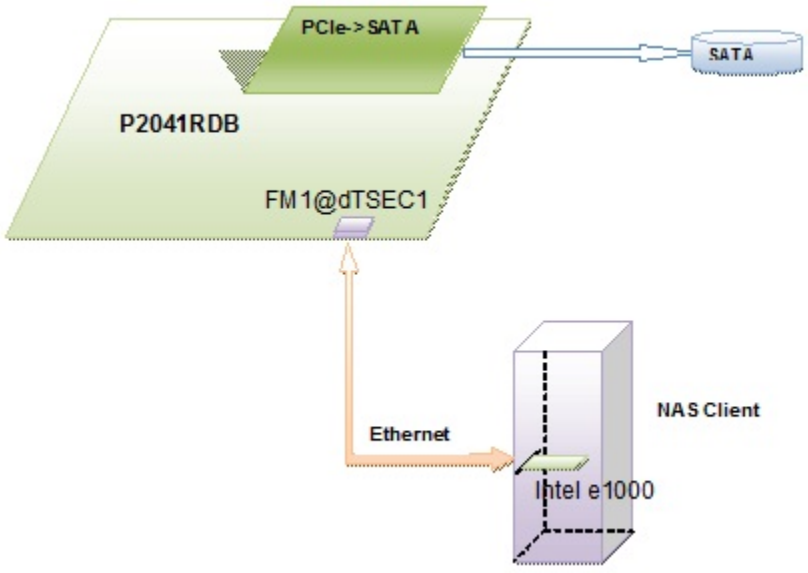


Figure 382. Figure 3.1.1

Test procedures

Please follow the below steps to have single-disk/single-client NAS performance test:

1. Power on the P2041 board, in u-boot, use the following command to disable other 3 cores and only enable 1 core:
   - >cpu disable 3
   - >cpu disable 2

- >cpu disable 1

2. Continue to boot up the P2041 board into Linux. Type “root” to enter the Linux command line.
3. Setup shared folder on P2041 board with below commands. Suppose the shared folder is /smbshare:
   - a)mkfs.ext2 /dev/sda1
   - b)mount /dev/sda1 /smbshare
   - c)echo 512 > /sys/block/sda/bdi/read_ahead_kb
   - d)ifconfig fm1-gb0 192.168.1.100
   - e)fmc –c NAS_cfg_P4080_P5020_P5040_P3041_P2041_P2040.xml –p NAS_policy_P4080_P5020_P5040_P3041_P2041_P2040.xml -a
   - f)ethtool –K fm1-gb0 gro on gso on sg on

   If you want to use EXT4 file system, change above command a) b) to:

   - a’) mkfs.ext4 /dev/sda1
   - b’) mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/sda1 /smbshare
4. Startup Samba service on board side with command:
   - smbd –s /etc/samba/smb.conf
5. On NAS client side, map the shared folder /smbshare as a windows drive, e.g, drive z. Please refer to the below steps for details, then you will get a new drive as figure 3.1.2.
   - a)Connect the NAS client and the board directly by a crossover network cable.
   - b)Set up the NAS client IP to 192.168.1.xxx
   - c)Open a command prompt window on NAS client and input the following command to map the shared folder as a windows drive:
     - Net use Z: \\192.168.1.100\public

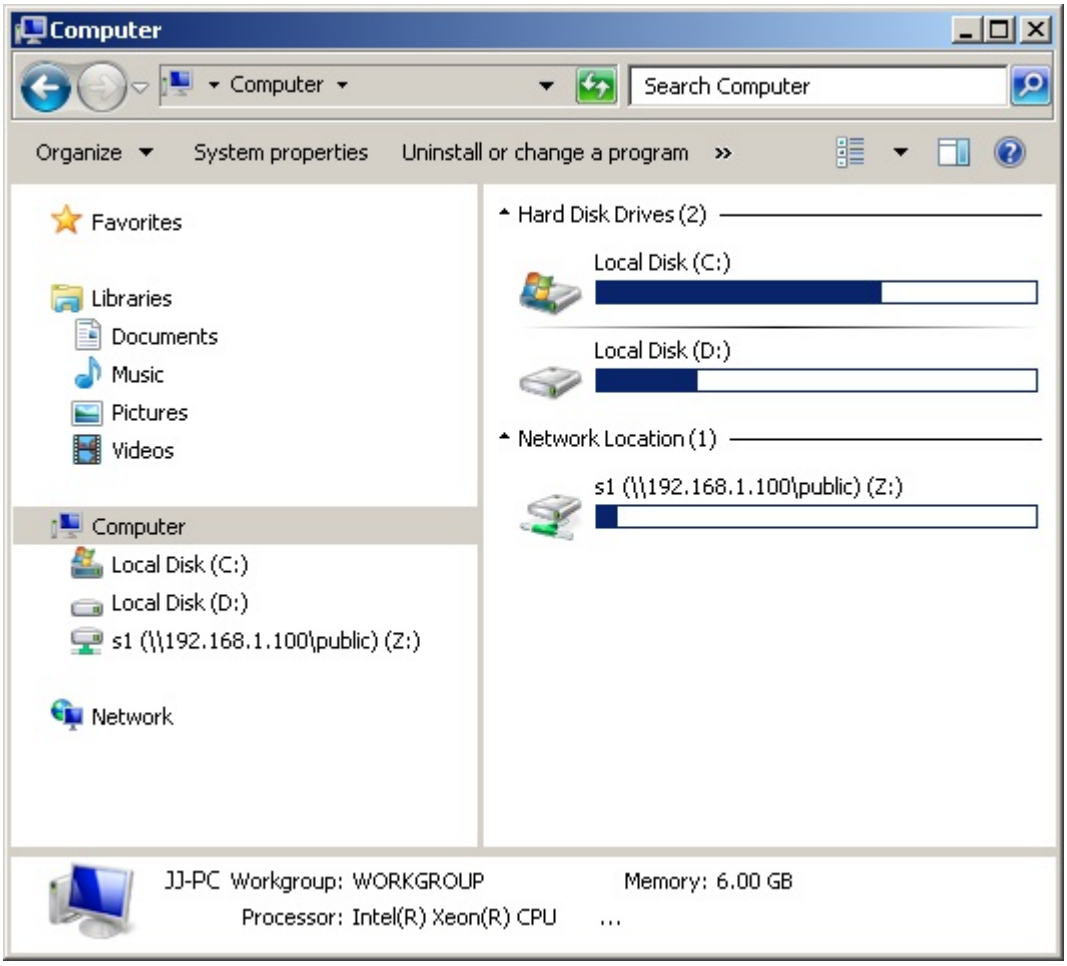


Figure 383. Figure 3.1.2

6. Setup up jumbo frame on both client and server side. With jumbo frame, better NAS performance can be achieved. Please skip this step if you want to keep the default 1500 Bytes MTU.
   - a)At the NAS client (windows 7) side, open network and sharing center by selecting Start-> Settings-> Control Panel-> System and Security-> System-> Device Manager-> Intel(R) PRO/1000 PT Desktop Adapter and configure Jumbo Packets as 9014 bytes. Please refer to Figure 3.1.3.
   - b)At the NAS server (Linux) side, config the ethernet MTU to 9014 with command:
     - # ifconfig fm1-gb0 mtu 9014

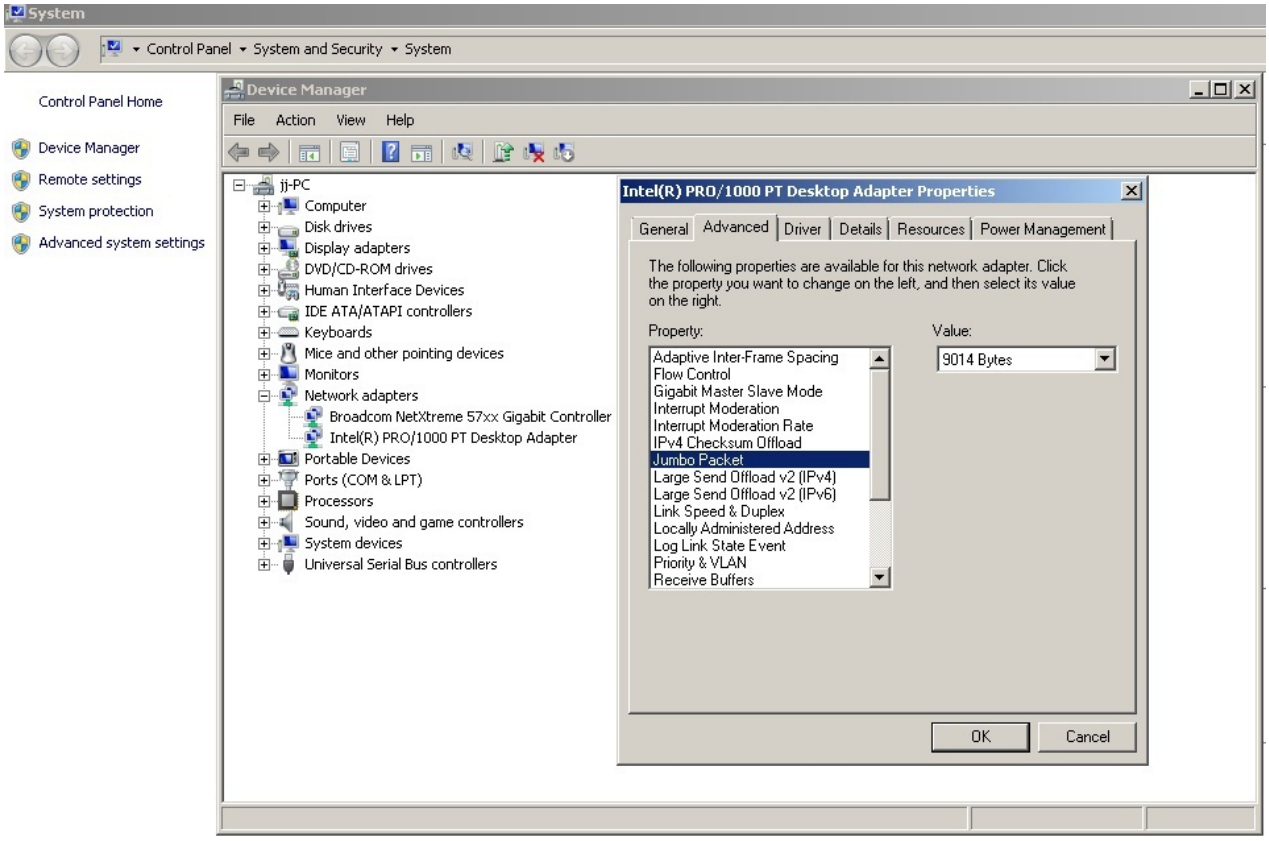


Figure 384. Figure 3.1.3

7. Generate a dummy file with 2GB with the following command at NAS client side:
   - prompt> fsutil file createnew 2G.001 2147483648
8. Begin the test with below commands at NAS client side:
   - To test writing performance: Prompt> robocopy . z:\ 2G.001 /NP /NS /COPY:D
   - To test reading performance: Prompt> robocopy z:\ . 2G.001 /NP /NS /COPY:D
9. Recording the performance number:
   - a)To warm up the testing, it is better drop the first result.
   - b)To reduce the impact of the fluctuation among the tests, it's better test the same scenario more than 5 times and use the average number.
   - c)Between tests, it is better to wait for more than 30 seconds so that the cached pages are flushed to the disk.

Understanding the result

You are expected to get the result something like the below Figure 3.1.4 shows:

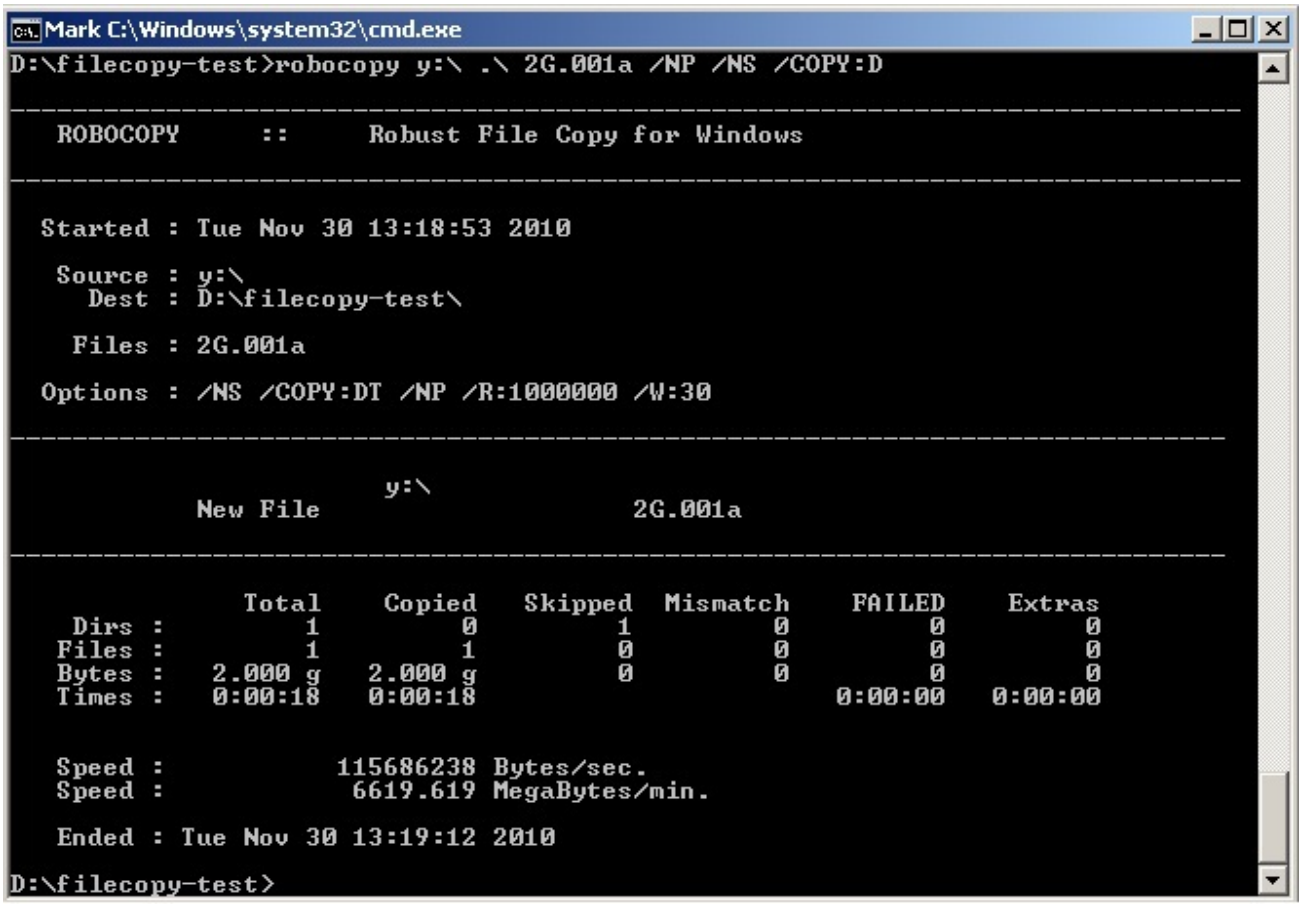


Figure 385. Figure 3.1.4

Here is the example robocopy result of reading 2G file. The number is 6619.619 MegaBytes/min. It means the reading speed is about 110MB/s that can be computed by 6619.619/60.

12.2.8.3.2 Single-core RAID5 single-client

Hardware setup

4 SATA disks will be needed for RAID5. You can compose RAID5 on P2041RDB with 2 PCIe-SATA adaptors, as shown in figure 3.2.1.

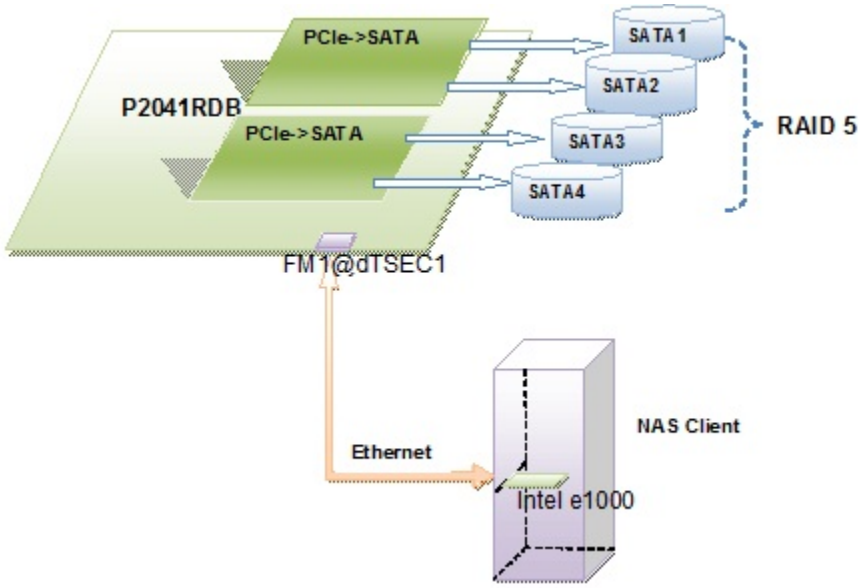


Figure 386. Figure 3.2.1

Test procedures

Please follow the below steps to have RAID5/single-client NAS performance test:

1. Power on the P2041 board, in u-boot, use the following command to disable other 3 cores and only enable 1 core:
   - >cpu disable 3
   - >cpu disable 2
   - >cpu disable 1
2. Continue to boot up the P2041 board into Linux. Type "root" to enter the Linux command line.
3. Setup RAID5, shared folder and IP address on P2041 board with below command. Suppose the shared folder is /smbshare, and EXT2 is used:
   - a)mdadm -C /dev/md0 -amd -R -l5 -n4 /dev/sd[abcd]1
   - b)mkfs.ext2 /dev/md0
   - c)mount /dev/md0 /smbshare
   - d)echo 512 > /sys/block/sda/bdi/read_ahead_kb
   - e)echo 512 > /sys/block/sdb/bdi/read_ahead_kb
   - f)echo 512 > /sys/block/sdc/bdi/read_ahead_kb
   - g)echo 512 > /sys/block/sdd/bdi/read_ahead_kb
   - h)echo 1024 >/sys/block/md0/md/stripe_cache_size
   - i)ifconfig fm1-gb0 192.168.1.100
   - j)fmc –c NAS_cfg_P4080_P5020_P5040_P3041_P2041_P2040.xml –p NAS_policy_P4080_P5020_P5040_P3041_P2041_P2040.xml -a
   - k)ethtool –K fm1-gb0 gro on gso on sg on

   If you want to use EXT4 file system, change above command b) c) to:

   - b') mkfs.ext4 /dev/md0

- c') mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/md0 /smbshare

4. Startup Samba service on board side with command:
   - smbd –s /etc/samba/smb.conf
5. On NAS client side, map the shared folder /smbshare as a windows drive, e.g, drive z. Please refer to the below steps for details, then you will get a new drive as figure 3.2.2.

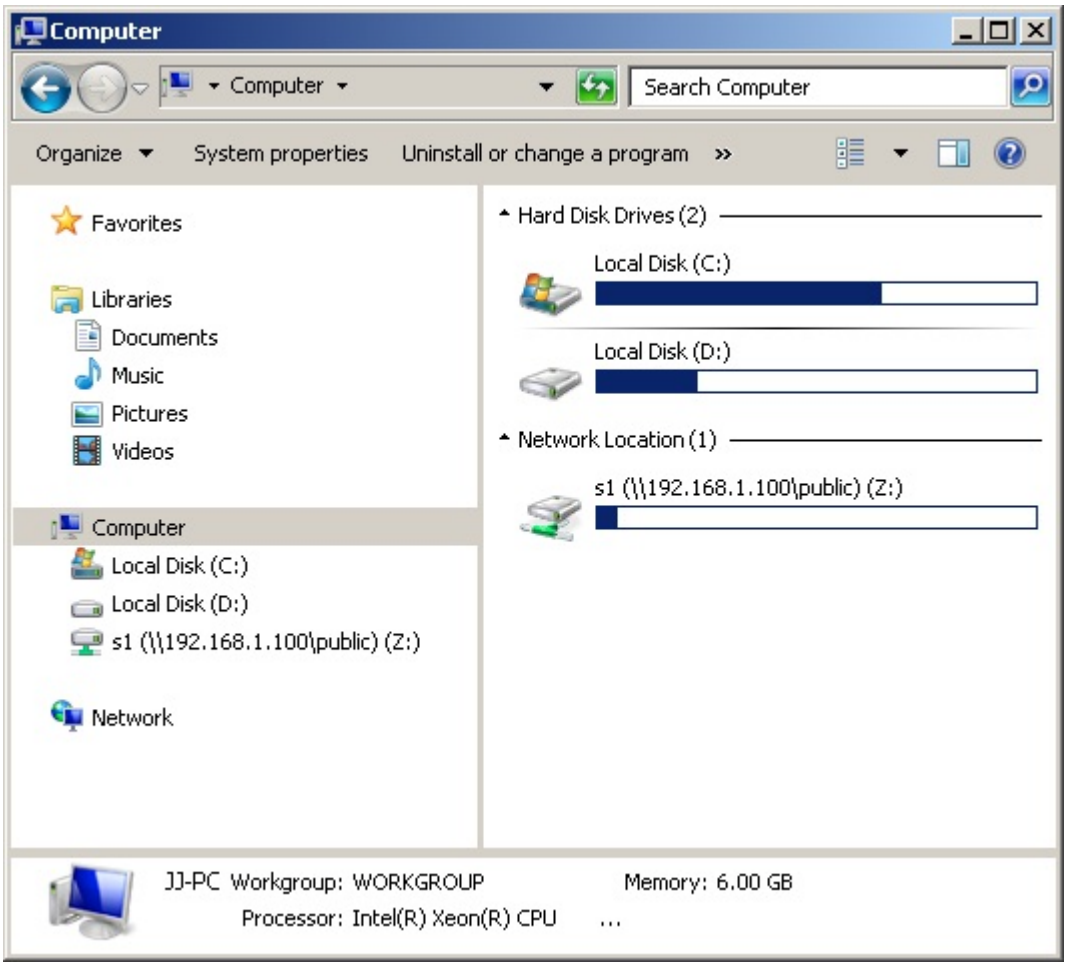


Figure 387. Figure 3.2.2

- a)Connect the NAS client and the board directly by a crossover network cable.
- b)Set up the NAS client IP to 192.168.1.xxx
- c)Open a command prompt window on NAS client and input the following command to map the shared folder as a windows drive:
  - Net use Z: \\192.168.1.100\public

6. Setup up jumbo frame on both client and server side. With jumbo frame, better NAS performance can be achieved. Please skip this step if you want to keep the default 1500 Bytes MTU.

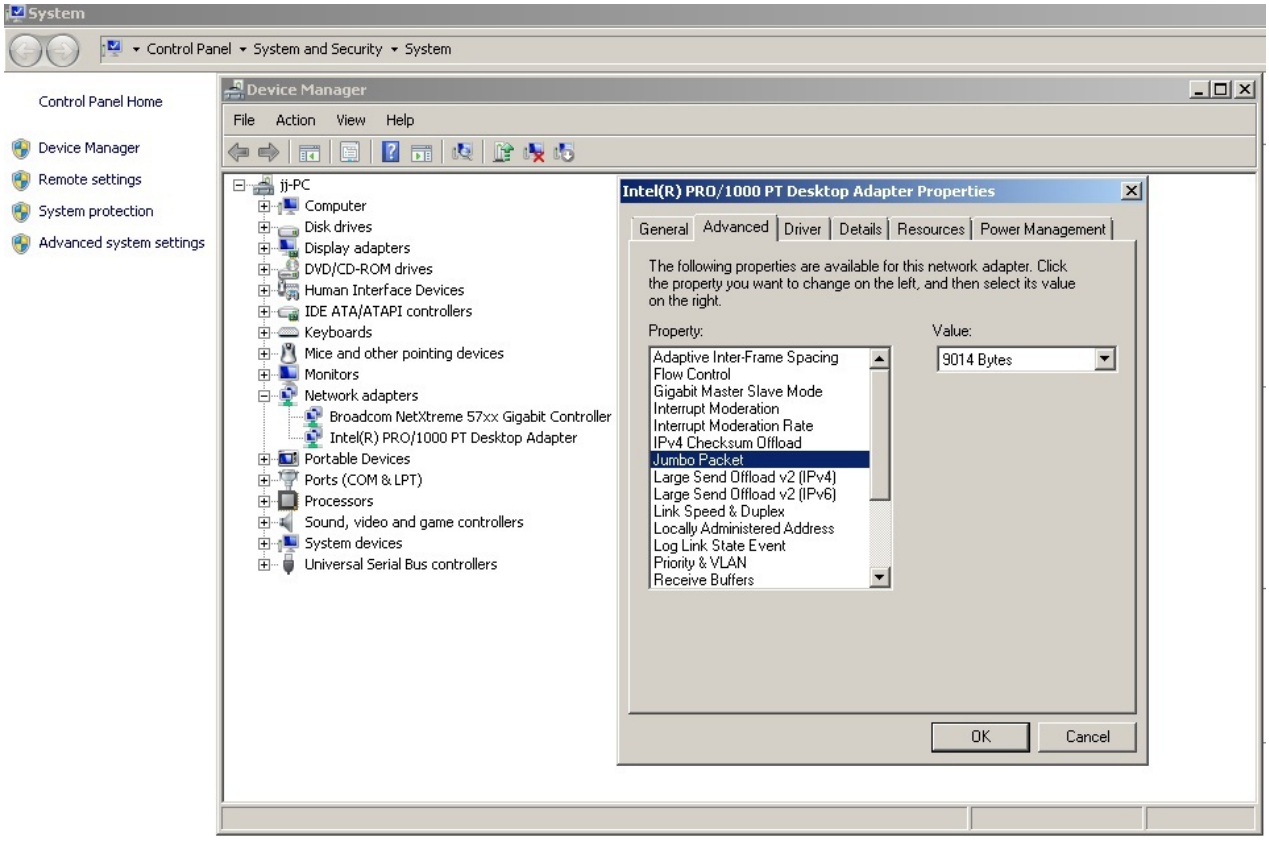


Figure 388. Figure 3.2.3

- a)At the NAS client (windows 7) side, open network and sharing center by selecting Start-> Settings-> Control Panel-> System and Security-> System-> Device Manager-> Intel(R) PRO/1000 PT Desktop Adapter and configure Jumbo Packets as 9014 bytes. Please refer to Figure 3.2.3.
- b)At the NAS server (Linux) side, config the ethernet MTU to 9014 with command:
  - # ifconfig fm1-gb0 mtu 9014

7. Generate a dummy file with 2GB with the following command at NAS client side:
   - prompt> fsutil file createnew 2G.001 2147483648
8. Begin the test with below commands at NAS client side:
   - To test writing performance: Prompt> robocopy . z:\ 2G.001 /NP /NS /COPY:D
   - To test reading performance: Prompt> robocopy z:\ . 2G.001 /NP /NS /COPY:D
9. Recording the performance number:
   - a)To warm up the testing, it is better drop the first result.
   - b)To reduce the impact of the fluctuation among the tests, it's better test the same scenario more than 5 times and use the average number.
   - c)Between tests, it is better to wait for more than 30 seconds so that the cached pages are flushed to the disk.

Understanding the result

You are expected to get the result something like the below Figure 3.2.4 shows:

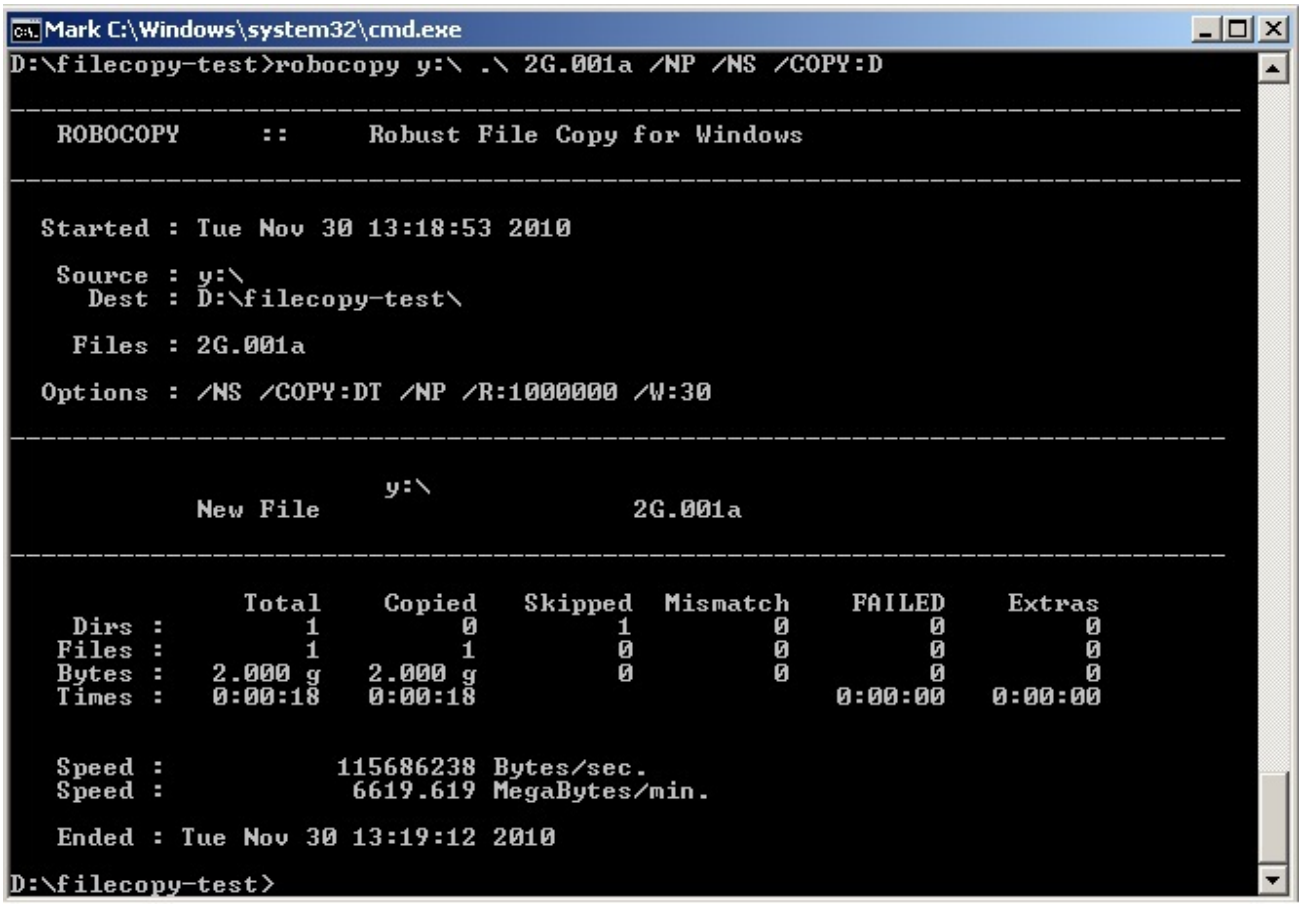


Figure 389. Figure 3.2.4

Here is the example robocopy result of reading 2G file. The number is 6619.619 MegaBytes/min. It means the reading speed is about 110MB/s that can be computed by 6619.619/60.

12.2.8.3.3 Two-core RAID5 single-client

Hardware setup

4 SATA disks will be needed for RAID5. You can compose RAID5 on P2041RDB with 2 PCIe-SATA adaptors, as shown in figure 3.3.1.

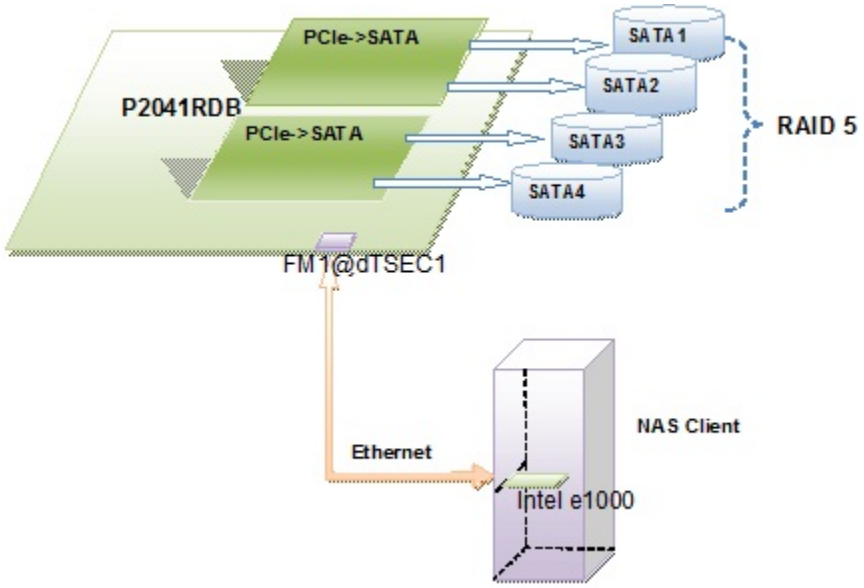


Figure 390. Figure 3.1.1

Test procedures

Please follow the below steps to have RAID5/single-client NAS performance test:

1. Power on the P2041 board, in u-boot, use the following command to disable other 3 cores and only enable 2 cpu core:
   - >cpu disable 3
   - >cpu disable 2
2. Continue to boot up the P2041 board into Linux. Type “root” to enter the Linux command line.
3. Setup RAID5, shared folder and IP address on P2041 board with below command. Suppose the shared folder is /smbshare, and EXT2 is used:
   - a)mdadm -C /dev/md0 -amd -R -l5 -n4 /dev/sd[abcd]1
   - b)mkfs.ext2 /dev/md0
   - c)mount /dev/md0 /smbshare
   - d)echo 512 > /sys/block/sda/bdi/read_ahead_kb
   - e)echo 512 > /sys/block/sdb/bdi/read_ahead_kb
   - f)echo 512 > /sys/block/sdc/bdi/read_ahead_kb
   - g)echo 512 > /sys/block/sdd/bdi/read_ahead_kb
   - h)echo 1024 >/sys/block/md0/md/stripe_cache_size
   - i)ifconfig fm1-gb0 192.168.1.100
   - j)fmc –c NAS_cfg_P4080_P5020_P5040_P3041_P2041_P2040.xml –p NAS_policy_P4080_P5020_P5040_P3041_P2041_P2040.xml -a
   - k)ethtool –K fm1-gb0 gro on gso on sg on

   If you want to use EXT4 file system, change above command b) c) to:

   - b’) mkfs.ext4 /dev/md0
   - c’) mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/md0 /smbshare

4. Startup Samba service on board side with command:
   - smbd –s /etc/samba/smb.conf
5. On NAS client side, map the shared folder /smbshare as a windows drive, e.g, drive z. Please refer to the below steps for details, then you will get a new drive as figure 3.3.2.
   - a)Connect the NAS client and the board directly by a crossover network cable.
   - b)Set up the NAS client IP to 192.168.1.xxx
   - c)Open a command prompt window on NAS client and input the following command to map the shared folder as a windows drive:
     - Net use Z: \\192.168.1.100\public

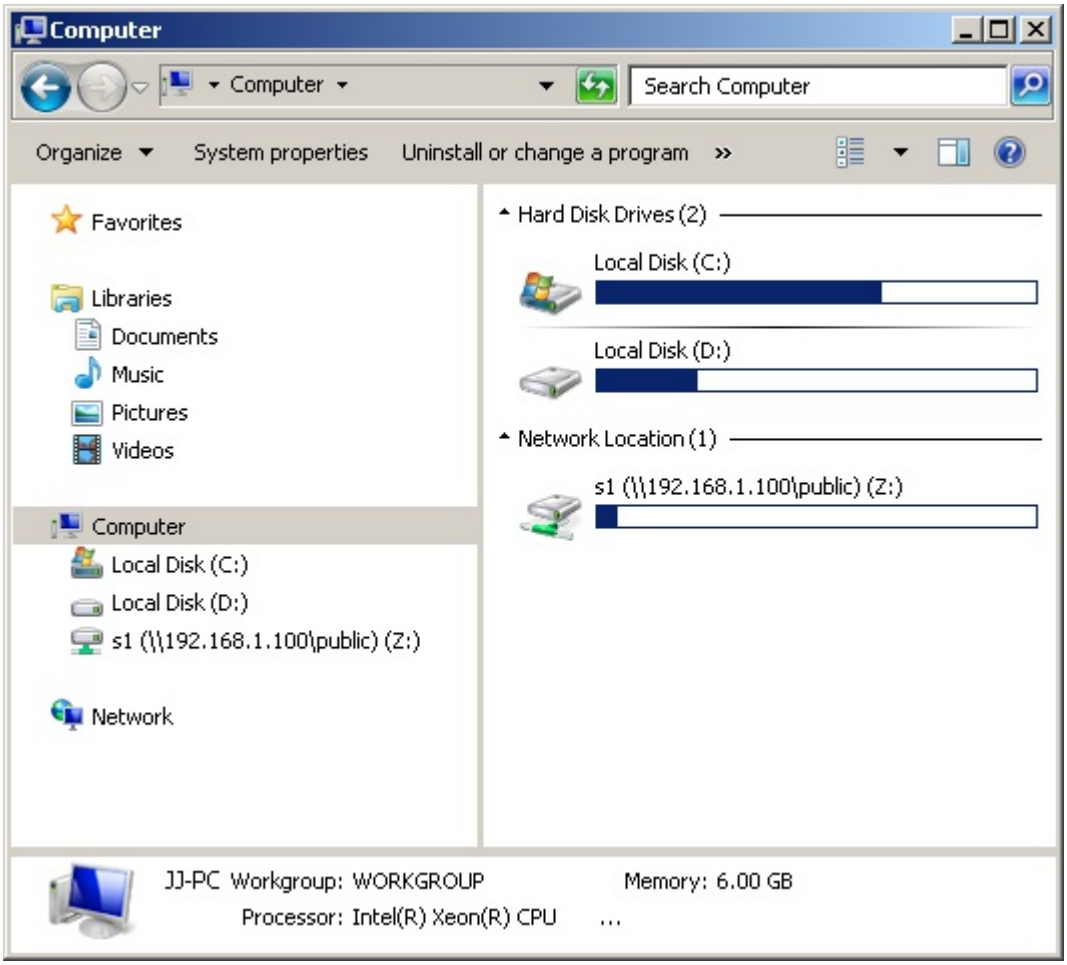


Figure 391. Figure 3.3.2

6. Setup up jumbo frame on both client and server side. With jumbo frame, better NAS performance can be achieved. Please skip this step if you want to keep the default 1500 Bytes MTU.

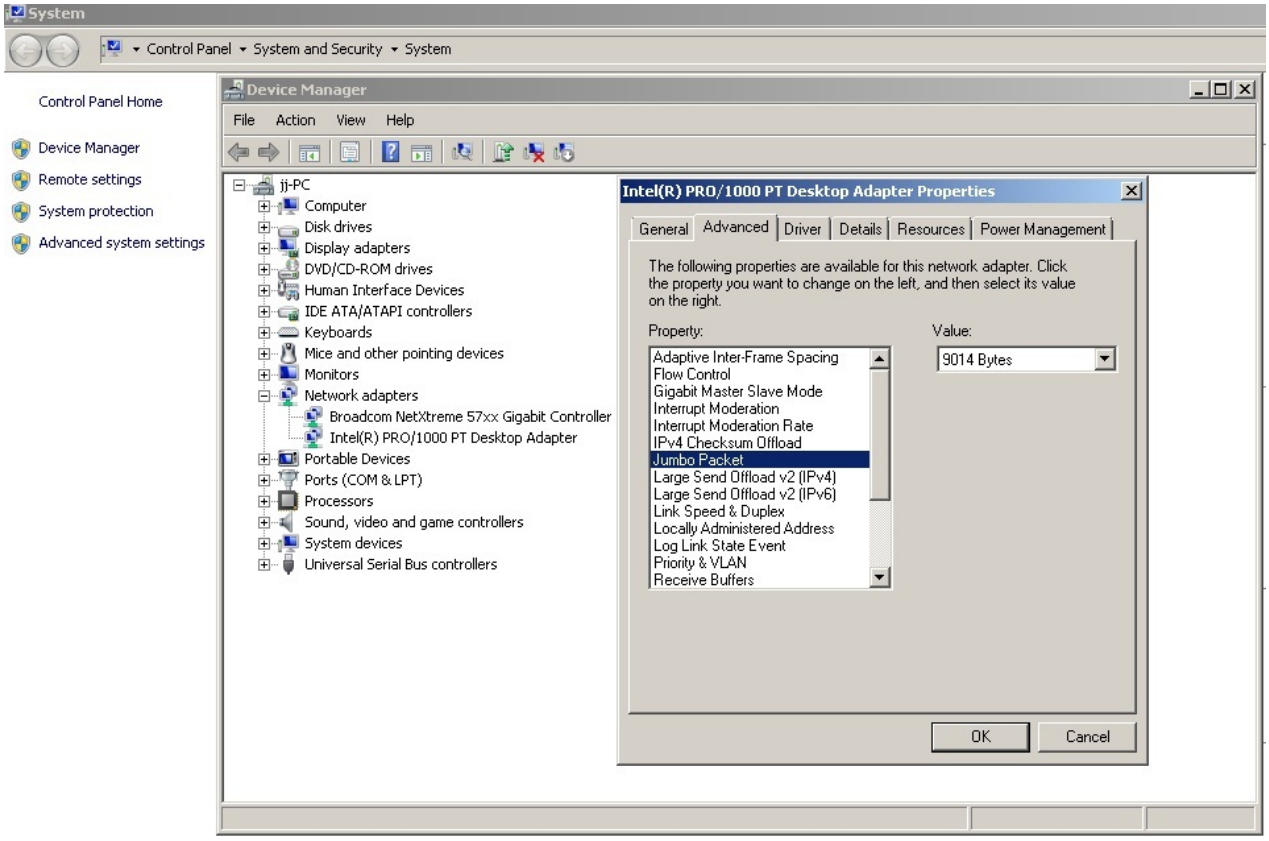


Figure 392. Figure 3.3.3

- a)At the NAS client (windows 7) side, open network and sharing center by selecting Start-> Settings-> Control Panel-> System and Security-> System-> Device Manager-> Intel(R) PRO/1000 PT Desktop Adapter and configure Jumbo Packets as 9014 bytes. Please refer to Figure 3.3.3.
- b)At the NAS server (Linux) side, config the ethernet MTU to 9014 with command:
  - # ifconfig fm1-gb0 mtu 9014

7. Generate a dummy file with 2GB with the following command at NAS client side:
   - prompt> fsutil file createnew 2G.001 2147483648
8. Begin the test with below commands at NAS client side:
   - To test writing performance: Prompt> robocopy . z:\ 2G.001 /NP /NS /COPY:D
   - To test reading performance: Prompt> robocopy z:\ . 2G.001 /NP /NS /COPY:D
9. Recording the performance number:
   - a)To warm up the testing, it is better drop the first result.
   - b)To reduce the impact of the fluctuation among the tests, it's better test the same scenario more than 5 times and use the average number.
   - c)Between tests, it is better to wait for more than 30 seconds so that the cached pages are flushed to the disk.

Understanding the result

You are expected to get the result something like the below Figure 3.3.4 shows:

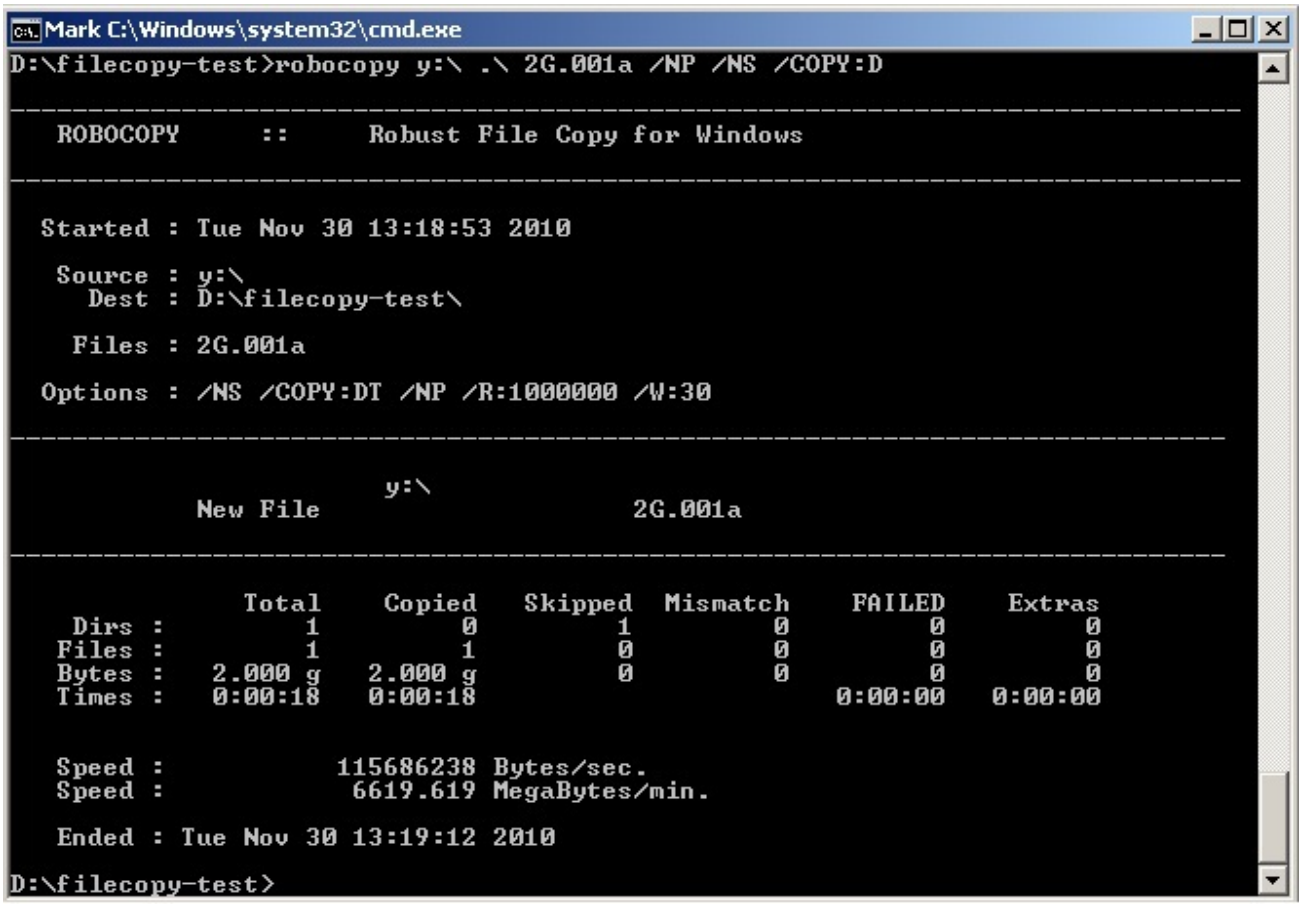


Figure 393. Figure 3.3.4

Here is the example robocopy result of reading 2G file. The number is 6619.619 MegaBytes/min. It means the reading speed is about 110MB/s that can be computed by 6619.619/60.

12.2.8.3.4 Two-core RAID5 two-client

Hardware setup

4 SATA disks will be needed for RAID5. You can compose RAID5 on P2041RDB with 2 PCIe-SATA adaptors. The network connections are shown below:

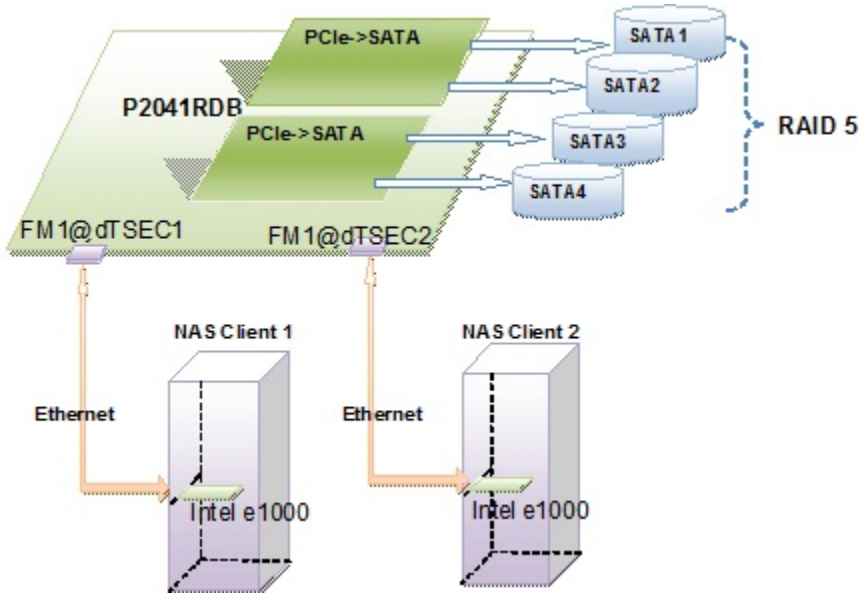


Figure 394. Method 1 is as figure 3.4.1:

Test procedures

Please follow the below steps to have RAID5/two-client NAS performance test:

1. Power on the P2041 board, in u-boot, use the following command to disable other 3 cores and only enable 2 cpu core:
   - >cpu disable 3
   - >cpu disable 2
2. Continue to boot up the P2041 board into Linux. Type “root” to enter the Linux command line.
3. Setup RAID5, shared folder and IP address on P2041 board with below command. Suppose the shared folder is /smbshare, and EXT2 is used:
   - a)mdadm -C /dev/md0 -amd -R -l5 -n4 /dev/sd[abcd]1
   - b)mkfs.ext2 /dev/md0
   - c)mount /dev/md0 /smbshare
   - d)mkdir /smbshare/{s1,s2} -p
   - e)chmod 777 /smbshare
   - f)echo 512 > /sys/block/sda/bdi/read_ahead_kb
   - g)echo 512 > /sys/block/sdb/bdi/read_ahead_kb
   - h)echo 512 > /sys/block/sdc/bdi/read_ahead_kb
   - i)echo 512 > /sys/block/sdd/bdi/read_ahead_kb
   - j)echo 1024 >/sys/block/md0/md/stripe_cache_size
   - k)ifconfig fm1-gb0 192.168.1.100/24
   - l)ifconfig fm1-gb1 192.168.2.100/23
   - m)fmc –c NAS_cfg_P4080_P5020_P5040_P3041_P2041_P2040.xml –p NAS_policy_P4080_P5020_P5040_P3041_P2041_P2040.xml -a
   - n)ethtool –K fm1-gb0 gro on gso on sg on

- o)ethtool –K fm1-gb1 gro on gso on sg on

If you want to use EXT4 file system, change above command b) c) to:

- b') mkfs.ext4 /dev/md0
- c') mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/md0 /smbshare

4. Startup Samba service on board side with command:
   - smbd –s /etc/samba/smb.conf
5. On each NAS client side, map the shared folder /smbshare as 2 windows drives, e.g, drive z. Please refer to the below steps for details, then you will get 2 new drive as figure 3.4.2.
   - a)Connect the NAS client and the board directly by 2 crossover network cable.
   - b)Set up the NAS client IP to 192.168.1.xxx/24 and 192.168.2.yyy/23
   - c)Open a command prompt window on NAS client and input the following command to map the shared folders as 2 windows drives:
     - Net use Z: \\192.168.1.100\public\s1
     - Net use Y: \\192.168.2.100\public\s2

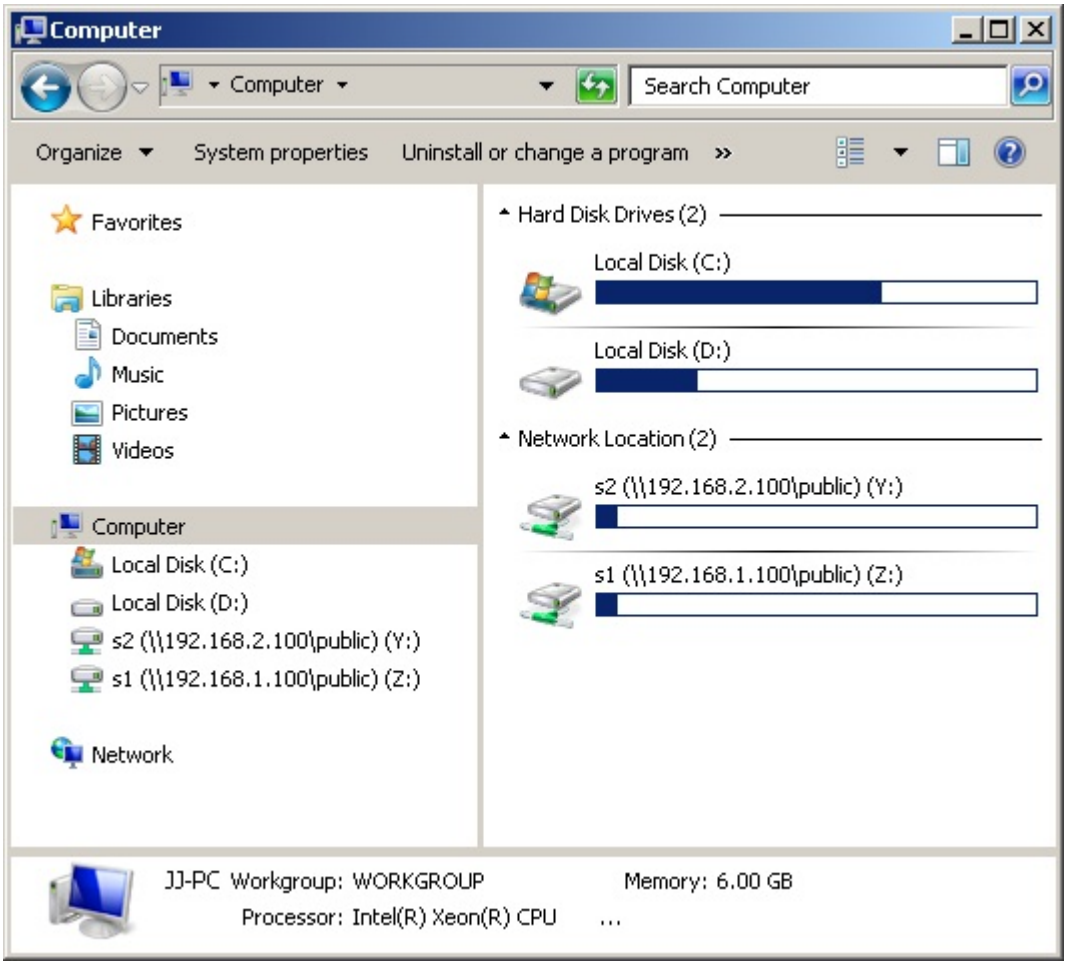


Figure 395. Figure 3.4.2

6. Setup up jumbo frame on both client and server side. With jumbo frame, better NAS performance can be achieved. Please skip this step if you want to keep the default 1500 Bytes MTU.

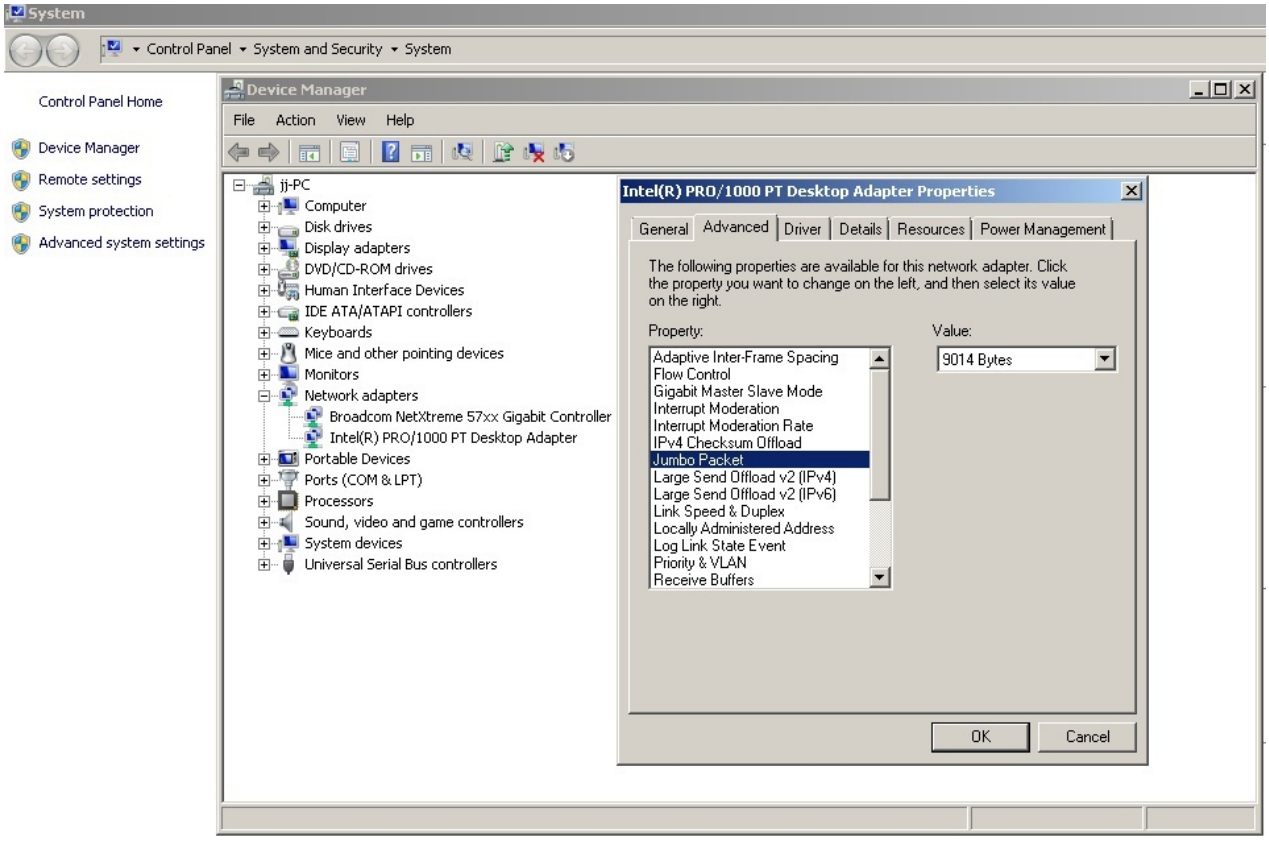


Figure 396. Figure 3.4.3

- a)At the NAS client (windows 7) side, open network and sharing center by selecting Start-> Settings-> Control Panel->System and Security->System->Device Manager->" Intel(R) PRO/1000 PT Quad Port Server Adapter " and configure Jumbo Packets as 9014 bytes. Please refer to Figure 3.4.3.
- b)At the NAS server (Linux) side, config the ethernet MTU to 9014 with command:
  - # ifconfig fm1-gb0 mtu 9014
  - # ifconfig fm1-gb1 mtu 9014

7. Generate 2 dummy files with 2GB with the following command in two NAS client side command prompt windows, it's better put the two files in different hard disk:
   - prompt> fsutil file createnew 2G.001 2147483648
   - prompt> fsutil file createnew 2G.002 2147483648
8. Begin the test with below commands at NAS client side:
   - To test writing performance:
     - Run the following command in the 1st command prompt window: prompt> robocopy C:\ Z:\ 2G.001 /NP /NS /COPY:D
     - Run the following command in the 2nd command prompt window: prompt> robocopy D:\ Y:\ 2G.002 /NP /NS /COPY:D
   - To test reading performance:
     - Run the following command in the 1st command prompt window: prompt> robocopy Z:\ C:\ 2G.001 /NP /NS /COPY:D
     - Run the following command in the 2nd command prompt window: prompt> robocopy Y:\ D:\ 2G.002 /NP /NS /COPY:D
9. Recording the performance number:
   - a)To warm up the testing, it is better drop the first result.

- b)To reduce the impact of the fluctuation among the tests, it's better test the same scenario more than 5 times and use the average number.
- c)Between tests, it is better to wait for more than 30 seconds so that the cached pages are flushed to the disk.

Understanding the result

You are expected to get the result something like the below Figure 3.4.4 shows:

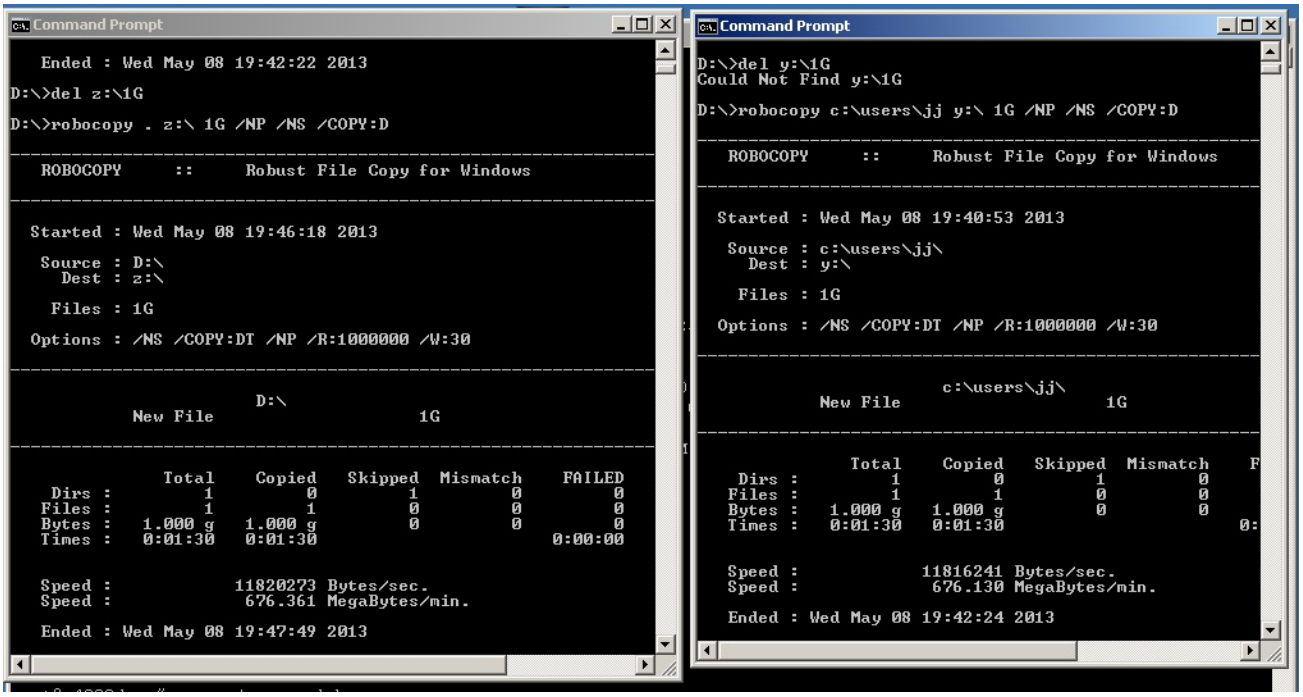


Figure 397. Figure 3.4.4

Here are the example robocopy results of reading 1G file in 2 command prompt windows of the NAS client. The numbers are 676.361 MegaBytes/min and 676.130 MegaBytes/min. It means the 1st client reading speed is about 11MB/s that can be computed by (676.361 MegaBytes/min) / (60second/min) and the 2nd client reading speed is about 11MB/s that can be computed by (676.130 MegaBytes/min) / (60 second/min). You should calculate the last result by adding two single client result:

The last result = $\sum_{i=1}^{2} result\ \ in\ \ client_i$ = 11MB/s +11MB/s =22MB/s

12.2.8.3.5 Four-core RAID5 single-client

Hardware setup

4 SATA disks will be needed for RAID5. You can compose RAID5 on P2041RDB with 2 PCIe-SATA adaptors, as shown in figure 3.5.1.

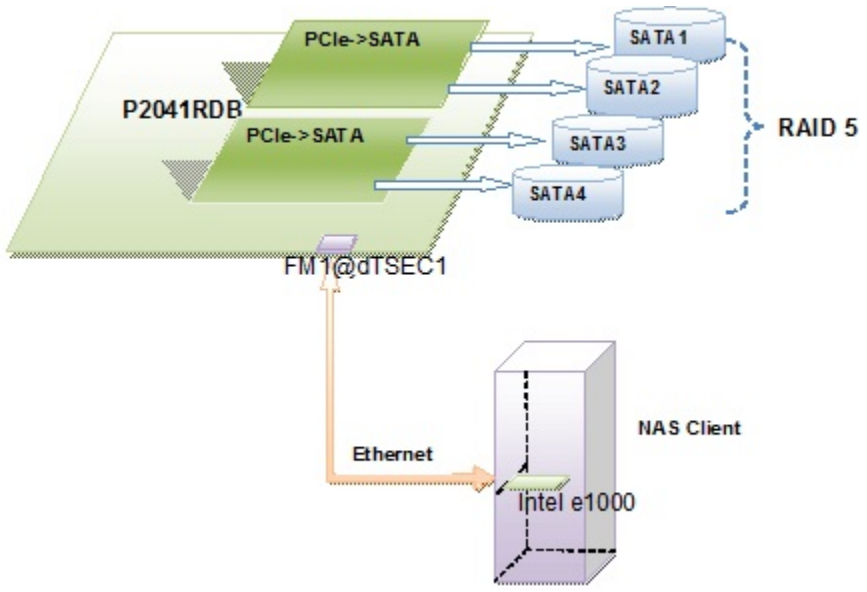


Figure 398. Figure 3.5.1

Test procedures

Please follow the below steps to have Four-core RAID5/single-client NAS performance test:

1. 1.Power on the P2041 board and make sure 4 cores enabled.
2. Continue to boot up the P2041 board into Linux. Type "root" to enter the Linux command line.
3. Setup RAID5, shared folder and IP address on P2041 board with below command. Suppose the shared folder is /smbshare, and EXT2 is used:
   - a)mdadm -C /dev/md0 -amd -R -l5 -n4 /dev/sd[abcd]1
   - b)mkfs.ext2 /dev/md0
   - c)mount /dev/md0 /smbshare
   - d)echo 512 > /sys/block/sda/bdi/read_ahead_kb
   - e)echo 512 > /sys/block/sdb/bdi/read_ahead_kb
   - f)echo 512 > /sys/block/sdc/bdi/read_ahead_kb
   - g)echo 512 > /sys/block/sdd/bdi/read_ahead_kb
   - h)echo 1024 >/sys/block/md0/md/stripe_cache_size
   - i)ifconfig fm1-gb0 192.168.1.100
   - j)fmc –c NAS_cfg_P4080_P5020_P5040_P3041_P2041_P2040.xml –p NAS_policy_P4080_P5020_P5040_P3041_P2041_P2040.xml -a
   - k)ethtool –K fm1-gb0 gro on gso on sg on

   If you want to use EXT4 file system, change above command b) c) to:

   - b') mkfs.ext4 /dev/md0
   - c') mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/md0 /smbshare
4. Startup Samba service on board side with command:
   - smbd –s /etc/samba/smb.conf

5. 5.On NAS client side, map the shared folder /smbshare as a windows drive, e.g, drive z. Please refer to the below steps for details, then you will get a new drive as figure 3.5.2.
   - a)Connect the NAS client and the board directly by a crossover network cable.
   - b)Set up the NAS client IP to 192.168.1.xxx
   - c)Open a command prompt window on NAS client and input the following command to map the shared folder as a windows drive:
     - Net use Z: \\192.168.1.100\public

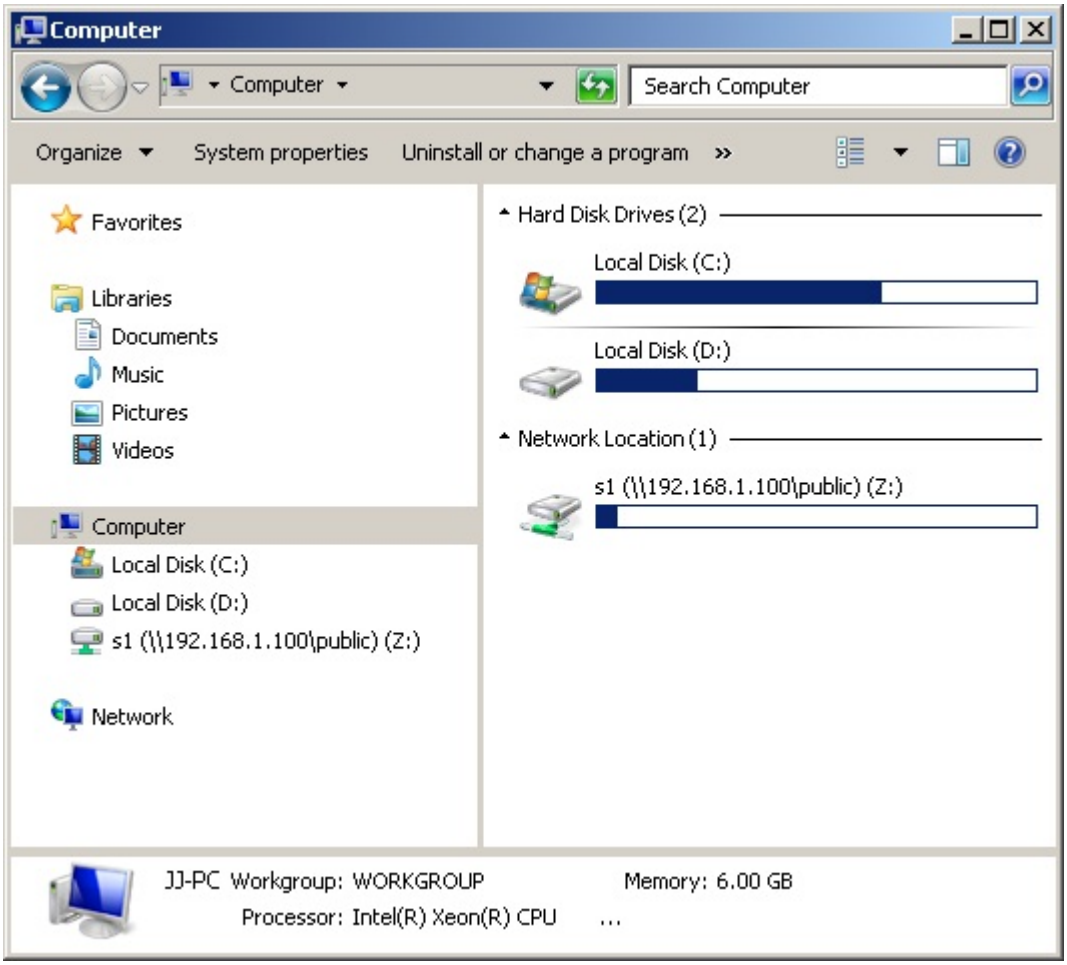


Figure 399. Figure 3.5.2

6. Setup up jumbo frame on both client and server side. With jumbo frame, better NAS performance can be achieved. Please skip this step if you want to keep the default 1500 Bytes MTU.

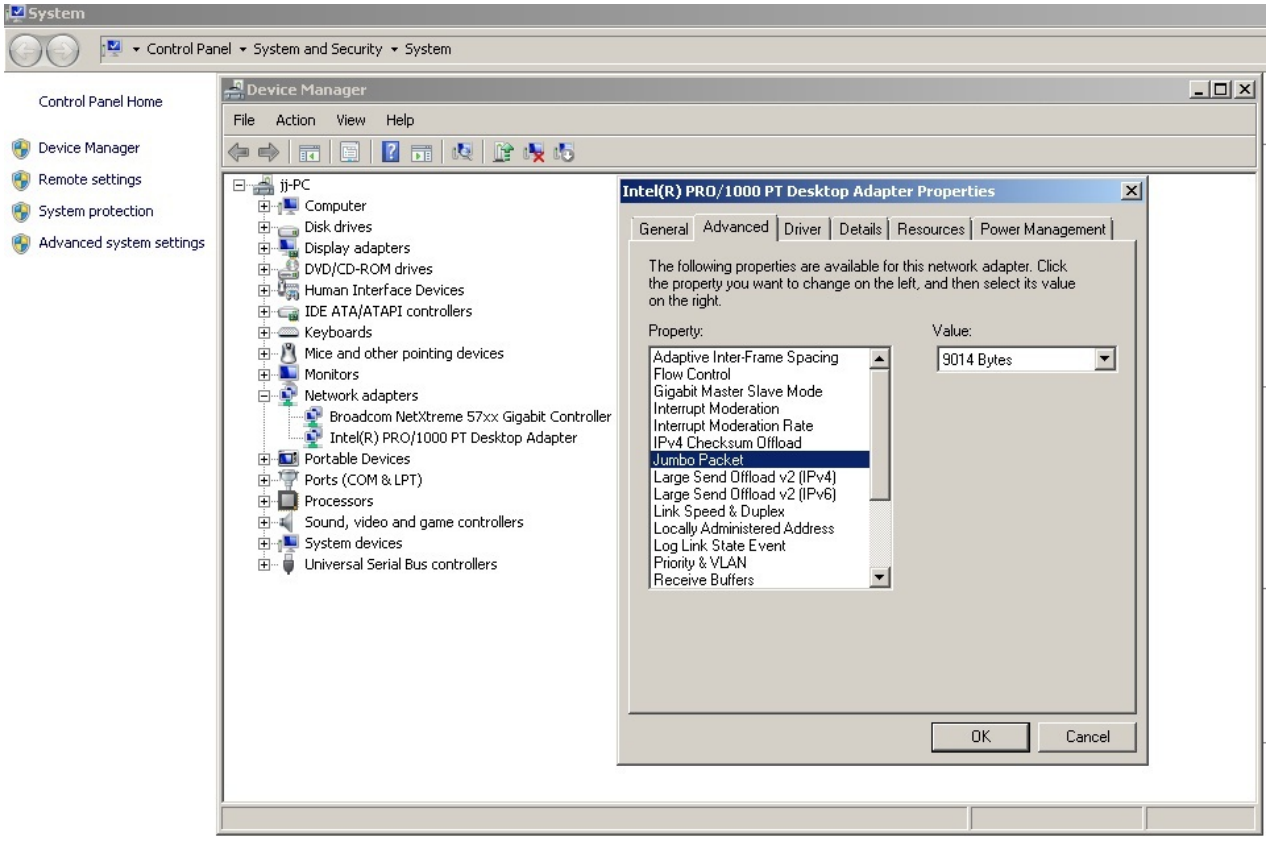


Figure 400. Figure 3.5.3

- a)At the NAS client (windows 7) side, open network and sharing center by selecting Start-> Settings-> Control Panel->System and Security->System->Device Manager->Intel(R) PRO/1000 PT Desktop Adapter and configure Jumbo Packets as 9014 bytes. Please refer to Figure 3.5.3.
- b)At the NAS server (Linux) side, config the ethernet MTU to 9014 with command:
  - # ifconfig fm1-gb0 mtu 9014

7. Generate a dummy file with 2GB with the following command at NAS client side:
   - prompt> fsutil file createnew 2G.001 2147483648
8. Begin the test with below commands at NAS client side:
   - To test writing performance: Prompt> robocopy . z:\ 2G.001 /NP /NS /COPY:D
   - To test reading performance: Prompt> robocopy z:\ . 2G.001 /NP /NS /COPY:D
9. Recording the performance number:
   - a)To warm up the testing, it is better drop the first result.
   - b)To reduce the impact of the fluctuation among the tests, it's better test the same scenario more than 5 times and use the average number.
   - c)Between tests, it is better to wait for more than 30 seconds so that the cached pages are flushed to the disk.

Understanding the result

You are expected to get the result something like the below Figure 3.5.4 shows:

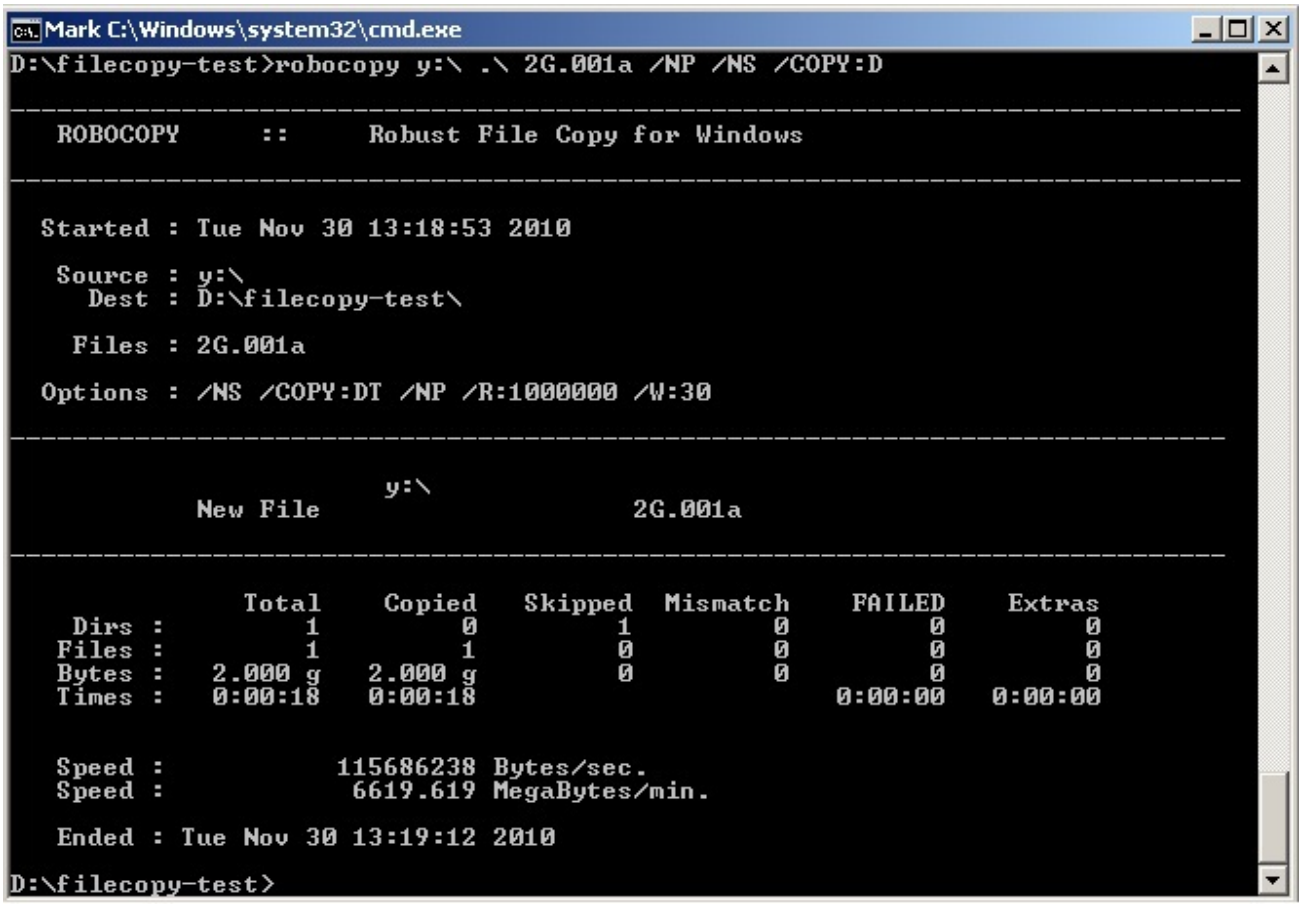


Figure 401. Figure 3.5.4

Here is the example robocopy result of reading 2G file. The number is 6619.619 MegaBytes/min. It means the reading speed is about 110MB/s that can be computed by 6619.619/60.

12.2.8.3.6 Four-core RAID5 two-client

Hardware setup

4 SATA disks will be needed for RAID5. You can compose RAID5 on P2041RDB with 2 PCIe-SATA adaptors. The network connection is shown as below

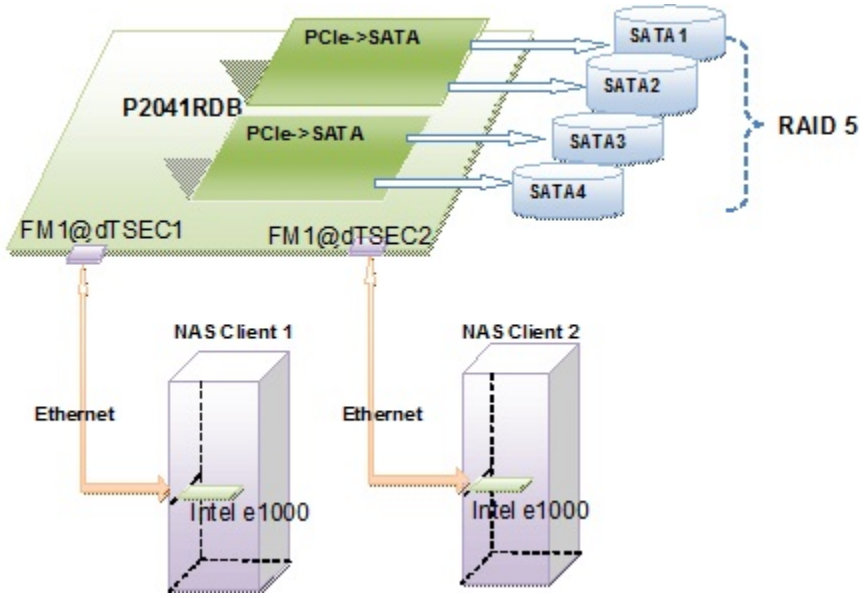


Figure 402. Method 1 is as figure 3.6.1:

Test procedures

Please follow the below steps to have RAID5/two-client NAS performance test:

1. Power on the P2041 board and make sure 4 cores enabled.
2. Continue to boot up the P2041 board into Linux. Type "root" to enter the Linux command line.
3. Setup RAID5, shared folder and IP address on P2041 board with below command. Suppose the shared folder is /smbshare, and EXT2 is used:
   - a)mdadm -C /dev/md0 -amd -R -l5 -n4 /dev/sd[abcd]1
   - b)mkfs.ext2 /dev/md0
   - c)mount /dev/md0 /smbshare
   - d)mkdir /smbshare/{s1,s2} -p
   - e)chmod -R 777 /smbshare
   - f)echo 512 > /sys/block/sda/bdi/read_ahead_kb
   - g)echo 512 > /sys/block/sdb/bdi/read_ahead_kb
   - h)echo 512 > /sys/block/sdc/bdi/read_ahead_kb
   - i)echo 512 > /sys/block/sdd/bdi/read_ahead_kb
   - j)echo 1024 >/sys/block/md0/md/stripe_cache_size
   - k)ifconfig fm1-gb0 192.168.1.100/24
   - l)ifconfig fm1-gb1 192.168.2.100/23
   - m)fmc –c NAS_cfg_P4080_P5020_P5040_P3041_P2041_P2040.xml –p NAS_policy_P4080_P5020_P5040_P3041_P2041_P2040.xml -a
   - n)ethtool –K fm1-gb0 gro on gso on sg on
   - o)ethtool –K fm1-gb1 gro on gso on sg on

   If you want to use EXT4 file system, change above command b) c) to:

- b') mkfs.ext4 /dev/md0
- c') mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/md0 /smbshare

4. Startup Samba service on board side with command:
   - smbd –s /etc/samba/smb.conf
5. On each NAS client side, map the shared folders /smbshare/{s1,s2} as 2 windows drive, e.g, drive Z, and drive Y. Please refer to the below steps for details, then you will get 2 new drive as figure 3.6.2.
   - a)Connect the NAS client and the board directly by 2 crossover network cable.
   - b)Set up the NAS client IP to 192.168.1.xxx/24 and 192.168.2.yyy/23
   - c)Open a command prompt window on NAS client and input the following command to map the shared folders as 2 windows drives:
     - Net use Z: \\192.168.1.100\public\s1
     - Net use Y: \\192.168.2.100\public\s2

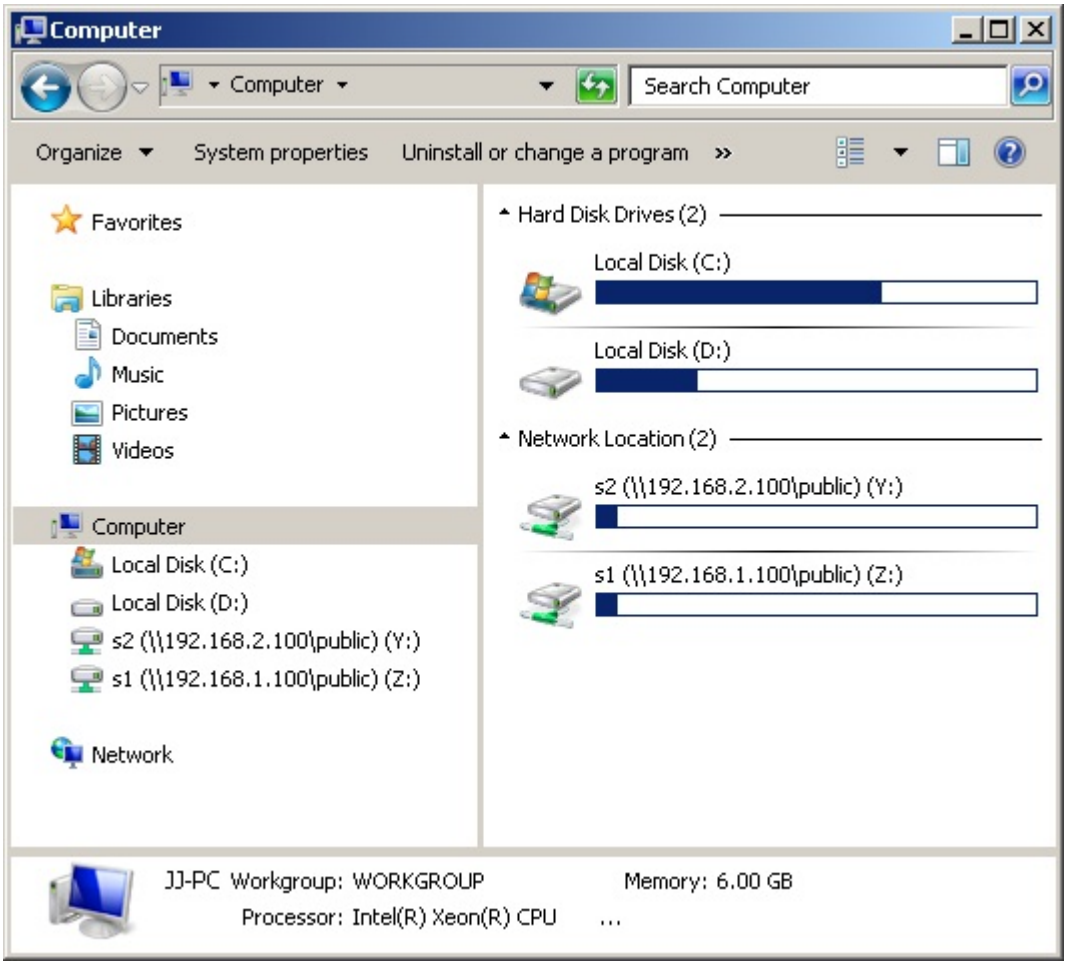


Figure 403. Figure 3.6.2

6. Setup up jumbo frame on both client and server side. With jumbo frame, better NAS performance can be achieved. Please skip this step if you want to keep the default 1500 Bytes MTU.

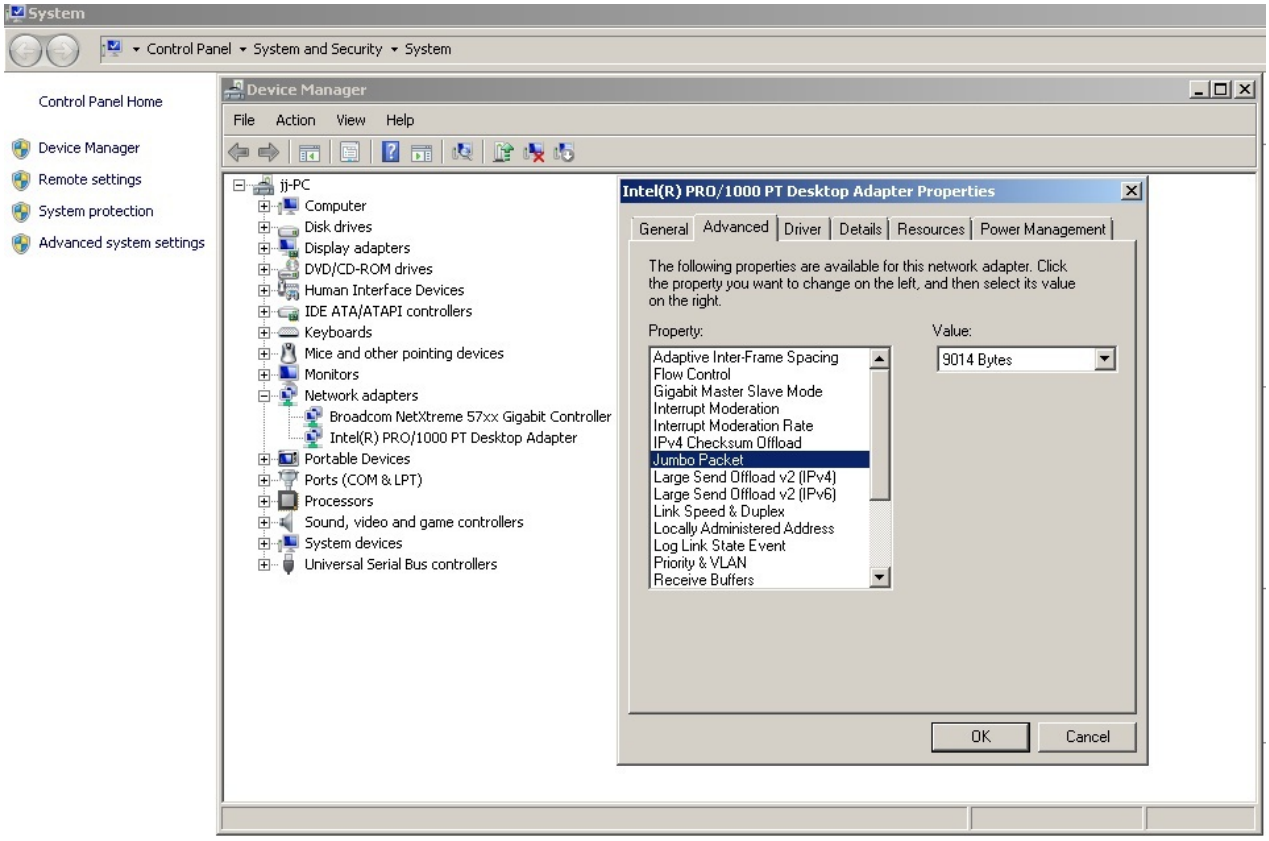


Figure 404. Figure 3.6.3

- a)At the NAS client (windows 7) side, open network and sharing center by selecting Start-> Settings-> Control Panel->System and Security->System->Device Manager->" Intel(R) PRO/1000 PT Quad Port Server Adapter " and configure Jumbo Packets as 9014 bytes. Please refer to Figure 3.6.3.
- b)At the NAS server (Linux) side, config the ethernet MTU to 9014 with command:
  - # ifconfig fm1-gb0 mtu 9014
  - # ifconfig fm1-gb1 mtu 9014

7. Generate 2 dummy files with 2GB with the following command in two NAS client side command prompt windows, it's better put the two files in different hard disk:
   - prompt> fsutil file createnew 2G.001 2147483648
   - prompt> fsutil file createnew 2G.002 2147483648
8. Begin the test with below commands at NAS client side:
   - To test writing performance:
     - Run the following command in the 1st command prompt window: prompt> robocopy C:\ Z:\ 2G.001 /NP /NS /COPY:D
     - Run the following command in the 2nd command prompt window: prompt> robocopy D:\ Y:\ 2G.002 /NP /NS /COPY:D
   - To test reading performance:
     - Run the following command in the 1st command prompt window: prompt> robocopy Z:\ C:\ 2G.001 /NP /NS /COPY:D
     - Run the following command in the 2nd command prompt window: prompt> robocopy Y:\ D:\ 2G.002 /NP /NS /COPY:D
9. Recording the performance number:
   - a)To warm up the testing, it is better drop the first result.

- b)To reduce the impact of the fluctuation among the tests, it's better test the same scenario more than 5 times and use the average number.
- c)Between tests, it is better to wait for more than 30 seconds so that the cached pages are flushed to the disk.

Understanding the result

You are expected to get the result something like the below Figure 3.6.4 shows:

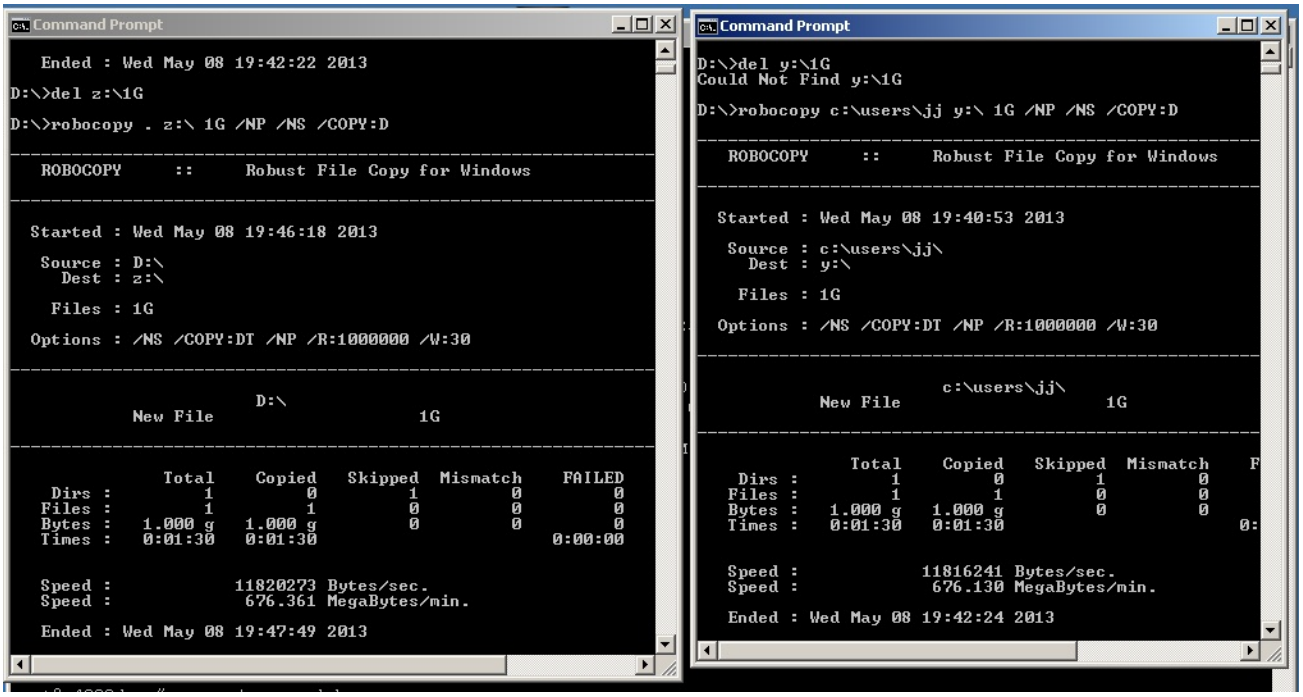


Figure 405. Figure 3.6.4

Here are the example robocopy results of reading 1G file in 2 command prompt windows of the NAS client. The numbers are 676.361 MegaBytes/min and 676.130 MegaBytes/min. It means the 1st client reading speed is about 11MB/s that can be computed by (676.361 MegaBytes/min) / (60second/min) and the 2nd client reading speed is about 11MB/s that can be computed by (676.130 MegaBytes/min) / (60 second/min). You should calculate the last result by adding two single client result:

The last result = $\sum_{i=1}^{2} result\ \ in\ \ client_i$ = 11MB/s +11MB/s =22MB/s

12.2.8.3.7 Four-core RAID5 four-client

Hardware setup

4 SATA disks will be needed for RAID5. You can compose RAID5 on P2041RDB with 2 PCIe-SATA adaptors. The network connection is shown below.

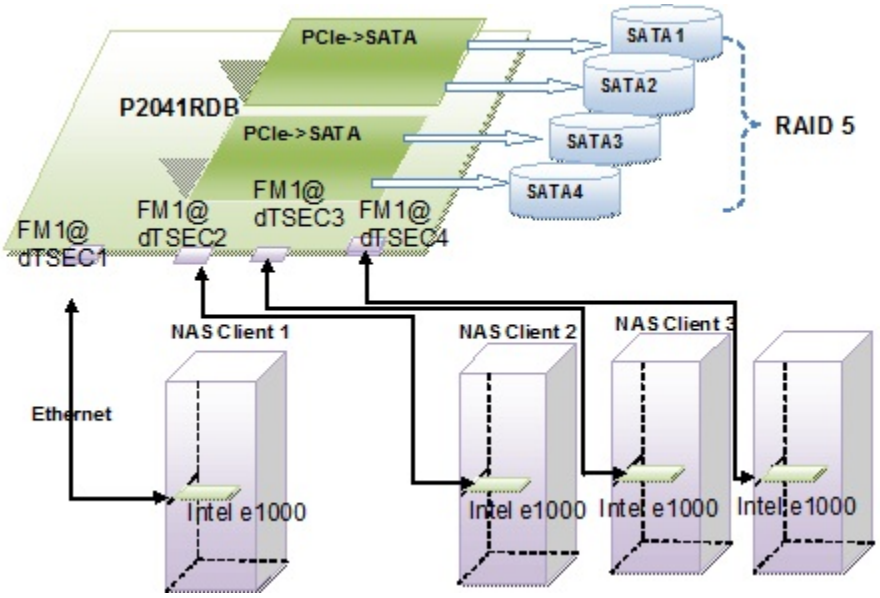


Figure 406. Method 1 is as figure 3.7.1:

Test procedures

Please follow the below steps to have RAID5/two-client NAS performance test:

1. Power on the P2041 board and make sure 4 cores enabled.
2. Continue to boot up the P2041 board into Linux. Type "root" to enter the Linux command line.
3. Setup RAID5, shared folder and IP address on P2041 board with below command. Suppose the shared folder is /smbshare, and EXT2 is used:
   - a)mdadm -C /dev/md0 -amd -R -l5 -n4 /dev/sd[abcd]1
   - b)mkfs.ext2 /dev/md0
   - c)mount /dev/md0 /smbshare
   - d)mkdir -p /smbshare/{s1,s2,s3,s4}
   - e)chmod -R 777 /smbshare
   - f)echo 512 > /sys/block/sda/bdi/read_ahead_kb
   - g)echo 512 > /sys/block/sdb/bdi/read_ahead_kb
   - h)echo 512 > /sys/block/sdc/bdi/read_ahead_kb
   - i)echo 512 > /sys/block/sdd/bdi/read_ahead_kb
   - j)echo 1024 >/sys/block/md0/md/stripe_cache_size
   - k)ifconfig fm1-gb0 192.168.1.100/24
   - l)ifconfig fm1-gb1 192.168.2.100/23
   - m)ifconfig fm1-gb2 192.168.4.100/22
   - n)ifconfig fm1-gb3 192.168.8.100/21
   - o)fmc –c NAS_cfg_P4080_P5020_P5040_P3041_P2041_P2040.xml –p NAS_policy_P4080_P5020_P5040_P3041_P2041_P2040.xml -a
   - p)ethtool –K fm1-gb0 gro on gso on sg on

- q)ethtool –K fm1-gb1 gro on gso on sg on
- r)ethtool –K fm1-gb2 gro on gso on sg on
- s)ethtool –K fm1-gb3 gro on gso on sg on

If you want to use EXT4 file system, change above command b) c) to:

- b') mkfs.ext4 /dev/md0
- c') mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/md0 /smbshare

4. Startup Samba service on board side with command:
   - smbd –s /etc/samba/smb.conf
5. On each NAS client side, map the shared folder /smbshare as 4 windows drives, e.g, drive Z, drive Y, drive X, drive W. Please refer to the below steps for details, then you will get 4 new drive as figure 3.7.2.
   - a)Connect the NAS client and the board directly by 4 crossover network cable.
   - b)Set up the NAS client IP to 192.168.1.xxx/24 ,192.168.2.yyy/23, 192.168.4.zzz/22, and 192.168.8.www/21
   - c)Open a command prompt window on NAS client and input the following command to map the shared folders as 4 windows drives:
     - Net use Z: \\192.168.1.100\public\s1
     - Net use Y: \\192.168.2.100\public\s2
     - Net use X: \\192.168.4.100\public\s3
     - Net use W: \\192.168.8.100\public\s4

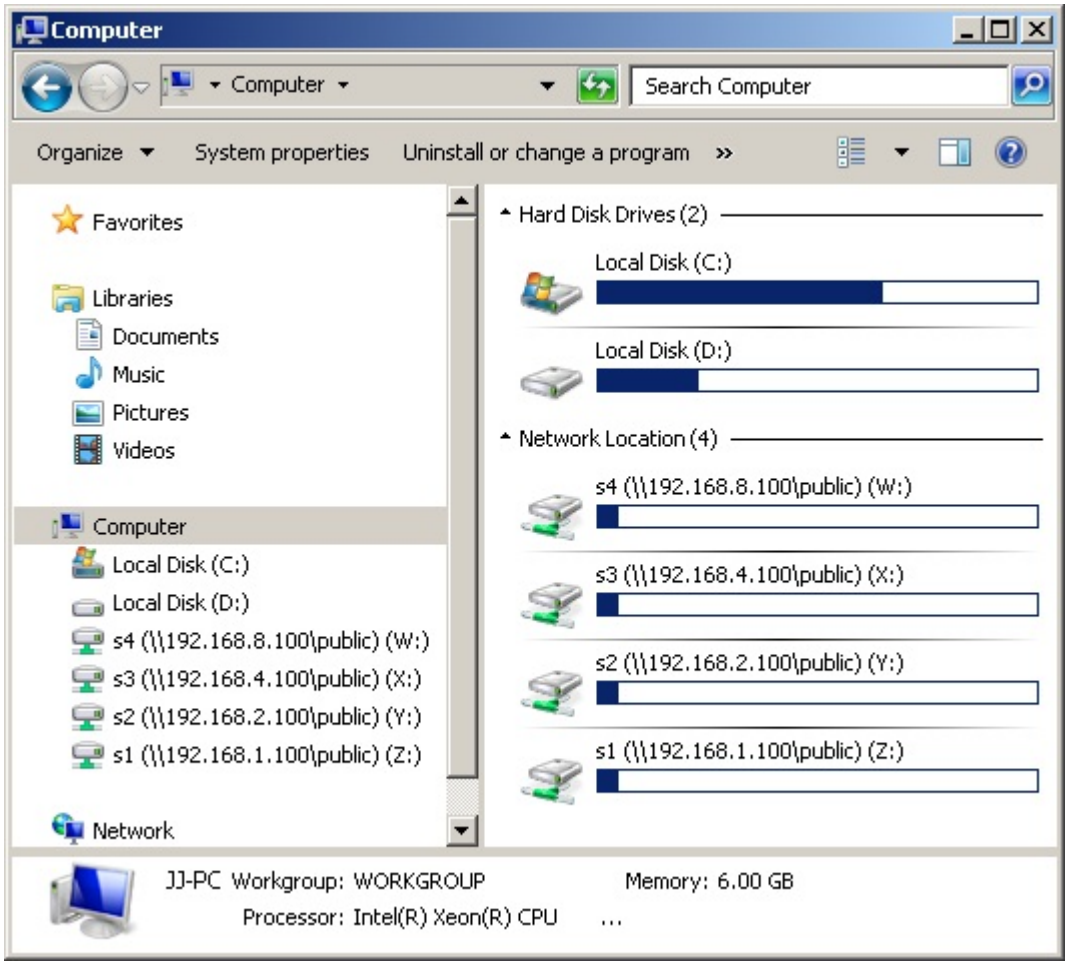


Figure 407. Figure 3.7.2

6. Setup up jumbo frame on both client and server side. With jumbo frame, better NAS performance can be achieved. Please skip this step if you want to keep the default 1500 Bytes MTU.
   - a)At the NAS client (windows 7) side, open network and sharing center by selecting Start-> Settings-> Control Panel->System and Security->System->Device Manager->" Intel(R) PRO/1000 PT Quad Port Server Adapter " and configure Jumbo Packets as 9014 bytes. Please refer to Figure 3.7.3.
   - b)At the NAS server (Linux) side, config the ethernet MTU to 9014 with command:
     - # ifconfig fm1-gb0 mtu 9014
     - # ifconfig fm1-gb1 mtu 9014
     - # ifconfig fm1-gb2 mtu 9014
     - # ifconfig fm1-gb3 mtu 9014

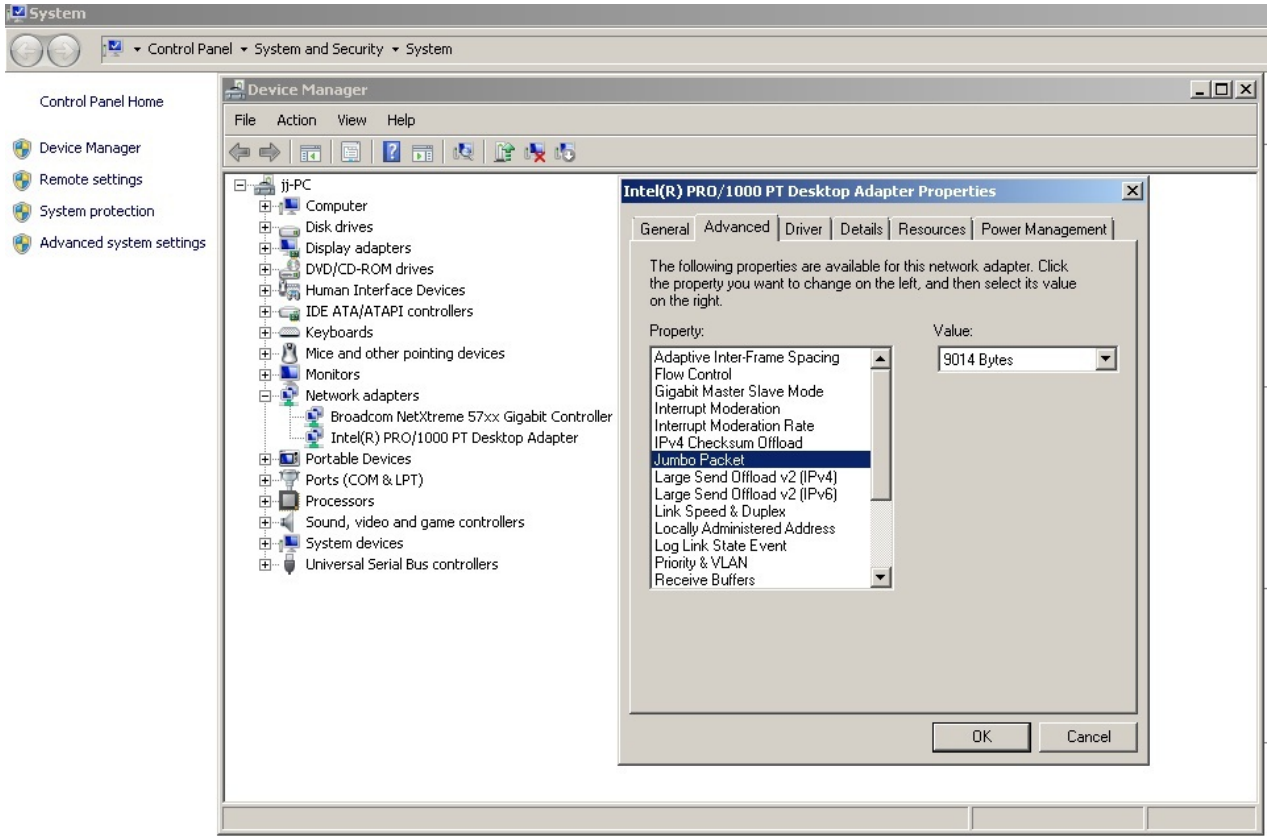


Figure 408. Figure 3.7.3

7. Generate 4 dummy files with 2GB with the following command in 4 NAS client side command prompt windows, it's better put the 4 files in different hard disk:
   - prompt> fsutil file createnew 2G.001 2147483648
   - prompt> fsutil file createnew 2G.002 2147483648
   - prompt> fsutil file createnew 2G.003 2147483648
   - prompt> fsutil file createnew 2G.004 2147483648
8. Begin the test with below commands at NAS client side:
   - To test writing performance:
     - Run the following command in the 1st command prompt window: prompt> robocopy C:\ Z:\ 2G.001 /NP /NS /COPY:D
     - Run the following command in the 2nd command prompt window: prompt> robocopy D:\ Y:\ 2G.002 /NP /NS /COPY:D
     - Run the following command in the 3rd command prompt window: prompt> robocopy E:\ X:\ 2G.003 /NP /NS /COPY:D
     - Run the following command in the 4th command prompt window: prompt> robocopy F:\ W:\ 2G.004 /NP /NS /COPY:D
   - To test reading performance:
     - Run the following command in the 1st command prompt window: prompt> robocopy Z:\ C:\ 2G.001 /NP /NS /COPY:D
     - Run the following command in the 2nd command prompt window: prompt> robocopy Y:\ D:\ 2G.002 /NP /NS /COPY:D
     - Run the following command in the 3rd command prompt window: prompt> robocopy X:\ E:\ 2G.003 /NP /NS /COPY:D
     - Run the following command in the 4th command prompt window: prompt> robocopy W:\ F:\ 2G.004 /NP /NS /COPY:D
9. Recording the performance number:
   - a)To warm up the testing, it is better drop the first result.

- b)To reduce the impact of the fluctuation among the tests, it's better test the same scenario more than 5 times and use the average number.
- c)Between tests, it is better to wait for more than 30 seconds so that the cached pages are flushed to the disk.

Understanding the result

You are expected to get the result something like the below Figure 3.7.4 shows:

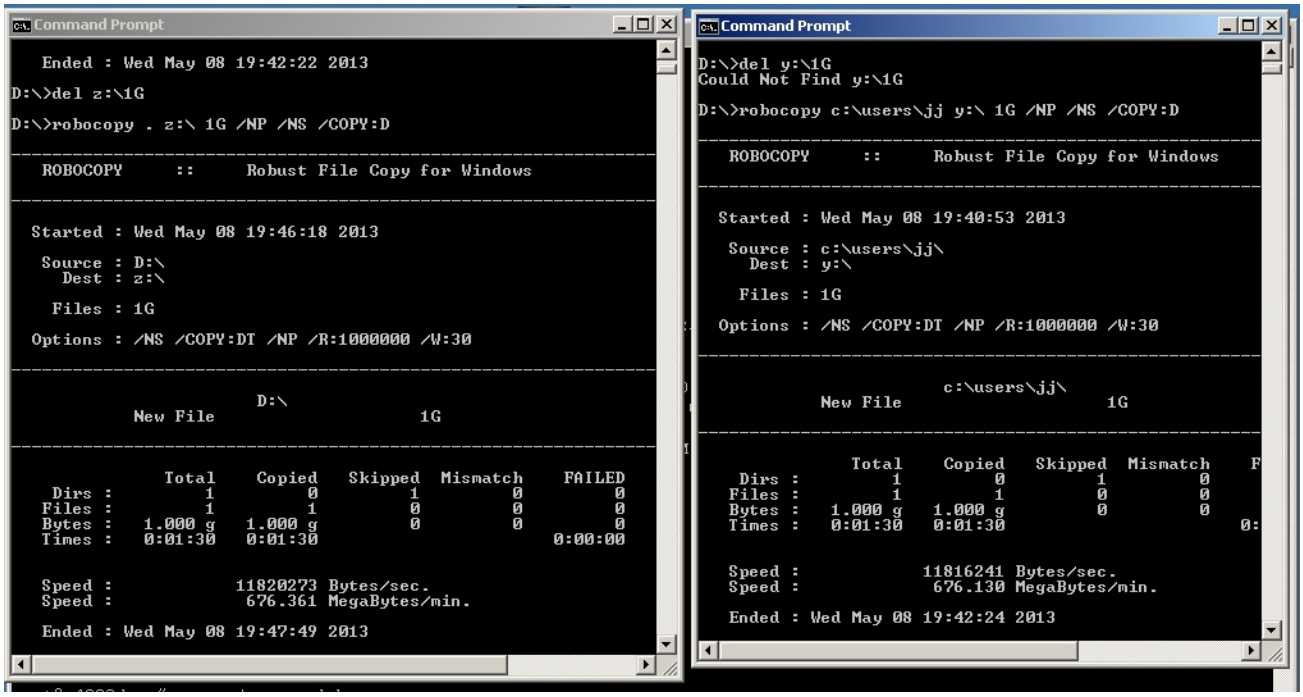


Figure 409. Figure 3.7.4

Here are the example robocopy results of reading 1G file in 2 command prompt windows of the NAS client. In fact, you should have 4 command prompt windows and 4 testing results.

The 1st NAS client result is (676.361 MegaBytes/min) / (60second/min) = 11MB/s

The 2nd NAS client result is (676.130 MegaBytes/min) / (60 second/min) =11 MB/s

The 3rd NAS client result is…

The 4th NAS client result is…

You should calculate the last result by adding 4 single client result:

The last result = $\sum_{i=1}^{4} result\ in\ client_i$ =result1 + result2 + result3 + result 4

12.2.8.4 Appendix

This section describes NAS PCD files, the FQID base table, client scripts, and the SAMBA configuration file.

12.2.8.4.1 NAS PCD files

PCD config file

```
<cfgdata>
<config>
<engine name="fm0">
<port type="1G" number="0" policy="fm1_gb0_NAS_policy"/>
<port type="1G" number="1" policy="fm1_gb1_NAS_policy"/>
<port type="1G" number="2" policy="fm1_gb2_NAS_policy"/>
<port type="1G" number="3" policy="fm1_gb3_NAS_policy"/>
<port type="1G" number="4" policy="fm1_gb4_NAS_policy"/>
<port type="10G" number="0" policy="fm1_10g_NAS_policy"/>
</engine>
<engine name="fm1">
<port type="1G" number="0" policy="fm2_gb0_NAS_policy"/>
<port type="1G" number="1" policy="fm2_gb1_NAS_policy"/>
<port type="1G" number="2" policy="fm2_gb2_NAS_policy"/>
<port type="1G" number="3" policy="fm2_gb3_NAS_policy"/>
<port type="1G" number="4" policy="fm2_gb4_NAS_policy"/>
<port type="10G" number="0" policy="fm2_10g_NAS_policy"/>
</engine>
</config>
</cfgdata>
<!-- end of PCD config file-->
```

NOTE

if you do not use some network interface, please make sure to delete those network interface in the PCD config files accordingly.

PCD policy file:

```
<?xml version="1.0" encoding="utf-8"?>
<netpcd xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="xmlProject/pcd.xsd" name="example" description="GRE Traffic Routing demo configuration">
<policy name="fm1_gb0_NAS_policy">
<dist_order>
<distributionref name="fm1_gb0_NAS_dist"/>
<distributionref name="fm1_gb0_default_dist"/>
</dist_order>
</policy>
<policy name="fm1_gb1_NAS_policy">
<dist_order>
<distributionref name="fm1_gb1_NAS_dist"/>
```

```
<distributionref name="fm1_gb1_default_dist"/>
</dist_order>
</policy>
<policy name="fm1_gb2_NAS_policy">
<dist_order>
<distributionref name="fm1_gb2_NAS_dist"/>
<distributionref name="fm1_gb2_default_dist"/>
</dist_order>
</policy>
<policy name="fm1_gb3_NAS_policy">
<dist_order>
<distributionref name="fm1_gb3_NAS_dist"/>
<distributionref name="fm1_gb3_default_dist"/>
</dist_order>
</policy>
<policy name="fm1_gb4_NAS_policy">
<dist_order>
<distributionref name="fm1_gb4_NAS_dist"/>
<distributionref name="fm1_gb4_default_dist"/>
</dist_order>
</policy>
<policy name="fm1_10g_NAS_policy">
<dist_order>
<distributionref name="fm1_10g_NAS_dist"/>
<distributionref name="fm1_10g_default_dist"/>
</dist_order>
</policy>
<policy name="fm2_gb0_NAS_policy">
<dist_order>
<distributionref name="fm2_gb0_NAS_dist"/>
<distributionref name="fm2_gb0_default_dist"/>
</dist_order>
</policy>
<policy name="fm2_gb1_NAS_policy">
<dist_order>
<distributionref name="fm2_gb1_NAS_dist"/>
<distributionref name="fm2_gb1_default_dist"/>
</dist_order>
```

```
</policy>
<policy name="fm2_gb2_NAS_policy">
<dist_order>
<distributionref name="fm2_gb2_NAS_dist"/>
<distributionref name="fm2_gb2_default_dist"/>
</dist_order>
</policy>
<policy name="fm2_gb3_NAS_policy">
<dist_order>
<distributionref name="fm2_gb3_NAS_dist"/>
<distributionref name="fm2_gb3_default_dist"/>
</dist_order>
</policy>
<policy name="fm2_gb4_NAS_policy">
<dist_order>
<distributionref name="fm2_gb4_NAS_dist"/>
<distributionref name="fm2_gb4_default_dist"/>
</dist_order>
</policy>
<policy name="fm2_10g_NAS_policy">
<dist_order>
<distributionref name="fm2_10g_NAS_dist"/>
<distributionref name="fm2_10g_default_dist"/>
</dist_order>
</policy>
<distribution name="fm1_gb0_NAS_dist">
<queue count="128" base="0x3800"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm1_gb0_default_dist">
<queue count="1" base="0x3800"/>
</distribution>
```

<distribution name="fm1_gb1_NAS_dist">
<queue count="128" base="0x3880"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm1_gb1_default_dist">
<queue count="1" base="0x3880"/>
</distribution>
<distribution name="fm1_gb2_NAS_dist">
<queue count="128" base="0x3900"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm1_gb2_default_dist">
<queue count="1" base="0x3900"/>
</distribution>
<distribution name="fm1_gb3_NAS_dist">
<queue count="128" base="0x3980"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm1_gb3_default_dist">
<queue count="1" base="0x3980"/>

```
</distribution>
<distribution name="fm1_gb4_NAS_dist">
<queue count="128" base="0x3a00"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm1_gb4_default_dist">
<queue count="1" base="0x3a00"/>
</distribution>
<distribution name="fm1_10g_NAS_dist">
<queue count="128" base="0x3c00"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm1_10g_default_dist">
<queue count="1" base="0x3c00"/>
</distribution>
<distribution name="fm2_gb0_NAS_dist">
<queue count="128" base="0x7800"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm2_gb0_default_dist">
```

```
<queue count="1" base="0x7800"/>
</distribution>
<distribution name="fm2_gb1_NAS_dist">
<queue count="128" base="0x7880"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm2_gb1_default_dist">
<queue count="1" base="0x7880"/>
</distribution>
<distribution name="fm2_gb2_NAS_dist">
<queue count="128" base="0x7900"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm2_gb2_default_dist">
<queue count="1" base="0x7900"/>
</distribution>
<distribution name="fm2_gb3_NAS_dist">
<queue count="128" base="0x7980"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
```

```
<distribution name="fm2_gb3_default_dist">
<queue count="1" base="0x7980"/>
</distribution>
<distribution name="fm2_gb4_NAS_dist">
<queue count="128" base="0x7a00"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm2_gb4_default_dist">
<queue count="1" base="0x7a00"/>
</distribution>
<distribution name="fm2_10g_NAS_dist">
<queue count="128" base="0x7c00"/>
<key>
<fieldref name="ipv4.src"/>
<fieldref name="ipv4.dst"/>
<fieldref name="ipv4.nextp"/>
<fieldref name="tcp.sport"/>
<fieldref name="tcp.dport"/>
</key>
</distribution>
<distribution name="fm2_10g_default_dist">
<queue count="1" base="0x7c00"/>
</distribution>
</netpcd>
```

12.2.8.4.2 FQID base table

FQID base calculation

This table is only a demo for FQID base calculation, please refer to the latest version of "*Core Affined Queues*" within the NXP linux_ethernet_driver documentation.

Table 611. FMAN v2 devices core affined queues

Interface	FQID base	P4080	P5020	P5040	P3041	P2041
fm1-gb0	0x3800	Y	Y	Y	Y	Y
fm1-gb1	0x3880	Y	Y	Y	Y	Y
fm1-gb2	0x3900	Y	Y	Y	Y	Y
fm1-gb3	0x3980	Y	Y	Y	Y	Y
fm1-gb4	0x3a00		Y	Y	Y	
fm1-10g	0x3c00	Y	Y	Y		
fm2-gb0	0x7800	Y		Y		
fm2-gb1	0x7880	Y		Y		
fm2-gb2	0x7900	Y		Y		
fm2-gb3	0x7980	Y		Y		
fm2-gb4	0x7a00			Y		
fm2-10g	0x7c00	Y		Y		

Table 612. FMAN v3 devices core affined queues

Interface	FQID base	T4240	T4160	B4860	B4420
fm1-mac1	0x3800	Y	Y	Y	Y
fm1-mac2	0x3880	Y	Y	Y	Y
fm1-mac3	0x3900	Y	Y	Y	Y
fm1-mac4	0x3980	Y	Y	Y	Y
fm1-mac5	0x3a00	Y	Y	Y	
fm1-mac6	0x3a80	Y	Y	Y	
fm1-mac9	0x3c00	Y	Y	Y	
fm1-mac10	0x3c80	Y		Y	
fm2-mac1	0x7800	Y	Y		
fm2-mac2	0x7880	Y	Y		
fm2-mac3	0x7900	Y	Y		

Table continues on the next page...

Table 612. FMAN v3 devices core affined queues (continued)

Interface	FQID base	T4240	T4160	B4860	B4420
fm2-mac4	0x7980	Y	Y		
fm2-mac5	0x7a00	Y	Y		
fm2-mac6	0x7a80	Y	Y		
fm2-mac9	0x7c00	Y	Y		
fm2-mac10	0x7c80	Y			

12.2.8.4.3 NAS client test scripts

NAS client test scripts for 1 client

This test script is only for 1 client and 4GByte files reading and writing benchmarking, for 2,4 or more client or other size files, the script should be modified accordingly.

Robocopy_4G_file.bat

```
echo ----------Write----------
del z:\4G
robocopy . z:\ 4G /NP /NS /COPY:D
del z:\4G
robocopy . z:\ 4G /NP /NS /COPY:D
del z:\4G
robocopy . z:\ 4G /NP /NS /COPY:D
del z:\4G
robocopy . z:\ 4G /NP /NS /COPY:D
del z:\4G
robocopy . z:\ 4G /NP /NS /COPY:D
del z:\4G
robocopy . z:\ 4G /NP /NS /COPY:D
del z:\4G
robocopy . z:\ 4G /NP /NS /COPY:D
echo ----------Read----------
del 4G
robocopy z:\ . 4G /NP /NS /COPY:D
del 4G
robocopy z:\ . 4G /NP /NS /COPY:D
del 4G
robocopy z:\ . 4G /NP /NS /COPY:D
```

```
del 4G
robocopy z:\ . 4G /NP /NS /COPY:D
del 4G
robocopy z:\ . 4G /NP /NS /COPY:D
del 4G
robocopy z:\ . 4G /NP /NS /COPY:D
del 4G
robocopy z:\ . 4G /NP /NS /COPY:D
pause
```

12.2.8.4.4 SAMBA config file

smb.conf

```
#make sure to enable requiring network interface in the [global] section and in the interfaces fields.
[global]
interfaces = fm1-gb0, fm1-gb2, fm1-gb3, fm1-gb4
map to guest = Bad User
client NTLMv2 auth = Yes
client lanman auth = No
client plaintext auth = No
max xmit = 2097152
socket options = TCP_NODELAY IPTOS_LOWDELAY SO_RCVBUF=262142
SO_SNDBUF=262142
dns proxy = No
#For 64bit kernel, please set use sendfile = No.
use sendfile = Yes
guest account = root
pre-reply write = No
min receivefile size = 1
dos charset = UTF-8
unix charset = UTF-8
display charset = UTF-8
[public]
path = /smbshare
read only = No
guest ok = Yes
```

12.2.8.5 Known Bugs, Limitations, or Technical Issues

1. NAS can't reading bigger than 2GB files while set "use sendfile = Yes" in smb.conf on DPAA platform 64bit.Workaround:modify smb.conf file, set "use sendfile = No".
2. Out of memory (OOM) situations:OOM sometimes occurs when storage systems and heavy network traffic are used at the same time. Linux kernel uses the Page Cache mechanism to cache in the main memory data ultimately intended for disk storage. Dedicated flusher threads will fetch the dirty cached pages from the main memory and, when the kernel permits, write them on the disk. This greatly increases the performance of the average systems that do not issue gigabyte transfers in short time bursts. In NAS case, network ports are used for network file transfer. Because of this, the flusher threads are not able to free the main memory fast enough to be used by other kernel applications, like Ethernet driver's softirq. In order to optimize the kernel to support write-heavy operations, some tweaks should be made to the kernel virtual machine to improve responsiveness and aggressiveness of the page cache manager. These optimizations are well documented in the kernel documentation pages (Documentation/sysctl/vm.txt). For NAS application, the parameter that forces the Linux LVM to keep a minimum number of kilobytes free in lowmem and the one that controls the aggressiveness of the kernel to reclaim memory used for caching were modified. "echo 10000 > /proc/sys/vm/min_free_kbytes echo 200 > /proc/sys/vm/vfs_cache_pressure" With above modifications, the OOM often disappears, the system achieves necessary balance between caching and memory availability. Although some of the parameters have a great impact on the system behavior, some of them are not scalable or may actually hurry the appearance of the OOM situation. For example, this can happen when setting the min_free_kbytes to a very high value. It should be also noted that the overall performance of the system during write-heavy operations greatly depends on the disk operation speed, which, as in most current architectures, is the system's bottleneck. Other systems and applications that do not behave satisfactorily with the kernel default values should use their own empirical values. https://www.kernel.org/doc/Documentation/sysctl/vm.txt http://www.westnet.com/~gsmith/content/linux-pdflush.htm
3. NAS (SAMBA) write performance sometimes are not stable enough even with the same hard environment, the same software environment and the same test steps according hundreds of observation. The best performance sometimes are +20% better than the worst performance. Many factors result in such a phenomenon, such as hard disks aging, hard disk cache size and other storage factor. In order to make the result more stable, more smooth, and easy reproduce, it is recommended to test the same case multiple times, more than 20 times is plus, then average those 20 times specimen as the last benchmarking result.

12.2.9 NAS - eTSEC

How to setup a NAS system on NXP QorIQ/LayerScape system, and how to measure its performance.

12.2.9.1 Benchmarking objectives

The purpose of this guide is to show how to setup a NAS system on a NXP QorIQ/LayerScape processor based system and how to measure its performance.

NOTE

LS1021ATWR was used for the demo. Please refer to the board's manual for other boards.

12.2.9.2 Test Environment

NAS Client

The suggested NAS client is described in table 1. The client has high throughput performance to make sure the testbed would not limit the NAS performance. Any degradation to the suggested NAS client may lead to degradation of overall NAS performance.

Table 613. NAS client configuration

Item	Description
CPU	Minimum requires Intel® i5 CPU 750@2.67 GHz
RAM	4G DDR3 RAM
Hard Drives	Western Digital® WD3000HLFS 10000 RPM,Two drives configured in RAID 0 array
Ethernet	Standalone Intel PRO/1000 PT Desktop Adapter (PCIE)
OS	Microsoft® Windows® 7 (32bit) or Microsoft Windows Vista®
Anti-virus software	off
Firewall software	off
Connection to NAS server	Direct connection through Ethernet cable without switches

NAS server

In this document, the NAS benchmark will be done on the LS1021ATWR. For details, please refer to user manual of the board.

The SATA disk being used is Western Digital WD3000HLFS, 10000RPM. If lower speed SATA is used, it will impact the overall NAS performance.

LS1021ATWR has only 1 onboard SATA connector, which is not enough to compose RAID5 array. Therefore, 2 ports PCIe to SATA adaptor will be used for RAID5 tests.

See Table 2 for details.

Table 614. NAS server configuration

Item	Description
Silicon and board	LS1021ATWR
RCW	RSR_PPS_70:rcw_1000.bin RSR_PPS_70 means: - RGMII@EC1 - SGMII@EC2 - RGMII@EC3 - PCIe*1 on slot 1 - PCIe*1 on slot 2 - SATA - SERDES1 Protocol is 0x70
RAM	1 G DDR3 RAM
Hard Drives	Western Digital WD3000HLFS 10000 RPM, Single drive, four drives configured in RAID 5 array or 2 drives configured in RAID0
Ethernet	Onboard VeTSEC
PCIe-SATA adaptor	Silicon Image 3132
MTU setting	MTU=1500, 9014
OS	Linux BSP, kernel version 4.1.8

Measurement

File copy under Windows7

The tests are done using the robocopy command built into Microsoft Windows 7 (and available as a download as part of the free Microsoft Windows Server® 2003 Resource Kit Tools).

The benchmarks consist of Write and Read transfers of a single file which size is 2 GB. The separate RAID 0 array in the NAS testbed is specifically designed to support this test.

12.2.9.3 Test Procedure

RAID5 NAS hardware setup

4 SATA disks are required for RAID5. You can compose RAID5 array on target QorIQ platform with a PCIe-SATA adapter and on-board SATA controllers. Please connect the board, PCIe-SATA adapter and SATA disks are as figure 1.

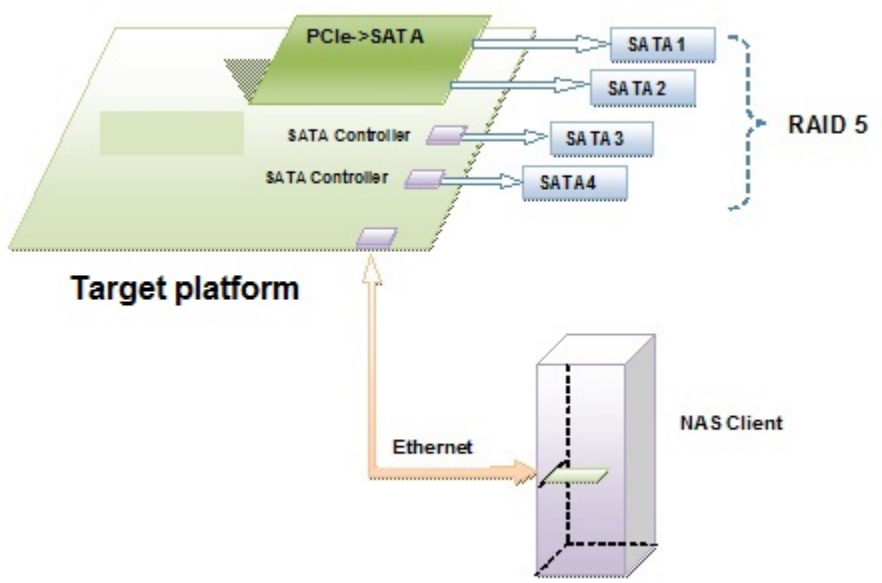


Figure 410. Connecting PCIe-SATA adapter and SATA disks

An alternative way: Compose RAID5 with 2 PCIe-SATA adapters on ls1021atwr, as shown in figure 2. This method is preferred since it is easy to reproduce.

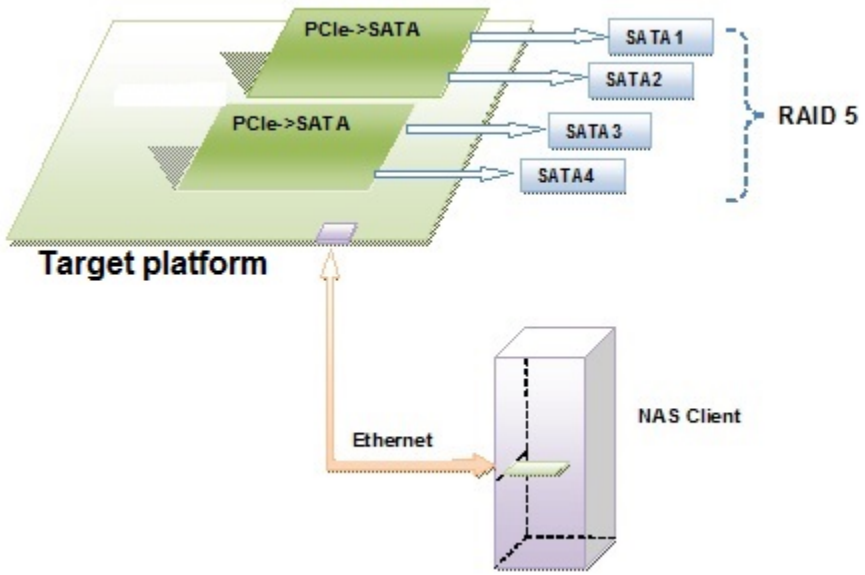


Figure 411. Using two PCIe-SATA adapters

Connect the board with NAS server via Ethernet, and now you have a NAS system.

RAID5 NAS test procedures

The default non-dpaa kernel configure does not support RAID5, In order to enabe RAID5 support and to make sure the best RAID5 local access performance, please refer to NXP Linux RAID reproducibility guide.

Please follow the below steps to have RAID5 NAS test:

1.Boot up the target QorIQ platform. Type “root” to enter the Linux command line.

2.Setup RAID5 on target QorIQ platform with below command. Suppose the shared folder is /smbshare, and EXT2 is used:

a) #mdadm -C /dev/md0 -amd -R -l5 -n4 /dev/sd[abcd]1

b) #mkfs.ext2 /dev/md0

c) #mount /dev/md0 /smbshare

d) #echo 512 > /sys/block/sda/bdi/read_ahead_kb

e) #echo 512 > /sys/block/sdb/bdi/read_ahead_kb

f) #echo 512 > /sys/block/sdc/bdi/read_ahead_kb

g) #echo 512 > /sys/block/sdd/bdi/read_ahead_kb

h) #echo 1024 >/sys/block/md0/md/stripe_cache_size

i) #ethtool -C eth[0,1] tx-frames 22 tx-usecs 32 (This step is needed only by mtu=1500)

j) #ethtool -C eth[0,1] rx-frames 22 rx-usecs 32 (This step is needed only by mtu=1500)

k) #ethtool -K eth[0,1] gro on gso on sg on

l) set CPU load automatically balance at ls1021atwr platform:

```
#cp /etc/samba/Heavy_Load_Assistant4LS1021A_NAS.sh ~/
#cd ~/
#chmod +x ./Heavy_Load_Assistant4LS1021A_NAS.sh
#./Heavy_Load_Assistant4LS1021A_NAS.sh auto &
```

```
Input below commands to restore default system setting at any time as will:
#killall -9 Heavy_Load_Assistant4LS1021A_NAS.sh
#cp /etc/samba/Heavy_Load_Assistant4LS1021A_NAS.sh ~/
#cd ~/
#chmod +x ./Heavy_Load_Assistant4LS1021A_NAS.sh
#./Heavy_Load_Assistant4LS1021A_NAS.sh restore
```

If you want to use EXT4 file system, change command line b) & c) to:

b') #mkfs.ext4 /dev/md0

c') #mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/md0 /smbshare

Startup Samba service on board side with command:

#smbd –s /etc/samba/smb.conf

On NAS client side, map the shared folder /smbshare as a windows drive, e.g, drive z. Please refer to the below steps a) to d) and figure 3, figure 4 for details.

a) Config NAS client and target QorIQ platform IP in the same network segment, e.g. 192.168.1.xxx

b) Connect the NAS client and the board directly by a wired network cable.

c) Enter board IP in the NAS client explorer window, such as \\192.168.1.100, then a folder named "public Share" will be shown.

d) Right click the folder "public Share", select "Map network drive…"in the popup menu. See Figure 3.

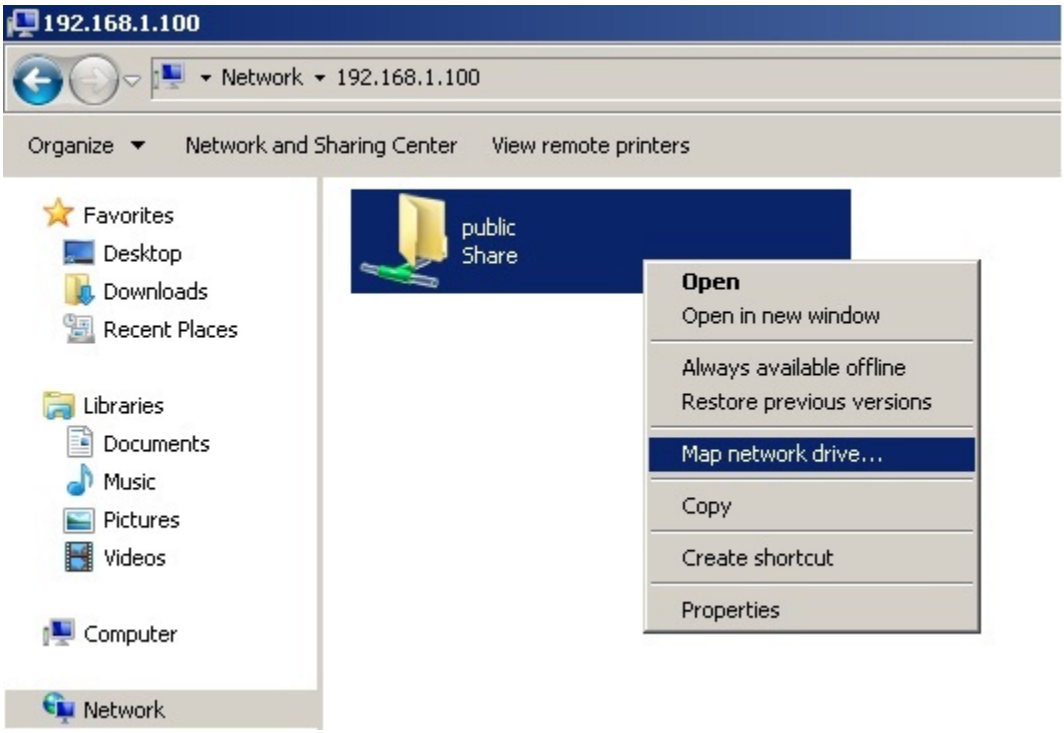


Figure 412. Selecting network driver

e) Click Finish, then the shared folder of NAS server is now mapped as a Windows drive Z. See Figure 4.

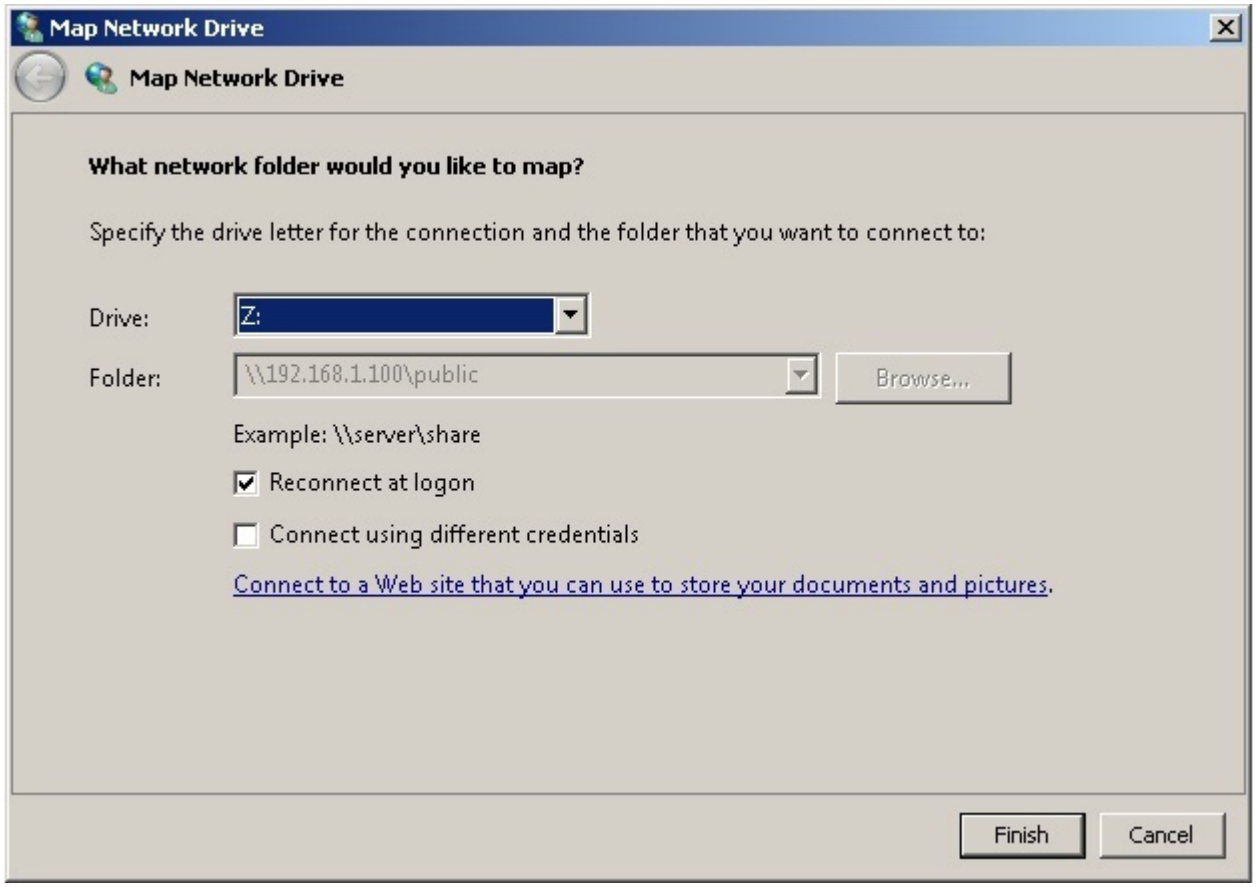


Figure 413. shared folder of NAS server mapped as Windows Drive Z

5.Setup up jumbo frame on both client and server side. With jumbo frame, better NAS performance can be achieved. Please skip this step and step 2.k) if you want to keep the default frame size, which is 1.5K.

a) At the NAS client (windows 7) side, open network and sharing center by selecting Start-> Settings-> Control Panel->System and Security->System->Device Manager->Intel(R) PRO/1000 PT Desktop Adapter and configure Jumbo Packets as 9014 bytes. Please refer to Figure 5.

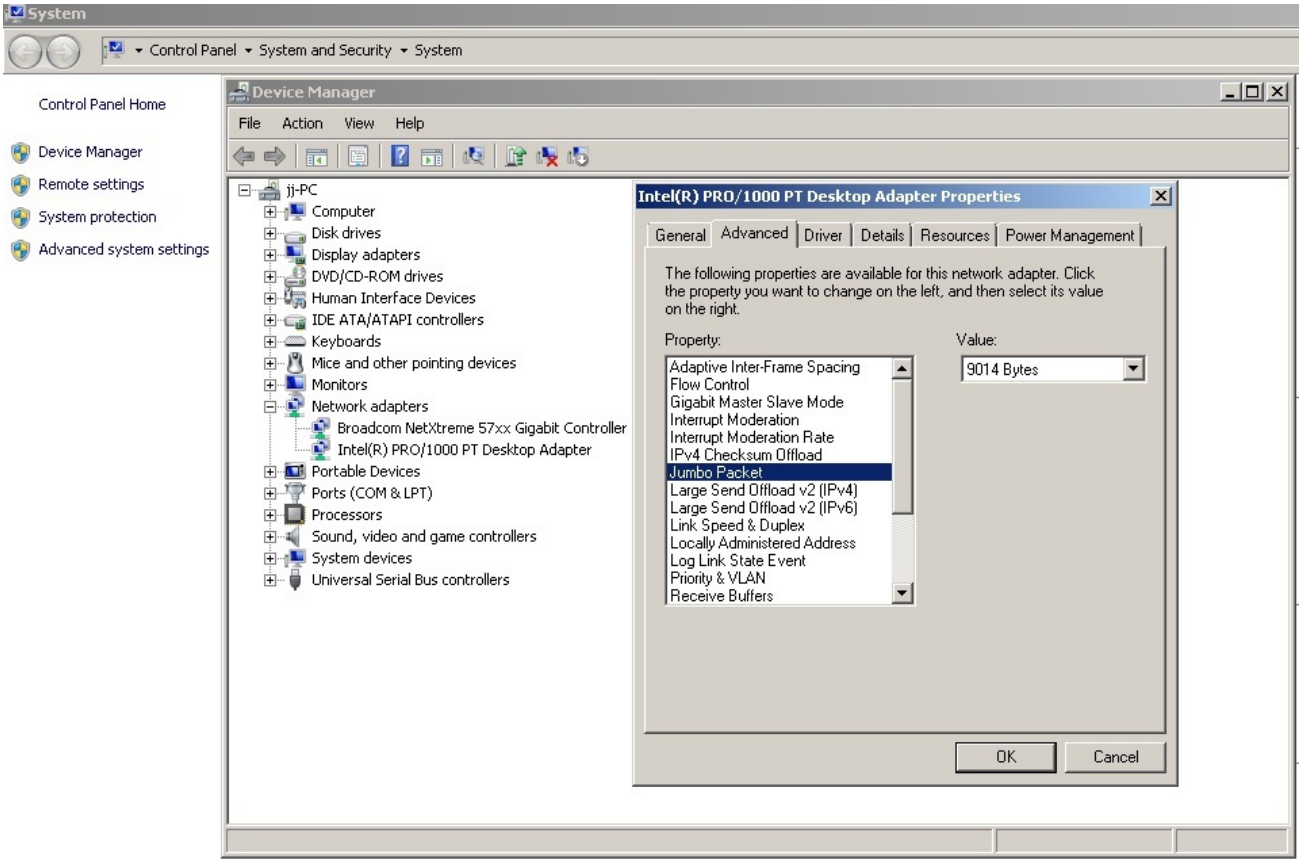


Figure 414. Jumbo Frame enablement

b) At the NAS server (Linux) side, config the ethernet MTU to 9014 with command:

ifconfig eth0 mtu 9014

6.Generate a dummy file with 2GB in size with the following command at NAS client side:

fsutil file createnew 2G.001 2147483648

7.Begin the test with below commands at NAS client side:

To test writing performance:

robocopy . z:\ 2G.001 /NP /NS /COPY:D

To test reading performance:

robocopy z:\ . 2G.001 /NP /NS /COPY:D

8.IMPRORTANT: Tips for getting the performance number:

a) To warm up the testing, it is better drop the first result.

b) To reduce the impact of the fluctuation among the tests, it's better test the same scenario more than 5 times and use the average number.

c) Between tests, it is better to wait for more than 30 seconds so that the cached pages are flushed to the disk.

Understanding the RAID5 NAS result

You are expected to get the result something like the below Figure 6 shows:

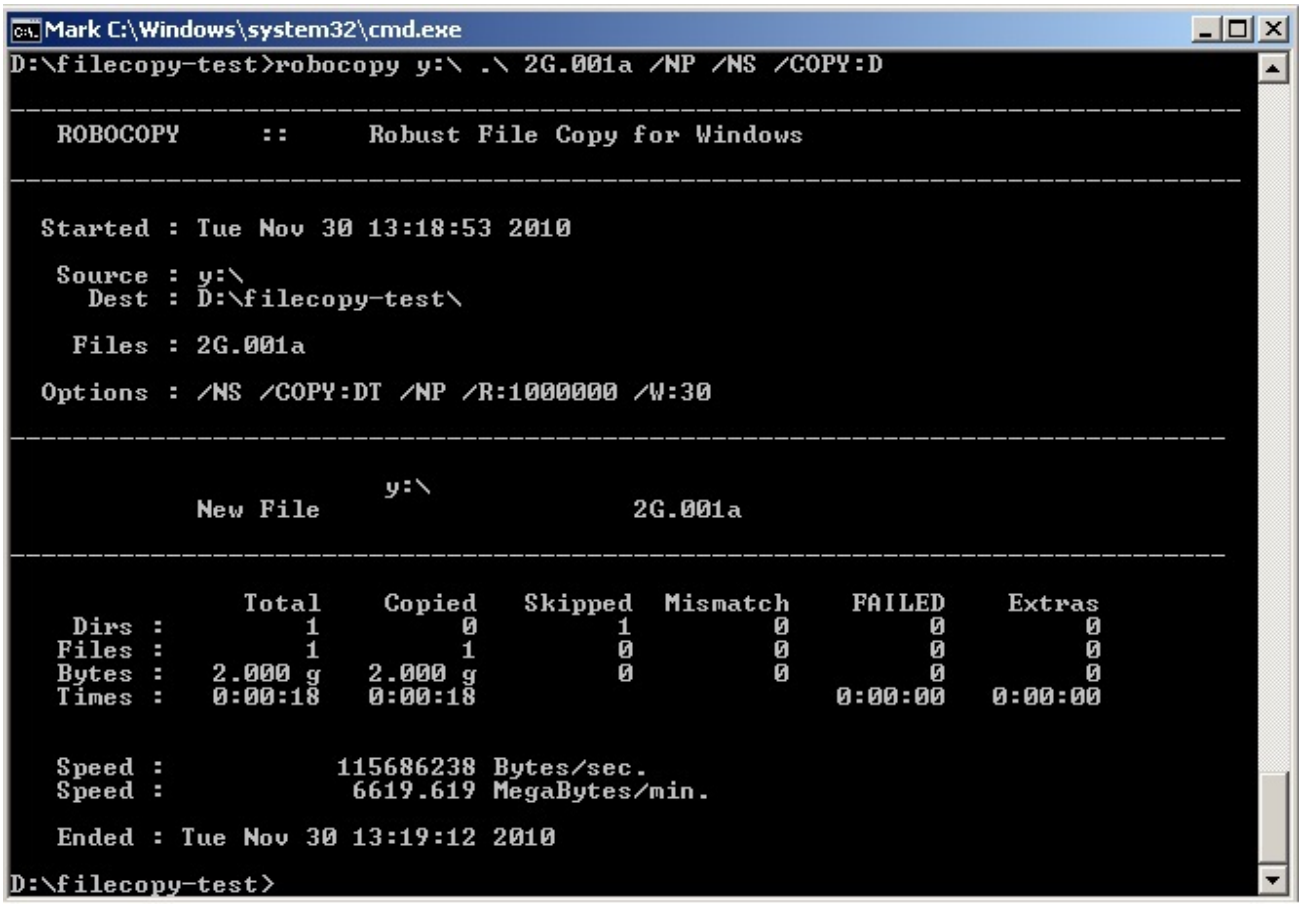


Figure 415. ROBOCOPY result

Here is the example robocopy result of reading 2G file. The number is 6619.619 MegaBytes/min. It means the reading speed is about 110MB/s that can be computed by 6619.619/60.

Single Disk NAS hardware setup

You can also connect the a single disk to on-board SATA controller (Figure 8) or to Silicon Image PCIe-SATA adapter (Figure 9) on ls1021atwr board, and benchmark the NAS performance of single disk.

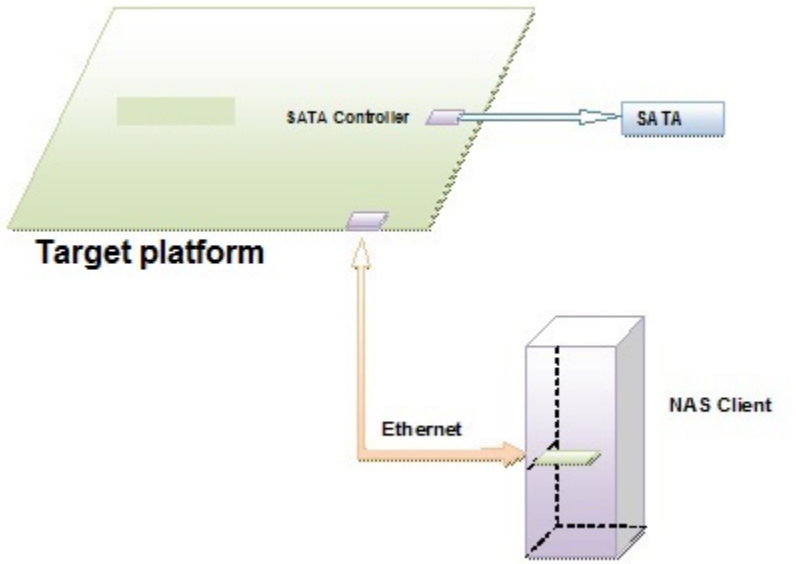


Figure 416. Soc Single SATA Disk NAS

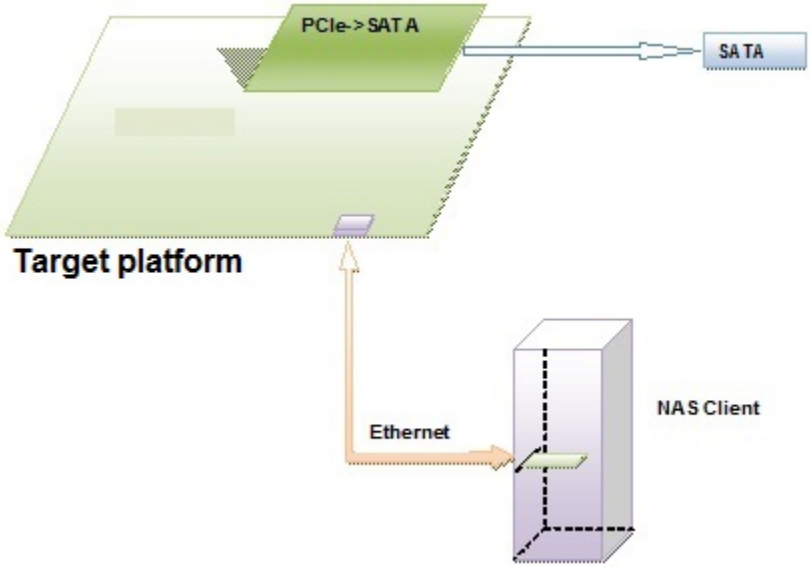


Figure 417. Single PCIe-SATA Disk NAS

Single disk NAS test procedures

Please follow below steps to benchmark single disk NAS performance:

1. Boot up the ls1021atwr board. Type “root” to enter the Linux command line.

2. Setup shared folder on ls1021atwr board with below commands. Suppose the shared folder is /smbshare:

a) #mkfs.ext2 /dev/sda1

b) #mount /dev/sda1 /smbshare

c) #echo 512 > /sys/block/sda/bdi/read_ahead_kb

d) #ethtool -C eth[0,1] tx-frames 22 tx-usecs 32 (This step is needed only by mtu=1500)

e) #ethtool -C eth[0,1] rx-frames 22 rx-usecs 32 (This step is needed only by mtu=1500)

f) #ethtool -K eth[0,1] gro on gso on sg on

g) set CPU load automatically balance at ls1021atwr platform:

```
#cp /etc/samba/Heavy_Load_Assistant4LS1021A_NAS.sh ~/
#cd ~/
#chmod +x ./Heavy_Load_Assistant4LS1021A_NAS.sh
#./Heavy_Load_Assistant4LS1021A_NAS.sh auto &
Input below commands to restore default system setting at any time as will:
#killall -9 Heavy_Load_Assistant4LS1021A_NAS.sh
#cp /etc/samba/Heavy_Load_Assistant4LS1021A_NAS.sh ~/
#cd ~/
#chmod +x ./Heavy_Load_Assistant4LS1021A_NAS.sh
#./Heavy_Load_Assistant4LS1021A_NAS.sh restore
```

If you want to use EXT4 file system, change above command a) b) to:

a') #mkfs.ext4 /dev/sda1

b') #mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/sda1 /smbshare

3. Remaining testing procedures are the same as that of RAID5. Please refer the step 3 to 7 of RAID5 NAS test procedures.

Understanding the single disk NAS result

Please refer to Understanding the RAID5 NAS result. The way of reading the result is the same as that of RAID5.

12.2.9.4 Known Bugs, Limitations, or Technical Issues

1. Since SDK1.4, the RAID5 NAS write performance is significantly lower than SDK1.3 release. This is primarily due to the lack of the "receive file optimization" used in previous releases. Patches are not suitable upstream, so they are not in SDK1.4 and later release.
2. Out of memory (OOM) situations:OOM sometimes occurs when storage systems and heavy network traffic are used at the same time. Linux kernel uses the Page Cache mechanism to cache in the main memory data ultimately intended for disk storage. Dedicated flusher threads will fetch the dirty cached pages from the main memory and, when the kernel permits, write them on the disk. This greatly increases the performance of the average systems that do not issue gigabyte transfers in short time bursts. In NAS case, network ports are used for network file transfer. Because of this, the flusher threads are not able to free the main memory fast enough to be used by other kernel applications, like Ethernet driver's softirq. In order to optimize the kernel to support write-heavy operations, some tweaks should be made to the kernel virtual machine to improve responsiveness and aggressiveness of the page cache manager. These optimizations are well documented in the kernel documentation pages (Documentation/sysctl/vm.txt). For NAS application, the parameter that forces the Linux LVM to keep a minimum number of kilobytes free in lowmem and the one that controls the aggressiveness of the kernel to reclaim memory used for caching were modified.

   ```
   #echo 10000 > /proc/sys/vm/min_free_kbytes
   #echo 200 > /proc/sys/vm/vfs_cache_pressure
   ```

   With above modifications, the OOM often disappears, the system achieves necessary balance between caching and memory availability. Although some of the parameters have a great impact on the system behavior, some of them are not scalable or may actually hurry the appearance of the OOM situation. For example, this can happen when setting the min_free_kbytes to a very high value. It should be also noted that the overall performance of the system during write-heavy operations greatly depends on the disk operation speed, which, as in most current architectures, is the

system's bottleneck. Other systems and applications that do not behave satisfactorily with the kernel default values should use their own empirical values. For details, please refer to:

```
https://www.kernel.org/doc/Documentation/sysctl/vm.txt
http://www.westnet.com/~gsmith/content/linux-pdflush.htm
```

3. NAS (SAMBA) write performance sometimes are not stable enough even with the same hard environment, the same software environment and the same test steps according hundreds of observation. The best performance sometimes are +20% better than the worst performance. Many factors result in such a phenomenon, such as hard disks aging, hard disk cache size and other storage factor. In order to make the result more stable, more smooth, and easy reproduce, it is recommended to test the same case multiple times, more than 20 times is plus, then average those 20 times specimen as the last benchmarking result.
4. In SDK1.8, NAS performance degrades about 20%. This is primarily due to the kernel version upgrade from 3.12.19-rt30 to 3.12.37-rt51. This is tracked by NXP internal CR:ENGR00356091.

12.2.10 Linux RAID

How to setup a Linux RAID system on a NXP QorIQ processor-based system, and how to measure the performance.

12.2.10.1 Benchmarking Objectives

The purpose of this guide is to show how to setup a Linux RAID system on a NXP QorIQ processor based system and how to measure its performance.

12.2.10.2 Test Environment

Hardware Platform Identification

- NXP reference board based on QorIQ series processors
- Hard disks: Western Digital WD3000HLFS 10000RPM, 4 drives configured in RAID5/6 array
- SATA-PCIe card: Silicon Image 3132 (2x speed) or Silicon Image 3124 (1x speed)

Software Platform Identification: Host development System Software

- QorIQ SDK 2.0 Release
- Compiler from QorIQ SDK 2.0

How to compile the kernel

If the distributed binary image in the ISO does not meet your requirement and you want to compile the kernel yourself, please make sure following kernel option is selected.

Table 615. Compiling kernel options

Kernel Configure Tree View Options	Description
[*] Enable the block layer ---> IO Schedulers ---> <*> Anticipatory I/O scheduler <*> Deadline I/O scheduler <*> CFQ I/O scheduler Default I/O scheduler (CFQ)	permit the block layer us to use correct I/O scheduler in the kernel

Table continues on the next page...

Table 615. Compiling kernel options (continued)

Kernel Configure Tree View Options	Description

Device Drivers --->
    <*> Serial ATA (prod) and Parallel ATA (experimental) drivers --->
     [*] Silicon Image 3124/3132 SATA support
     [*]   ATA SFF support
     <*>   Silicon Image SATA support
	Enable Silicon Image SATA driver

Device Drivers --->
    SCSI device support --->   [*] legacy /proc/scsi/ support
        --- SCSI support type (disk, tape, CD-ROM)
        <*> SCSI disk support
        <*> SCSI tape support
        <*> SCSI CDROM support
        [*]   Enable vendor-specific extensions (for SCSI CDROM)
        <*> SCSI generic support
   SCSI Transports --->   <*> Parallel SCSI (SPI) Transport Attributes
        [*] SCSI low-level drivers --->
	Enable SCSI disk, tape and CD-ROM support

[*] Multiple devices driver support (RAID and LVM) ---> --- Multiple devices driver support (RAID and LVM) <*> RAID support <*> Linear (append) mode <*> RAID-0 (striping) mode <*> RAID-1 (mirroring) mode <*> RAID-4/RAID-5/RAID-6 mode	Enable RAID support
Device Drivers ---> [*] DMA Engine support ---> [*] Freescale Elo and Elo Plus DMA support [*] Async_tx: Offload support for the async_tx api [*] Cryptographic API ---> --- Cryptographic API [*] Hardware crypto devices ---> --- Hardware crypto devices **For SEC3.x** <*> Talitos Freescale Security Engine (SEC) **For P5020/P5040** Device Drivers ---> [*] DMA Engine support ---> [*] Freescale RAID Engine Device Driver [*] Async_tx: Offload support for the async_tx api	**Important Note: As you know, different processors are equipped with different offloading capability. When you want to use the offload engine to offload the XOR calculation, you should know which hardware component is doing the offloading work and enable the corresponding kernel option. The guide lies below: SEC3.x: enable Talitos. P5020/P5040: enable RAID engine.**

12.2.10.3 Test Procedure

Hardware setup

Four SATA disks will be needed for a RAID system. You can compose a RAID array with two PCIe-SATA adaptors. Please connect the board, PCIe-SATA adaptor, and SATA disks as shown in figure 1. Here is an example.

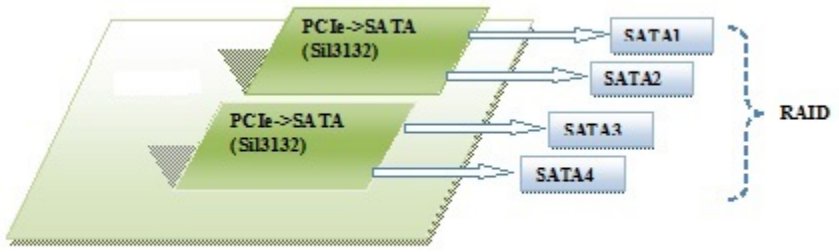


Figure 418. SATA connection for a RAID array

RAID setup

1. Create partitions for RAID

```
# fdisk /dev/sda
```

The number of cylinders for this disk is set to 19457. There is nothing wrong with that, but this is larger than 1024, and could in certain setups cause problems with:

- software that runs at boot time (e.g., old versions of LILO)
- booting and partitioning software from other OSs (e.g., DOS FDISK, OS/2 FDISK)

```
Command (m for help):___
   -  Enter "n"
Command action
   e   extended
   p   primary partition (1-4)
   -  Enter "p"
First cylinder (1-19457, 1): "1"
Last cylinder or +size or +sizeM or +sizeK (default 19457): "+4000M"
Change the Type of partition to Linux raid autodetect.
Command (m for help):___
   Enter "t"
Partition number (1-4): "1"
Hex code (type L to list codes):"fd"
Command (m for help): "w"
The partition table has been altered!
With the same procedures make other three partitions sdb1, sdc1 and sdd1
After creating all four partitions Log should be,
# fdisk -l
Disk /dev/sda: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
   Device Boot    Start       End    Blocks   Id  System
/dev/sda1               1         487     3911796   fd  Linux raid autodetect
Disk /dev/sdb: 160.0 GB, 160041885696 bytes
```

```
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
   Device Boot    Start       End    Blocks   Id  System
/dev/sdb1               1         487     3911796   fd  Linux raid autodetect
Disk /dev/sdc: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
   Device Boot    Start       End    Blocks   Id  System
/dev/sdc1               1         487     3911796   fd  Linux raid autodetect
Disk /dev/sdd: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
   Device Boot    Start       End    Blocks   Id  System
/dev/sdd1               1         487     3911796   fd  Linux raid autodetect
```

2. Create RAID array with Linux raid partition

```
# mdadm --create --verbose /dev/md0 --level=raid5 --raid-devices=4 /dev/sda1 /dev/sdb1 /dev/
sdc1 /dev/sdd1
md: bind<sda1>
md: bind<sdb1>
md: bind<sdc1>
md: bind<sdd1>
raid5: device sdc1 operational as raid disk 2
raid5: device sdb1 operational as raid disk 1
raid5: device sda1 operational as raid disk 0
raid5: allocated 4203kB for md0
raid5: raid level 5 set md0 active with 3 out of 4 devices, algorithm 2
RAID5 conf printout:
 --- rd:4 wd:3
 disk 0, o:1, dev:sda1
 disk 1, o:1, dev:sdb1
 disk 2, o:1, dev:sdc1
RAID5 conf printout:
 --- rd:4 wd:3
 disk 0, o:1, dev:sda1
 disk 1, o:1, dev:sdb1
 disk 2, o:1, dev:sdc1
 disk 3, o:1, dev:sdd1
mdadm: array /demd: recovery of RAID array md0
md: minimum _guaranteed_  speed: 1000 KB/sec/disk.
md: using maximum available idle IO bandwidth (but not more than 200000 KB/sec) for recovery.
md: using 128k window, over a total of 3911680 blocks.
dev/md0 started.

If want to create RAID6, please specify –level=raid6

Progress of the RAID array can be seen by,
# cat /proc/mdstat
Personalities : [raid6] [raid5] [raid4]
md0 : active raid5 sdd1[4] sdc1[2] sdb1[1] sda1[0]
      11735040 blocks level 5, 64k chunk, algorithm 2 [4/3] [UUU_]
      [==========>.........]  recovery = 52.7% (2065152/3911680) finish=0.9min speed=32080K/sec
Once recovery is completed RAID array is ready for data transfer.
```

3. Create File system on the RAID array and mount it

```
# mkfs.ext4 /dev/md0
# mount -o oldalloc,data=writeback,barrier=0,delalloc /dev/md0 /mnt
```

4. Tuning disk readahead and RAID stripe cache size

```
# echo 512 > /sys/block/sda/bdi/read_ahead_kb
# echo 512 > /sys/block/sdb/bdi/read_ahead_kb
# echo 512 > /sys/block/sdc/bdi/read_ahead_kb
# echo 512 > /sys/block/sdd/bdi/read_ahead_kb
# echo 1024 >/sys/block/md0/md/stripe_cache_size
```

Benchmarking and result explanation

1. Benchmarking RAID performance with iozone

```
# iozone -Rab result_file -i0 -i1 -n32M -g4G -r64 -f /mnt/test
```

Where,

- i stands for the kind of I/O to be run and values 0 and 1 means write and read respectively
- n and g represents the file sizes for the I/O
- r is the record size which will be used to the I/O's throughput
- f means the file on which I/O's will be performed

Here 4G is an example; it should be 2X RAM size.

2. Reading result

After iozone command is completed without error, there should be output such as the following table:

```
          32768      64  422159  703085   764872   766146

          65536      64  421909  703682   780245   781969

         131072      64  327741  432575   781069   781763

         262144      64  213168  254712   771225   773137

         524288      64  191111  201107   772176   774159

        1048576      64  170138  180182   771197   772426

        2097152      64  164549  160689   321871   324975

        4194304      64  156162  154649   307762   309286
```

This table shows write/rewrite/read/reread performance of different file sizes, from 32M to 4G. As the red text in the table, the write performance result is 152MB/s=156162/1024, read performance result is 300MB/s.

12.3 USDPAA

USDPAA - Benchmark Reproducibility Guides

12.3.1 Reflector

Details for reproducing USDPAA Refelctor performance results.

12.3.1.1 Introduction

This document describes the steps to verify performance of USDPAA Reflector.

12.3.1.2 Platform Identification

Hardware Platform Identification

Depending on the platform, the application uses the setup shown in the table below.

Table 616. System Setup

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
B4860QDS	Rev B2	Rev 2.2	• CPU - 1600 MHz • CCB - 667 MHz • DDR - 933 MHz • FMan - 667 MHz	`0x2A_0x8D`	`N_RSSS_0x2A_ 0x8D`
P2041RDB	Rev B	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 667 MHz • FMan - 583 MHz	`0x09`	`RR_PX_0x09`
P3041DS	Rev X1	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 667 MHz • FMan - 583 MHz	`0x36`	`RR_HXAPNSP_0 x36`

Table continues on the next page...

Table 616. System Setup (continued)

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
P4080DS	Rev X3	Rev 3.0	• CPU - 1500 MHz • CCB - 800 MHz • DDR - 650 MHz • FMan - 600 MHz	0xe	R_PPSXX_0xe
P5020DS	Rev X7	Rev 2.0	• CPU - 2000 MHz • CCB - 800 MHz • DDR - 667 MHz • FMan - 600 MHz	0x36	RR_HXAPNSP_0x36
P5040DS	Rev X4	Rev 2.1	• CPU - 2266 MHz • CCB - 800 MHz • DDR - 800 MHz • FMan - 600 MHz	0x02	RR_XXSNSpP_0x02
T1023RDB	Rev B	Rev 1.0	• CPU - 1400 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 700 MHz	0x77	RNSS_NPP_77
T1024RDB	Rev A	Rev 1.0	• CPU - 1400 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 700 MHz	0x95	RRX_PP_95

Table continues on the next page...

Table 616. System Setup (continued)

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
T1040RDB	Rev A	Rev 1.1	• CPU - 1400 MHz • CCB - 600 MHz • DDR - 800 MHz • FMan - 600 MHz	`0x66`	`RR_P_66`
T2080QDS	Rev X5	Rev 1.1	• CPU - 1800 MHz • CCB - 600 MHz • DDR - 933 MHz • FMan - 700 MHz	`6c_2d`	`RR_PNSNNR_6C_2D`
T4240QDS	Rev X4	Rev 2.0	• CPU - 1800 MHz • CCB - 733 MHz • DDR - 933 MHz • FMan - 733 MHz	`1_1_5_5`	`RR_XXXXPRPR_1_1_5_5`
T4240RDB	Rev A	Rev 2.0	• CPU - 1800 MHz • CCB - 733 MHz • DDR - 933 MHz • FMan - 733 MHz	`27_55_1_9`	`SSFFPPH_27_55_1_9`
LS1043ARDB	Rev A	Rev 1.1	• CPU - 1600 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 500 MHz	`1455`	`RR_FQPP_1455`

Table continues on the next page...

Table 616. System Setup (continued)

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
LS1046ARDB	Rev A	Rev 1.0	• CPU - 1800 MHz • CCB - 700 MHz • DDR - 1050 MHz • FMan - 800 MHz	1133_5559	RR_FFSSPPPH_ 1133_5559

Network Simulator: Spirent TestCenter Performance Analysis System

The simulator version used was Chassis ver. 4.24.

NOTE

This particular benchmarking process used Spirent TestCenter, although other traffic generators may also be used.

Software Platform Identification

Target Development System Software:

The table below shows the different file names needed for each target development system. Files enumerated are platform specific. The rx.x string in the file names indicate the silicon revision number.

Table 617. Software Platform Identification

Target Development System Software	Image file name/Description
FMan ucode	fsl_fman_ucode_<soc>_<silicon_rev>_<ucode_version>.bin Please refer to SDK document to use correct ucode version

Table continues on the next page...

Table 617. Software Platform Identification (continued)

Target Development System Software	Image file name/Description
RCW	P2041: rcw/RR_PX_0x09/rcw_14g_1500mhz.bin P3041: rcw/RR_HXAPNSP_0x36/rcw_15g_1500mhz.bin P4080: rcw/R_PPSXX_0xe/rcw_2sgmii_1500mhz_rev3.bin P5020: rcw/RR_HXAPNSP_0x36/rcw_15g_2000mhz.bin P5040: rcw/RR_XXSNSpP_0x02/rcw_26g_2267mhz.bin B4860: rcw/N_RSSS_0x2A_0x8D/ rcw_4sgmii_4srio_2xfi_1600mhz_1866ddr_rev2.bin T4240QDS: rcw/RR_XXXXPRPR_1_1_5_5/rcw_1_1_5_5_1666MHz_rev2.bin T4240RDB: rcw/SSFFPPH_27_55_1_9/rcw_27_55_1_9_1666MHz.bin T2080QDS: rcw/RR_PNSNNR_6C_2D/rcw_6c_2d_1800MHz.bin T1040RDB: rcw/RR_P_66/rcw_1400MHz.bin T1023RDB: rcw/RNSS_NPP_77/rcw_77_1400MHz.bin T1024RDB: rcw/RRX_PPP_95/rcw_95_1400MHz.bin LS1043ARDB: rcw/RR_FQPP_1455/rcw_1600.bin LS1046ARDB: rcw/ls1046ardb/RR_FFSSPPPH_1133_5559/ rcw_1800_qspiboot.bin
Boot loader	u-boot-${PLATFORM}.bin
Extra bootargs*	bportals=s0 qportals=s0
Linux Kernel (For PowerPC)	uImage-${platform}.bin
Device Tree (For PowerPC)	uImage-${platform}-usdpaa.dtb
File system (For PowerPC)	fsl-image-core-${platform}.ext2.gz.u-boot
FIT (For ARM)	kernel-fsl-${platform}-usdpaa.itb

12.3.1.3 Applications and Traffic Flow

USDPAA Reflector Applications

The USDPAA package provides 2 reflector applications:

- reflector
- hello_reflector

The table below provides a brief description for each reflector application.

Table 618. USDPAA Reflector Applications

Applications	Description
USDPAA PPAC reflector application	USDPAA reflector is implemented based on PPAC software framework. Please refer to USDPAA Reflector and PPAC User Guide for details.
USDPAA hello_reflector Application	USDPAA hello_reflector is a stand-alone application. It is a simplified implementation of reflector.

Reflector traffic flow

The reflector application swaps the source and destination MAC addresses of the received packet, swaps the IPv4 source and destination addresses of the received packet, and then sends the packet back out the same interface on which the packet was received.

On P2041RBD/P3041DS/P5020DS/LS1046ARDB, 14G reflecting is supported

On P4080DS, 22G reflecting is supported

On P5040DS, 26G reflecting is supported

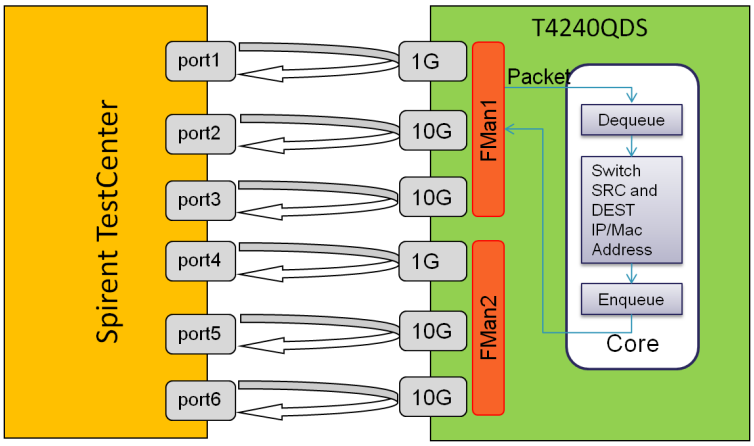
On B4860QDS, 20G reflecting is supported

On T4240QDS, 42G reflecting is supported

On T2080QDS, 24G reflecting is supported

On T1040RDB, 8G reflecting is supported

On LS1043ARDB, 6G reflecting is supported



DPAA Resource Configuration for Reflector

The configuration for DPAA blocks is critical to system performance. The Reflector application works with the following configuration:

Table 619. DPAA Resource Configuration for USDPAA Reflector

DPAA Resource	USDPAA IPv4 Forward Function

FMan	• Parsing on 32 bytes parser: result comes with every frame.
	  • Protocol parsing
	  • IP Header location
	  • IP Header checksum check
	  • Error code
	• Congestion management disabled. Fman side order restoration is on.
	• Hash is based on the SIP,DIP

QMan	• Push mode, Triple dequeue enable.
	• DQRR:
	  • Entry Stashing on.
	  • To L1 cache, with high priority.
	• Frame data, Annotation, Context stashing on: –Frame data = 1 line, Annotation = 1 line, Context = 1
	• Consumer Index write mode cache-inhibited: Update on every 3 frames dequeue.
	• EQRR: Access through Cache Enable Access (CENA, WIMGE=00000)

BMan	• Buffer Pool: bp9 • Buffer Size: 1728 Bytes • Number of buffers: 8192
DMA Memory	• 256MB for 32 bit platform • 256MB for 64 bit platform

12.3.1.4 Test Procedure

Steps for PPAC reflector application

The following instructions are specifically for the Spirent TestCenter, though other traffic generators can also be used.

Step 1: Configure FMan PCD:

- Login to the target
- Configure FMan with fmc:

```
# cd /usr/etc
# fmc -c <fmc_config> -p usdpaa_policy_hash_ipv4.xml –a
```

Please note that <fmc_config> can be one of the following:

Table 620. fmc Configuration

Platform	fmc_config file
B4860QDS	usdpaa_config_b4_serdes_0x2a_0x98.xml
P2041RDB/P3041DS/P5020DS	usdpaa_config_p2_p3_p5_14g.xml
P4080DS	usdpaa_config_p4_serdes_0xe.xml
P5040DS	usdpaa_config_p5_serdes_0x02.xml
T4240QDS	usdpaa_config_t4_serdes_1_1_5_5.xml
T4240RDB	usdpaa_config_t4_48g.xml
T2080QDS	usdpaa_config_t2_serdes_66_15.xml
T1024RDB	usdpaa_config_t1024_serdes_0x99.xml
T1040RDB	usdpaa_config_t1_serdes_0x66.xml
LS1043ARDB	usdpaa_config_ls1043.xml
LS1046ARDB	usdpaa_config_ls1046.xml

Step 2 (For T1040RDB only): Configure the L2switch FDB entries:

```
# killall l2sw_bin
# l2sw_bin
l2switch> mac add 00:11:22:33:44:01 0                  (Add mac address to FDB of port 0)
l2switch> mac add 00:11:22:33:55:01 1
l2switch> mac add 00:11:22:33:44:02 2
l2switch> mac add 00:11:22:33:55:02 3
l2switch> mac add 00:11:22:33:44:03 4
l2switch> mac add 00:11:22:33:55:03 4
l2switch> mac add <mac addr of fm1-gb0> 8
l2switch> mac add <mac addr of fm1-gb1> 9
l2switch> mac dump

Type VID MAC Address Ports
------ --- ----------------- -----
Static  1    00:11:22:33:44:01  0
Static  1    00:11:22:33:44:02  2
Static  1    00:11:22:33:44:03  4
Static  1    00:11:22:33:55:01  1
Static  1    00:11:22:33:55:02  3
Static  1    00:11:22:33:55:03  4
Static  1    00:e0:0c:00:31:00  8
Static  1    00:e0:0c:00:31:01  9

Static  entries: 8
Dynamic entries: 0
l2switch> Ctrl+C                                    (To terminate process without cleaning up
L2switch setting)
```

Reflector on T1040RDB is a special case because it involves L2 switch which has eight 1G front ports and is internally connected with DPAA through two 2.5G ports (fm1-gb0 & fm1-gb1). In order to achieve maximum throughput of DPAA, we have to send traffic to fm1-gb0/1 through port 0-4 of L2switch.

The Destination Mac address of packets that are sent to fm1-gb0 should be Mac address of fm1-gb0; the Source Mac address that are sent out from port 0 of L2switch should be 00:11:22:33:44:01 which we have already added into FDB of port 0 in step 3.

Step 3: Launch reflector application

```
# reflector <cpu_range> -c <fmc_config> -p usdpaa_policy_hash_ipv4.xml -d 0x10000000
```

NOTE

<cpu_range> could be a single core number, like 1, 2, or a range, like 1..8, 0..23 etc. <fmc_config> should be the same file used in Step 2.

On T2080QDS, user could get better performance with 8 cores if user adds parameter '-s' for reflector.

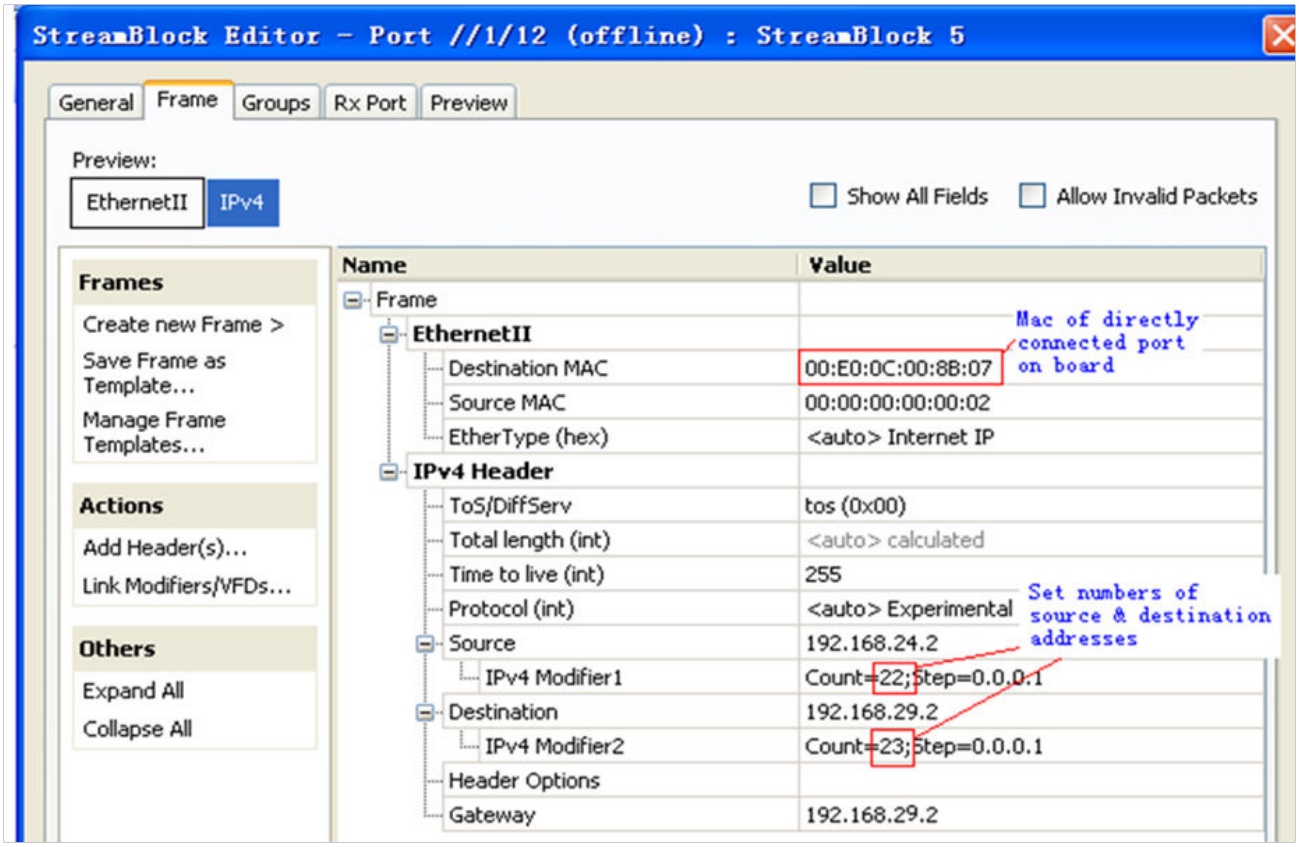
e.g. reflector 0..7 –c usdpaa_config_t2_serdes_66_15.xml –p usdpaa_policy_hash_ipv4.xml –d 0x10000000 -s

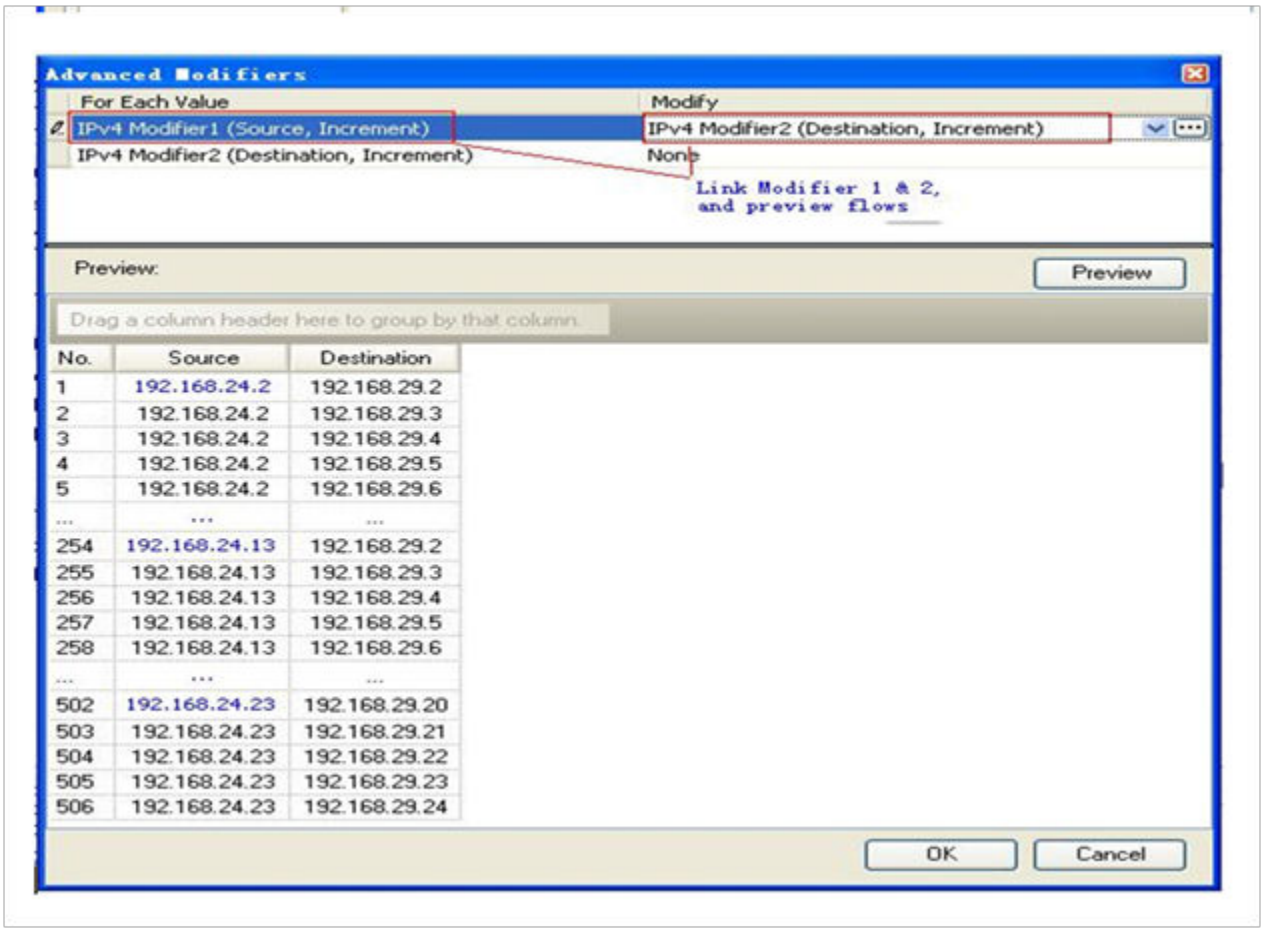
Step 4: Configure Spirent TestCenter project.

Create Command Sequencer in TestCenter and select RFC2544 Throughput test. Then set Test Parameters for TestCenter Throughput Sequencer as shown in table below.

Table 621. Spirent TestCenter Throughput Configuration

Spirent variable	Variable Setting
Test Duration	60 seconds
Resolution	0.1%
Initial rate	100%
Packet Loss Rate	0.001%
Packet Size	64, 390, 1024, 1518 bytes


StreamBlock Editor - Port //1/12 (offline) : StreamBlock 5
General
Frame
Groups
Rx Port
Preview
Preview:
EthernetII
IPv4
Show All Fields
Allow Invalid Packets
Frames
Create new Frame >
Save Frame as Template...
Manage Frame Templates...
Actions
Add Header(s)...
Link Modifiers/VFDs...
Others
Expand All
Collapse All
Name
Value
Frame
EthernetII
Destination MAC
00:E0:0C:00:8B:07
Mac of directly connected port on board
Source MAC
00:00:00:00:00:02
EtherType (hex)
<auto> Internet IP
IPv4 Header
ToS/DiffServ
tos (0x00)
Total length (int)
<auto> calculated
Time to live (int)
255
Protocol (int)
<auto> Experimental
Source
192.168.24.2
IPv4 Modifier1
Count=22;Step=0.0.0.1
Destination
192.168.29.2
IPv4 Modifier2
Count=23;Step=0.0.0.1
Set numbers of source & destination addresses
Header Options
Gateway
192.168.29.2


Advanced Modifiers
For Each Value
Modify
IPv4 Modifier1 (Source, Increment)
IPv4 Modifier2 (Destination, Increment)
IPv4 Modifier2 (Destination, Increment)
None
Link Modifier 1 & 2, and preview flows
Preview:
Preview
Drag a column header here to group by that column
No.
Source
Destination
1 192.168.24.2 192.168.29.2
2 192.168.24.2 192.168.29.3
3 192.168.24.2 192.168.29.4
4 192.168.24.2 192.168.29.5
5 192.168.24.2 192.168.29.6
254 192.168.24.13 192.168.29.2
255 192.168.24.13 192.168.29.3
256 192.168.24.13 192.168.29.4
257 192.168.24.13 192.168.29.5
258 192.168.24.13 192.168.29.6
502 192.168.24.23 192.168.29.20
503 192.168.24.23 192.168.29.21
504 192.168.24.23 192.168.29.22
505 192.168.24.23 192.168.29.23
506 192.168.24.23 192.168.29.24
OK
Cancel

Step 5: Now the PPAC reflector is ready to receive packet, Launch TestCenter project to start traffic

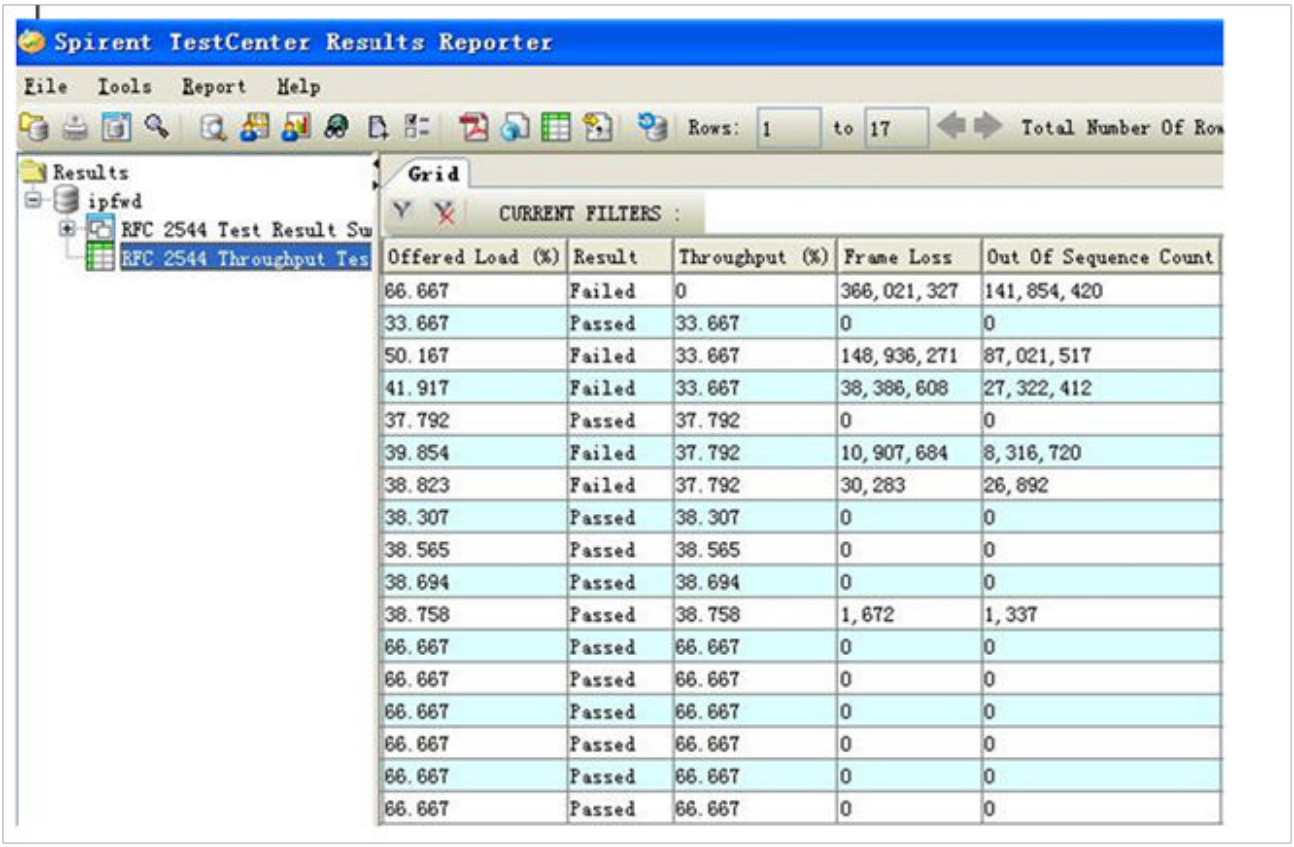
Steps for hello_reflector application

The steps for the hello_reflector application are the same as in PPAC reflector except for **Step 3**:

```
# hello_reflector -c <fmc_config> -p usdpaa_policy_hash_ipv4.xml -n <number of threads>
```

12.3.1.5 View of Test Results

When measuring performance using TestCenter RFC 2544 Command Sequence, the result summary can be viewed using the Spirent TestCenter Results Reporter. To prevent any "out of sequence" packets when measuring with Zero Loss Throughput, enable Order Preservation or Order Restoration.



Offered Load (%)	Result	Throughput (%)	Frame Loss	Out Of Sequence Count
66.667	Failed	0	366,021,327	141,854,420
33.667	Passed	33.667	0	0
50.167	Failed	33.667	148,936,271	87,021,517
41.917	Failed	33.667	38,386,608	27,322,412
37.792	Passed	37.792	0	0
39.854	Failed	37.792	10,907,684	8,316,720
38.823	Failed	37.792	30,283	26,892
38.307	Passed	38.307	0	0
38.565	Passed	38.565	0	0
38.694	Passed	38.694	0	0
38.758	Passed	38.758	1,672	1,337
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0

12.3.2 USDPAA IPv4 Forwarding

Details for reproducing USDPAA IPv4 forwarding performance results.

12.3.2.1 Introduction

This document describes the steps to verify performance of USDPAA IPv4 packet forwarding.

12.3.2.2 Platform Identification

Hardware Platform Identification

Depending on the platform, the application uses the setup shown in the table below.

Table 622. System Setup

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
B4860QDS	Rev B2	Rev 2.2	• CPU - 1600 MHz • CCB - 667 MHz • DDR - 933 MHz • FMan - 667 MHz	`0x2A_0x8D`	`N_RSSS_0x2A_ 0x8D`
P2041RDB	Rev B	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 667 MHz • FMan - 583 MHz	`0x09`	`RR_PX_0x09`
P3041DS	Rev X1	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 667 MHz • FMan - 583 MHz	`0x36`	`RR_HXAPNSP_0 x36`
P4080DS	Rev X3	Rev 3.0	• CPU - 1500 MHz • CCB - 800 MHz • DDR - 650 MHz • FMan - 600 MHz	`0xe`	`R_PPSXX_0xe`

Table continues on the next page...

Table 622. System Setup (continued)

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
P5020DS	Rev X7	Rev 2.0	• CPU - 2000 MHz • CCB - 800 MHz • DDR - 667 MHz • FMan - 600 MHz	`0x36`	`RR_HXAPNSP_0x36`
P5040DS	Rev X4	Rev 2.1	• CPU - 2266 MHz • CCB - 800 MHz • DDR - 800 MHz • FMan - 600 MHz	`0x02`	`RR_XXSNSpP_0x02`
T1023RDB	Rev B	Rev 1.0	• CPU - 1400 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 700 MHz	`0x77`	`RNSS_NPP_77`
T1024RDB	Rev A	Rev 1.0	• CPU - 1400 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 700 MHz	`0x95`	`RRX_PP_95`
T1040RDB	Rev A	Rev 1.1	• CPU - 1400 MHz • CCB - 600 MHz • DDR - 800 MHz • FMan - 600 MHz	`0x66`	`RR_P_66`

Table continues on the next page...

Table 622. System Setup (continued)

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
T2080QDS	Rev X5	Rev 1.1	• CPU - 1800 MHz • CCB - 600 MHz • DDR - 933 MHz • FMan - 700 MHz	6c_2d	RR_PNSNNR_6C_2D
T4240QDS	Rev X4	Rev 2.0	• CPU - 1800 MHz • CCB - 733 MHz • DDR - 933 MHz • FMan - 733 MHz	1_1_5_5	RR_XXXXPRPR_1_1_5_5
T4240RDB	Rev A	Rev 2.0	• CPU - 1800 MHz • CCB - 733 MHz • DDR - 933 MHz • FMan - 733 MHz	27_55_1_9	SSFFPPH_27_55_1_9
LS1043ARDB	Rev A	Rev 1.1	• CPU - 1600 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 500 MHz	1455	RR_FQPP_1455
LS1046ARDB	Rev A	Rev 1.0	• CPU - 1800 MHz • CCB - 700 MHz • DDR - 1050 MHz • FMan - 800 MHz	1133_5559	RR_FFSSPPPH_1133_5559

Network Simulator: Spirent TestCenter Performance Analysis System

The simulator version used was Chassis ver. 4.24.

NOTE

This particular benchmarking process used Spirent TestCenter, although other traffic generators may also be used.

Software Platform Identification

Target Development System Software:

The table below shows the different file names needed for each target development system. Files enumerated are platform specific. The rx.x string in the file names indicate the silicon revision number.

Table 623. Software Platform Identification

Target Development System Software	Image file name/Description
FMan ucode*	fsl_fman_ucode_<soc>_<silicon_rev>_<ucode_version>.bin Please refer to SDK document to use correct ucode version
RCW	P2041: rcw/p2041rdb/RR_PX_0x09/rcw_14g_1500mhz.bin P3041: rcw/p3041ds/RR_HXAPNSP_0x36/rcw_15g_1500mhz.bin P4080: rcw/p4080ds/R_PPSXX_0xe/rcw_2sgmii_1500mhz_rev3.bin P5020: rcw/p5020ds/RR_HXAPNSP_0x36/rcw_15g_2000mhz.bin P5040: rcw/p5040ds/RR_XXSNSpP_0x02/rcw_26g_2267mhz.bin B4860: rcw/b4860qds/N_RSSS_0x2A_0x8D/ rcw_4sgmii_4srio_2xfi_1600mhz_1866ddr_rev2.bin T4240QDS: rcw/t4240qds/RR_XXXXPRPR_1_1_5_5/ rcw_1_1_5_5_1666MHz_rev2.bin T4240RDB: rcw/t4240rdb/SSFFPPH_27_55_1_9/ rcw_27_55_1_9_1666MHz.bin T2080QDS: rcw/t2080qds/RR_PNSNNR_6C_2D/rcw_6c_2d_1800MHz.bin T1040RDB: rcw/t1040rdb/RR_P_66/rcw_1400MHz.bin T1023RDB: rcw/t1023rdb/RNSS_NPP_77/rcw_77_1400MHz.bin T1024RDB: rcw/t1024rdb/RRX_PPP_95/rcw_95_1400MHz.bin LS1043ARDB: rcw/ls1043ardb/RR_FQPP_1455/rcw_1600.bin LS1046ARDB: rcw/ls1046ardb/RR_FFSSPPPH_1133_5559/ rcw_1800_qspiboot.bin
Boot loader	u-boot-${PLATFORM}.bin
Extra bootargs*	bportals=s0 qportals=s0
Linux Kernel (For PowerPC)	uImage-${platform}.bin
Device Tree (For PowerPC)	uImage-${platform}-usdpaa.dtb
File system (For PowerPC)	fsl-image-core-${platform}.ext2.gz.u-boot
FIT (For ARM)	kernel-fsl-${platform}-usdpaa.itb

NOTE

*For more information about boot arguments, refer to USDPAA Boot Arguments.

12.3.2.3 Applications and traffic flow

USDPAA applications for IPv4 forwarding

USDPAA RC-based IPFwd Application Suite

Route cache(RC) based IPFwd takes route decision based upon hash results calculated by FMAN using source IP address and destination IP address in the frame. It contains two applications:

- ipfwd_app: The application that takes charge of initializing devices, managing threads and buffers and packets processing
- ipfwd_config: The command line interface(CLI) to configure ipfwd, such as IP addresses assignment to each port, ip routes and ARP routes management, starting packets processing

USDPAA LPM IPFwd Application Suite

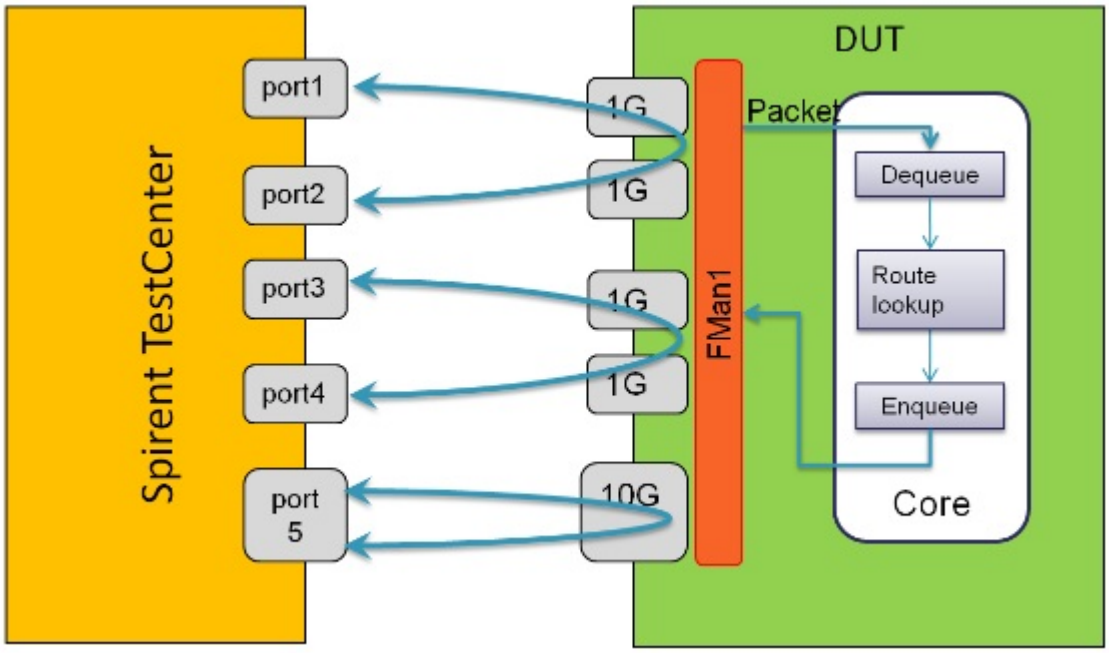
LPM-based IPFwd uses Longest Prefix Match algorithm for doing route lookup by using prefix for destination IP address. It contains two applications:

- lpm_ipfwd_app: The application that takes charge of initializing devices, managing threads and buffers and packets processing
- lpm_ipfwd_config: The command line interface(CLI) to configure ipfwd, such as IP addresses assignment to each port, destination routes and ARP routes management, starting packets processing

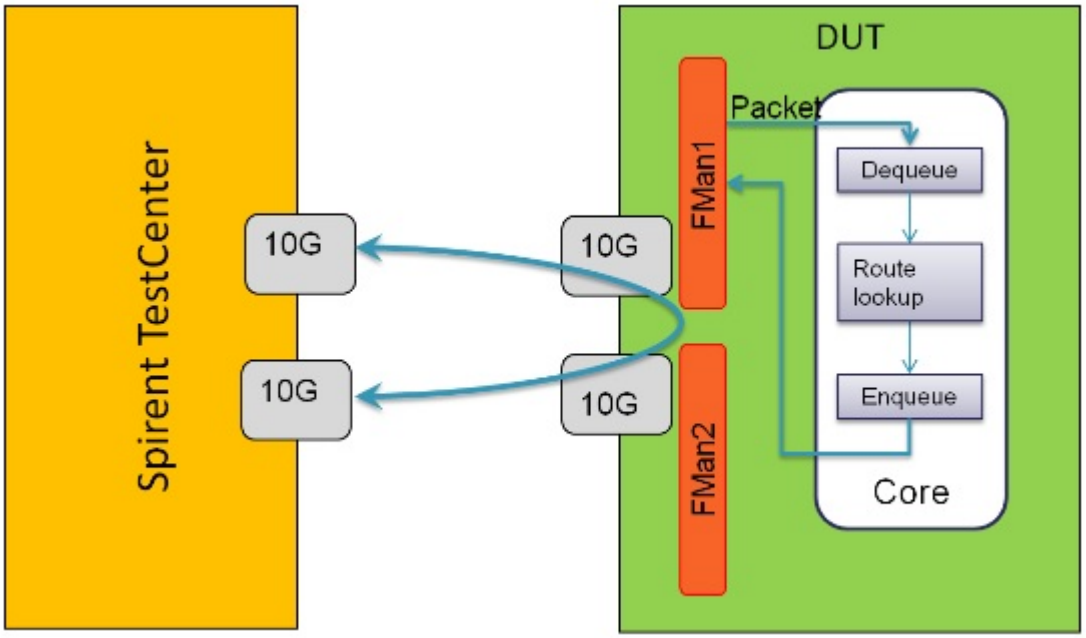
IP forwarding working flow

IP forwarding working flow varies on different platforms.

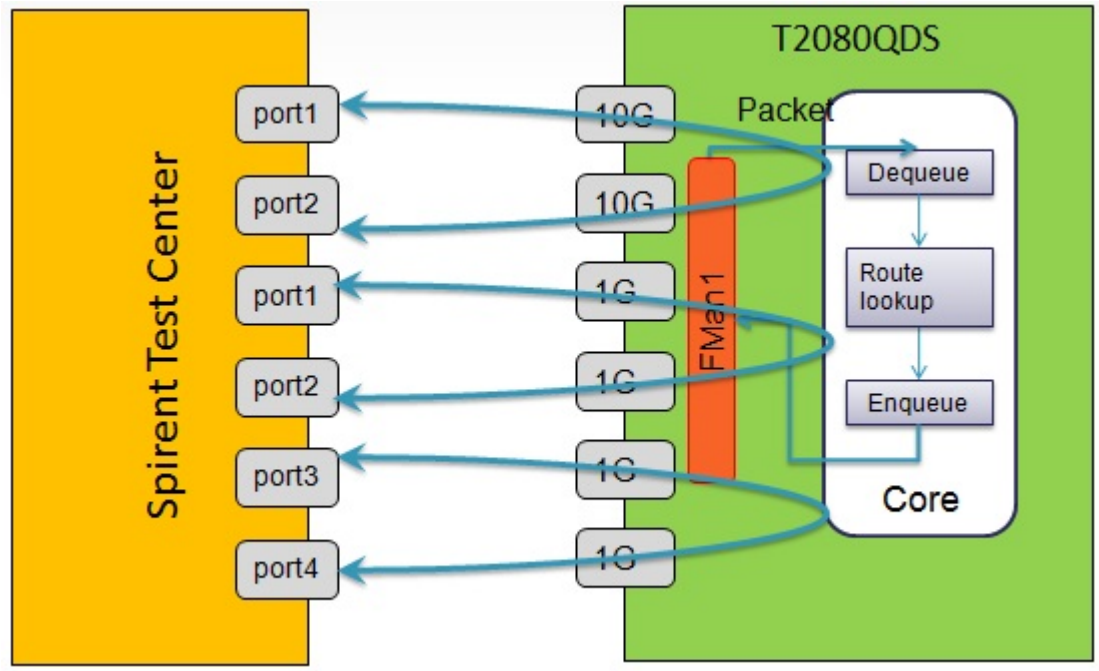
For P2041RBD/P3041DS/P5020DS/LS1046ARDB, they support 14G IP forwarding:



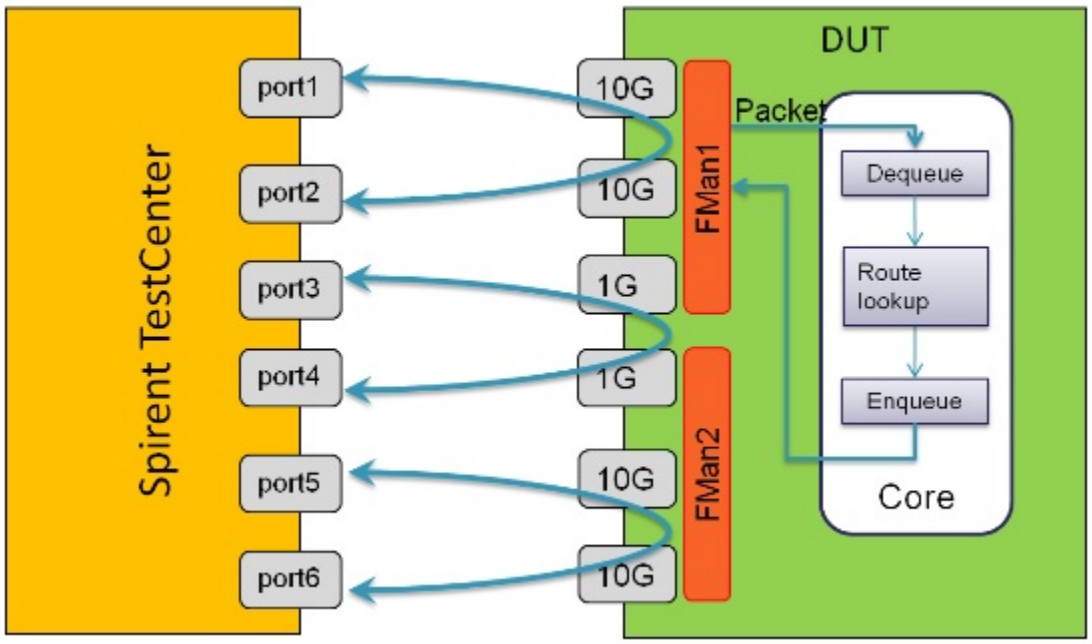
For P4080DS/P5040DS/B4860QDS, they support 20G IP forwarding: B4860QDS has two 10G ports on single FMan.



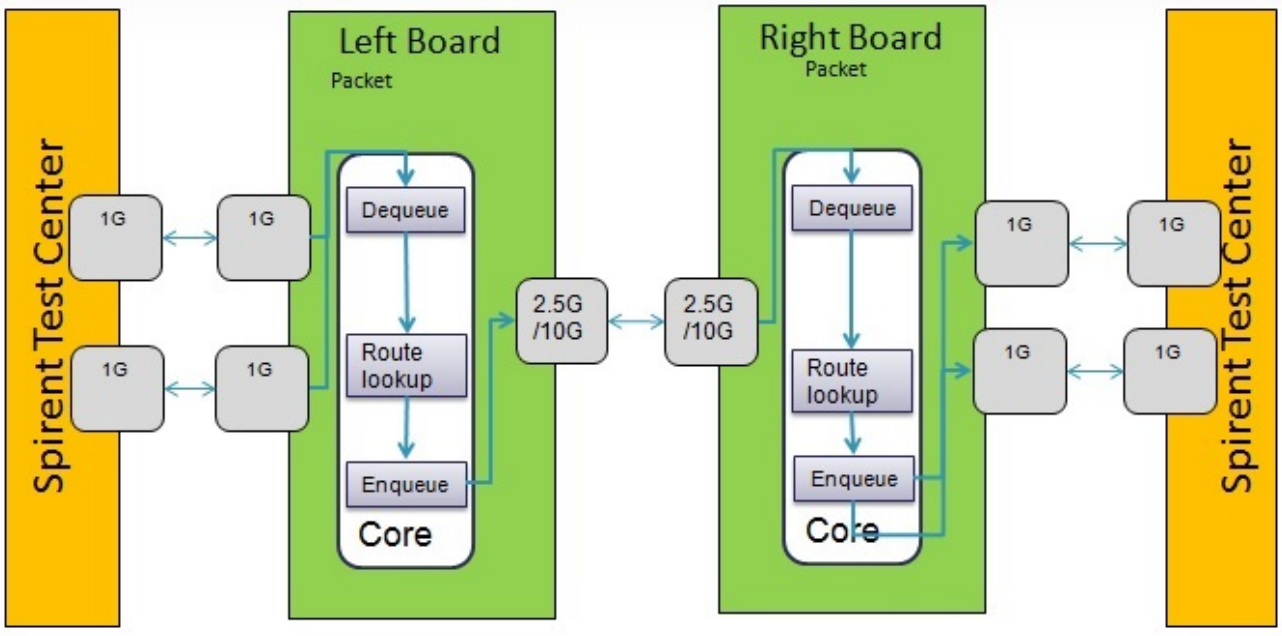
For T2080QDS, they support 24G IP forwarding:



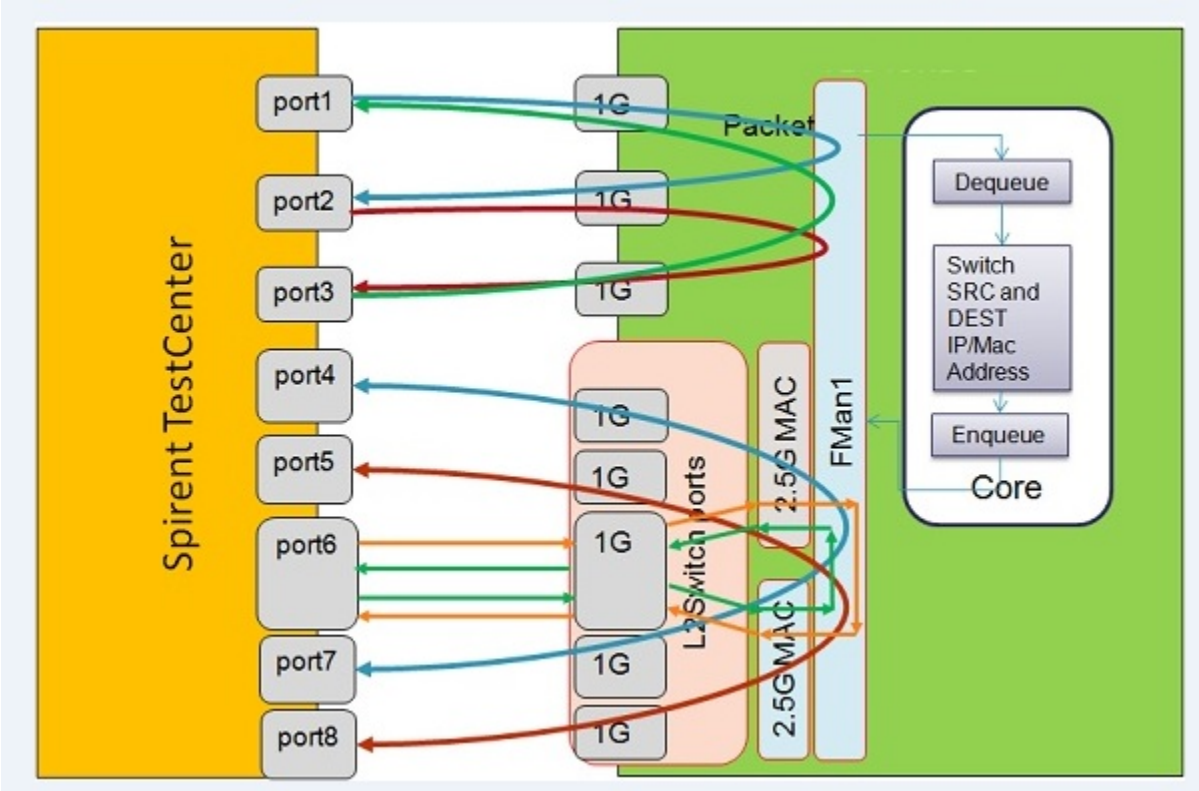
For T4240QDS, it supports 42G IP forwarding:



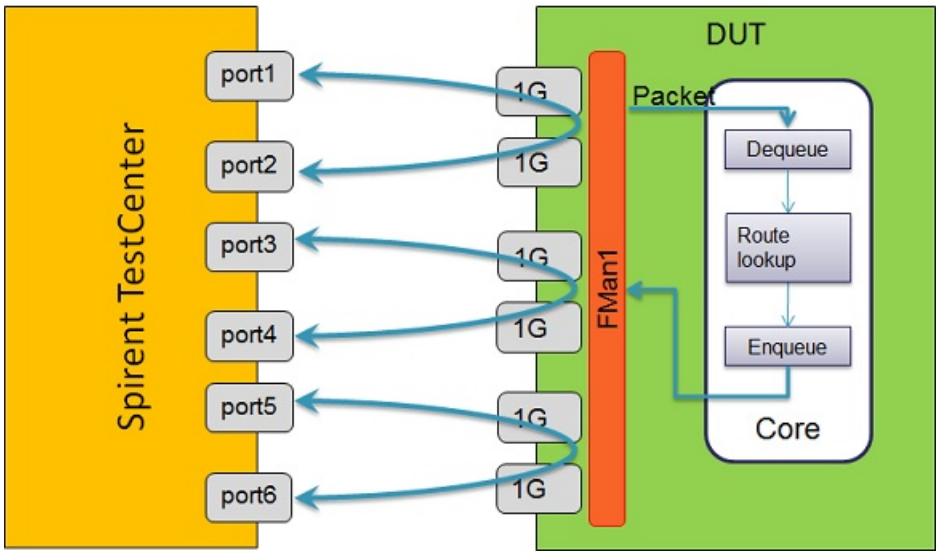
For T1023RDB/T1024RDB, it supports 4G IP forwarding:



For T1040D4RDB, it supports 8G IP forwarding, but traffic flow looks much more complex because there are five 1G MACs on the L2 switch used for traffic transmission:



For LS1043ARDB, it supports 6G IP forwarding



DPAA resource configuration for IPv4 forwarding

Table 624. DPAA resource configuration for USDPAA IP forwarding

DPAA Resource	USDPAA IPv4 Forward Function

FMan	• Parsing on : 32 bytes parser result comes with every frame, such as:
	  • Protocol parsing
	  • IP Header location
	  • IP Header checksum check
	  • Error code
	• Congestion management disabled. Fman side order restoration is on.
	• Hash is based on the SIP,DIP

QMan	• Push mode, Triple dequeue enable.
	• DQRR:
	  • Entry Stashing on.
	  • To L1 cache, with high priority.
	• Frame data, Annotation, Context stashing on: –Frame data = 1 line, Annotation = 1 line, Context = 1
	• Consumer Index write mode cache-inhibited: Update on every 3 frames dequeue.
	• EQRR: Access through Cache Enable Access (CENA, WIMGE=00000)

BMan	• Buffer Pool: bp9 • Buffer Size: 1728 Bytes • Number of buffers: 2048
DMA Memory	• 256MB for 32 bit platform • 256MB for 64 bit platform

12.3.2.4 RC-ipfwd for P2041RDB/P3041DS/P5020DS

Procedure for RC-ipfwd application

1. Boot board with usdpaa mode, then execute commands on board:

Table 625. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	`# ifconfig eth0 192.168.1.200`	set ethernet port IP addr.
b	---	`# cd /usr/etc`	---

Table continues on the next page...

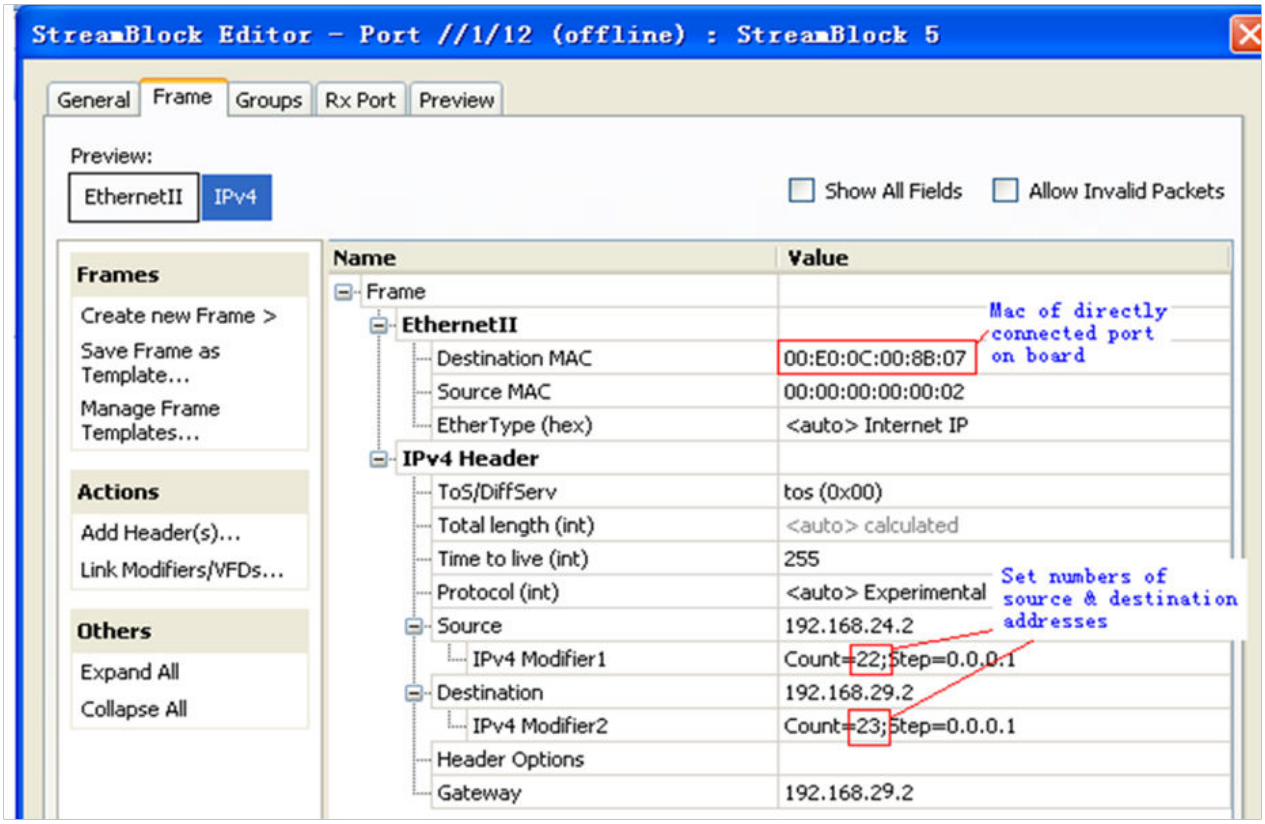
Table 625. Operations on board (continued)

Step	Action	Command on Linux Partition	Notes

c	Configure PCD	`# fmc -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_ipv4.xml -a`
		`# fmc -c usdpaa_config_ls1046.xml -p usdpaa_policy_hash_ipv4.xml -a`
		`(For LS1046ARDB)`	

d	Launch IP forward application which creates one thread on core 1 & 2 individually	`# ipfwd_app 0..1 -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000` `# ipfwd_app 0..1 -c usdpaa_config_ls1046.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	You may use any combination of available cores the platform supports
e	Login Linux from tftp server	`ssh root@192.168.1.200`	---
f	Run IP forwarding configuration script in ssh session	`# ipfwd_14G.sh $PID`	PID is the process id of ipfwd_app. It is printed when ipfwd_app starts
g	Launch TestCenter project to pump traffic	---	Now the ipfwd_app is ready to receive traffic

2. Open TestCenter project and reserve ports. Then create ‘Raw Stream’ in generator for each port, and compose packet as follows (change Mac/IP address, count number according to your environment):


StreamBlock Editor - Port //1/12 (offline) : StreamBlock 5
General
Frame
Groups
Rx Port
Preview
Preview:
EthernetII
IPv4
Show All Fields
Allow Invalid Packets
Frames
Create new Frame >
Save Frame as Template...
Manage Frame Templates...
Actions
Add Header(s)...
Link Modifiers/VFDs...
Others
Expand All
Collapse All
Name
Value
Frame
EthernetII
Destination MAC
00:E0:0C:00:8B:07
Mac of directly connected port on board
Source MAC
00:00:00:00:00:02
EtherType (hex)
<auto> Internet IP
IPv4 Header
ToS/DiffServ
tos (0x00)
Total length (int)
<auto> calculated
Time to live (int)
255
Protocol (int)
<auto> Experimental
Source
192.168.24.2
IPv4 Modifier1
Count=22;Step=0.0.0.1
Set numbers of source & destination addresses
Destination
192.168.29.2
IPv4 Modifier2
Count=23;Step=0.0.0.1
Header Options
Gateway
192.168.29.2

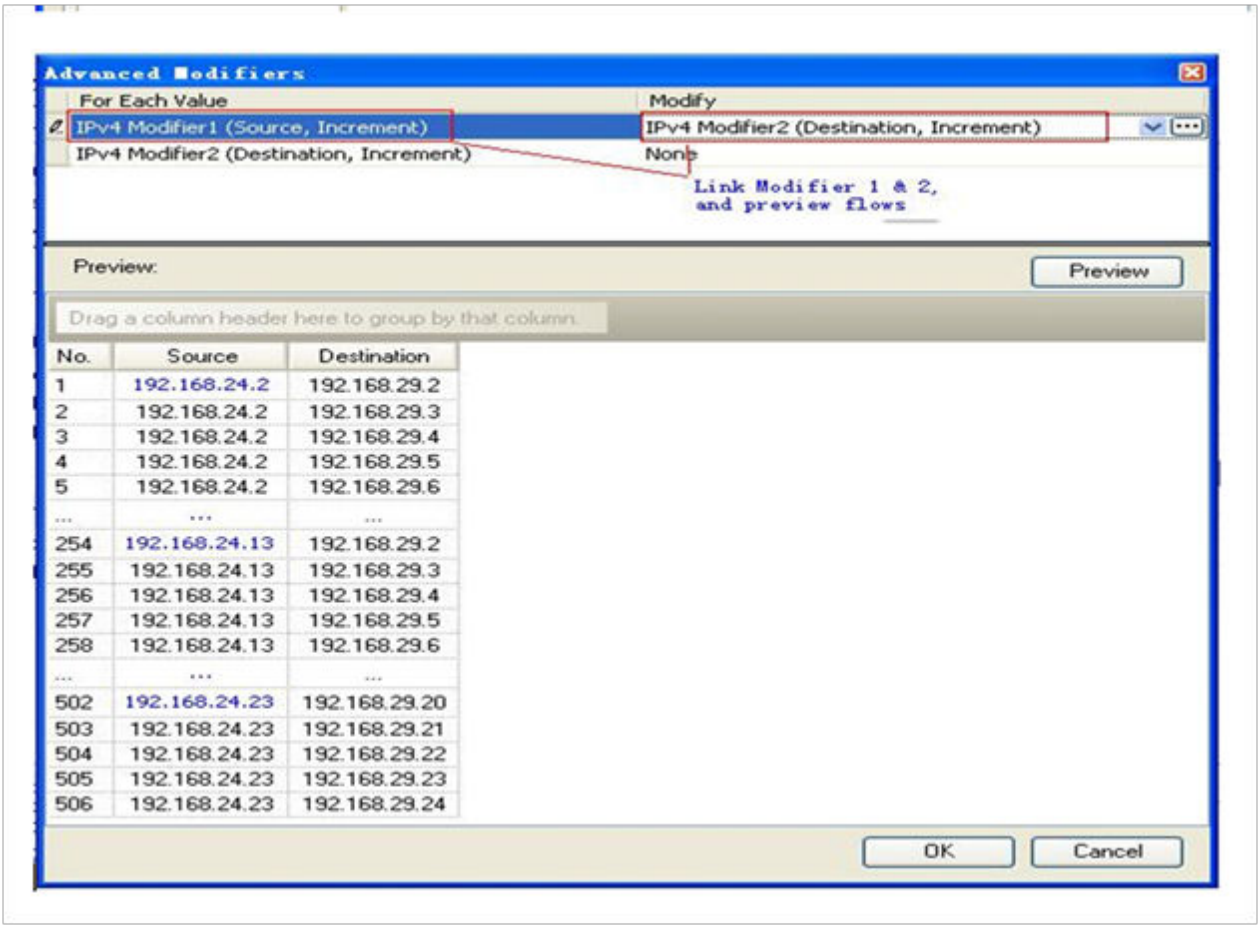

Advanced Modifiers
For Each Value
Modify
IPv4 Modifier1 (Source, Increment)
IPv4 Modifier2 (Destination, Increment)
IPv4 Modifier2 (Destination, Increment)
None
Link Modifier 1 & 2, and preview flows
Preview:
Preview
Drag a column header here to group by that column.
No.
Source
Destination
1 192.168.24.2 192.168.29.2
2 192.168.24.2 192.168.29.3
3 192.168.24.2 192.168.29.4
4 192.168.24.2 192.168.29.5
5 192.168.24.2 192.168.29.6
254 192.168.24.13 192.168.29.2
255 192.168.24.13 192.168.29.3
256 192.168.24.13 192.168.29.4
257 192.168.24.13 192.168.29.5
258 192.168.24.13 192.168.29.6
502 192.168.24.23 192.168.29.20
503 192.168.24.23 192.168.29.21
504 192.168.24.23 192.168.29.22
505 192.168.24.23 192.168.29.23
506 192.168.24.23 192.168.29.24
OK
Cancel

Table 626. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	196	192.168.10.2 –-- 192.168.10.15	192.168.20.2 –-- 192.168.20.15
Port 2 to Port 1	196	192.168.20.2 –-- 192.168.20.15	192.168.10.2 –-- 192.168.10.15
Port 3 to Port 4	196	192.168.40.2 –-- 192.168.40.15	192.168.50.2 –-- 192.168.50.15
Port 4 to Port 3	196	192.168.50.2 –-- 192.168.50.15	192.168.40.2 –-- 192.168.40.15
Port 5 to Port 5	196	192.168.60.2 –-- 192.168.60.15	192.168.60.2 –-- 192.168.60.15
Total # Flows =	980	n/a	n/a

3. Create Command Sequencer in TestCenter and select RFC2544 Throughput template. Then set options as shown in the following table:

Table 627. Command Sequencer configuration

Spirent variable	Variable Setting
Test Duration	60 seconds
Resolution	0.1%
Packet Loss Rate	0.001%
Packet Size	64, 128, 256, 512, 1024, 1280, 1518 bytes

4. Start TestCenter Command Sequencer to measure performance.

12.3.2.5 RC-ipfwd for P4080DS/P5040DS/B4860QDS

Procedure for RC-ipfwd application

1. Boot board with usdpaa mode, then execute commands on board:

Table 628. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	`# ifconfig eth0 192.168.1.200`	set ethernet port IP addr.
b	---	`# cd /usr/etc`	---

Table continues on the next page...

Table 628. Operations on board (continued)

Step	Action	Command on Linux Partition	Notes
c	Configure PCD	`# fmc -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -a`	xml policy file can be found in file system. <fman_config> is: for P4080DS usdpaa_config_p4_serdes_0xe.xml; P5040DS usdpaa_config_p5_serdes_0x02.xml; B4860QDS usdpaa_config_b4_serdes_0x2a_0x98.xml; T2080QDS usdpaa_config_t2_serdes_66_16_22g.xml
d	Launch IP forward application which creates one thread on core 1 & 2 individually	`# ipfwd_app 1..2 -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	You may use any combination of available cores the platform supports.
e	Login Linux from tftp server	`ssh root@192.168.1.200`	---
f	Run IP forward configuration script in ssh session	`# ipfwd_20G.sh $PID`	PID is the process id of ipfwd_app and it is printed when the application starts
g	Launch TestCenter project to pump traffic	---	Now the ipfwd_app is ready

2. Refer to Step 2 & 3 in the section, *RC-ipfwd for P4080DS/P5040DS/B4860QDS*, except IP address assignments.

Table 629. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	506	192.168.60.2 –-- 192.168.60.23	192.168.160.2 –-- 192.168.160.24
Port 2 to Port 1	506	192.168.160.2 –-- 192.168.160.23	192.168.60.2 –-- 192.168.60.24
Total # Flows =	1012	n/a	n/a

3. Start TestCenter Command Sequencer to measure performance.

12.3.2.6 RC-ipfwd for T2080QDS

Procedure for LPM-ipfwd application

1. Create a Shell script 'ipfwd_24G.sh' with following content:

```
pid=$1
   pid=$1
```

```
board_type=$(uname -n)
if [ "$pid" == "" ]
        then
                echo "Give PID to hook up with"
                exit 1
fi

net_pair_routes()
{
        for net in $1 $2
        do
                ipfwd_config -P $pid -B -s 192.168.$net.2 -c $3 \
                -d 192.168.$(expr $1 + $2 - $net).2 -n $4 -g \
                192.168.$(expr $1 + $2 - $net).2
        done
}

ipfwd_config -P $pid -F -a 192.168.0.1 -i 1
ipfwd_config -P $pid -F -a 192.168.1.1 -i 2
ipfwd_config -P $pid -F -a 192.168.2.1 -i 3
ipfwd_config -P $pid -F -a 192.168.3.1 -i 4
ipfwd_config -P $pid -F -a 192.168.8.1 -i 89
ipfwd_config -P $pid -F -a 192.168.9.1 -i 90

ipfwd_config -P $pid -G -s 192.168.0.2 -m 0F:11:22:33:44:00 -r true
ipfwd_config -P $pid -G -s 192.168.1.2 -m 0F:11:22:33:44:01 -r true
ipfwd_config -P $pid -G -s 192.168.2.2 -m 0F:11:22:33:44:02 -r true
ipfwd_config -P $pid -G -s 192.168.3.2 -m 0F:11:22:33:44:03 -r true
ipfwd_config -P $pid -G -s 192.168.8.2 -m 0F:11:22:33:44:08 -r true
ipfwd_config -P $pid -G -s 192.168.9.2 -m 0F:11:22:33:44:09 -r true

                                # 1014
net_pair_routes 0 1 13 13     # 2 * 13 * 13
net_pair_routes 2 3 13 13     # 2 * 13 * 13
net_pair_routes 8 9 13 13     # 2 * 13 * 13
echo IPFwd Route Creation completed
```

2. Boot board with USDPAA mode, then execute the following commands on the board:

Table 630. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	#ifconfig eth0 192.168.1.200	set ethernet port IP addr.
b	---	#cd /usr/etc	---
c	Configure PCD	#fmc –c <fman_config> –p usdpaa_policy_hash_ipv4.xml –a	xml policy file can be found in file system. <fman_config> here is: usdpaa_config_t2_serdes_66_15.xml for T2080QDS

Table continues on the next page...

Table 630. Operations on board (continued)

Step	Action	Command on Linux Partition	Notes
d	Launch IP forward application which creates one thread on core 1, 2 and 3 individually	#ipfwd_app 1..3 -c <fman_config> –p usdpaa_policy_hash_ipv4.xml –d 0x10000000	You may use any combination of available cores the platform supports. On T2080QDS, user could get better performance with 8 cores if user adds parameter '-s' for ipfwd_app. e.g. ipfwd_app 0..7 -c usdpaa_config_t2_serdes_66_16_22g.xml –p usdpaa_policy_hash_ipv4.xml –d 0x10000000
e	Login Linux from tftp server	ssh root@192.168.1.200	---
f	Run IP forward configuration script in ssh session	#ipfwd_24G.sh $PID	PID is the process id of lpm_ipfwd. It is printed when ipfwd starts
g	Launch TestCenter project to pump traffic	---	Now the ipfwd_app is ready

3. Refer to Step 2 & 3 in the section, *RC-ipfwd for T2080QDS*, except IP address assignments.

Table 631. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	169	192.168.0.2 – 192.168.0.14	192.168.01.2 – 192.168.01.14
Port 2 to Port 1	169	192.168.1.2 – 192.168.1.14	192.168.0.2 – 192.168.0.14
Port 3 to Port 4	169	192.168.2.2 – 192.168.2.14	192.168.3.2 – 192.168.3.14
Port 4 to Port 3	169	192.168.3.2 – 192.168.3.14	192.168.2.2 – 192.168.2.14
Port 10G 1 to Port 10G 2	169	192.168.8.2 – 192.168.8.14	192.168.9.2 – 192.168.9.14
Port 10G 2 to Port 10G 1	169	192.168.9.2 – 192.168.9.14	192.168.8.2 – 192.168.8.14
Total # Flows =	1014	n/a	n/a

4. Start TestCenter Command Sequencer to measure performance.

12.3.2.7 RC-ipfwd for T4240QDS/T4240RDB

Procedure for RC-ipfwd application

1. Boot board with USDPAA mode, then execute the following commands on the board:

Table 632. Operations on Board

Step	Action	Command on Linux Partition	Notes
a	---	`# ifconfig eth0 192.168.1.200`	set ethernet port IP addr.
b	---	`# cd /usr/etc`	---
c	Configure PCD	`# fmc -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -a`	<fman_config> for T4240QDS is usdpaa_config_t4_serdes_1_1_5_5.xml, for T4240RDB is usdpaa_config_t4_48g.xml
d	Launch IP forward application which creates one thread on each core	`# ipfwd_app 0..23 -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	You may use any combination of available cores the platform supports
e	Login Linux from tftp server	`ssh root@192.168.1.200`	---
f	Modify IP forwarding configuration script for T4240RDB	sed -i 's/-i 5/-i 1/;s/-i 105/-i 101/' /usr/bin/ipfwd_42G.sh	Ignore this step if you are using T4240QDS
g	Run IP forwarding configuration script in ssh session	`# ipfwd_42G.sh $PID`	PID is the process id of ipfwd_app and it is printed when the application starts
h	Launch TestCenter project to pump traffic	---	Now the ipfwd_app is ready

2. Refer to Step 2 & 3 in the section, *RC-ipfwd for P2041RDB/P3041DS/P5020DS*, except IP address assignments.

Table 633. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	225	192.168.50.2 – 192.168.50.16	192.168.60.2 – 192.168.60.16
Port 2 to Port 1	225	192.168.60.2 – 192.168.60.16	192.168.50.2 – 192.168.50.16
Port 3 to Port 4	49	192.168.40.2 – 192.168.40.8	192.168.130.2 – 192.168.130.8
Port 4 to Port 3	49	192.168.130.2 – 192.168.130.8	192.168.40.2 – 192.168.40.8
Port 5 to Port 6	225	192.168.140.2 – 192.168.140.16	192.168.150.2 – 192.168.150.16

Table continues on the next page...

Table 633. IP addresses assignment (continued)

Packet Flow	Number of Flows	Source Values	Destination Values
Port 6 to Port 5	225	192.168.150.2 – 192.168.150.16	192.168.140.2 – 192.168.140.16
Total # Flows =	998	n/a	n/a

3. Start TestCenter Command Sequencer to measure performance.

12.3.2.8 RC-ipfwd for T1023RDB/T1024RDB

Procedure for RC-ipfwd application

1. Boot two board with usdpaa mode, then execute commands on each board:

Table 634. Operations on board

Step	Action	Command on Linux Partition	Notes
a	Config ethernet port	`# ifconfig eth0 192.168.1.200`	set ethernet port IP addr.
b	Get into PCD file folder	`# cd /usr/etc`	---
c	Configure PCD	`# fmc -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -a`	xml policy file can be found in file system <fman_config> is: T1023RDB: usdpaa_config_t1023_serdes_0x77.xml T1024RDB: usdpaa_config_t1024_serdes_0x99.xml
d	Launch IP forward application which creates one thread on each core	`# ipfwd_app 0..1 -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	You may use any combination of available cores the platform supports
e	Login Linux from tftp server	`ssh root@192.168.1.200`	---

Table continues on the next page...

Table 634. Operations on board (continued)

Step	Action	Command on Linux Partition	Notes

f	Run IP forward configuration script in ssh session	`# cd /usr/bin`
		`# T1023RDB`
		`# sed -i 's/00:04:9f: 01:02:05/00:e0:0c:00:76:02/g' ipfwd_t1023_4.5G.sh`
		`# sed -i 's/00:04:9f: 11:12:24/00:e0:0c:00:4A:02/g' ipfwd_t1023_4.5G.sh`
		`# ipfwd_t1023_4.5G.sh $PID`
		`# T1024RDB`
		`# sed -i 's/00:04:9f: 01:02:05/00:e0:0c:00:6F:00/g' ipfwd_t1024_12G.sh`
		`# sed -i 's/00:04:9f: 11:12:24/00:e0:0c:00:71:00/g' ipfwd_t1024_12G.sh`
		`# ipfwd_t1024_12G.sh $PID`
			PID is the process id of ipfwd_app and it is printed when the application starts

g	Launch TestCenter project to pump traffic	---	Now the ipfwd_app is ready

2. Refer to Step 2 & 3 in the section, *RC-ipfwd for T1023RDB/T1024RDB*, except IP address assignments.

Table 635. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Board 1 Port 1 to board 2 Port 1	256	192.168.20.2 – 192.168.20.17	192.168.40.2 – 192.168.40.17
Board 2 Port 1 to board 1 Port 1	256	192.168.40.2 – 192.168.40.17	192.168.20.2 – 192.168.20.17
Board 1 Port 2 to Board 2 Port 2	256	192.168.30.2 – 192.168.30.17	192.168.50.2 – 192.168.50.17

Table continues on the next page...

Table 635. IP addresses assignment (continued)

Packet Flow	Number of Flows	Source Values	Destination Values
Board 2 Port 2 to board 1 Port 2	256	192.168.50.2 – 192.168.50.17	192.168.30.2 – 192.168.30.17
Total # Flows =	1024	n/a	n/a

3. Start TestCenter Command Sequencer to measure performance.

12.3.2.9 RC-ipfwd for T1040D4RDB

Procedure for RC-ipfwd application

1. Configure FDB entries for L2 switch through application 'l2sw_bin'

```
# l2sw_bin
l2sw_bin> mac add static 00:11:22:33:44:01 0
l2sw_bin> mac add static 00:11:22:33:44:02 1
l2sw_bin> mac add static 00:11:22:33:44:03 2
l2sw_bin> mac add static 00:11:22:33:55:03 2
l2sw_bin> mac add static 00:11:22:33:55:01 3
l2sw_bin> mac add static 00:11:22:33:55:02 4
l2sw_bin> mac add static <fm1-gb0 mac> 8   # replace <...> with real MAC address
l2sw_bin> mac add static <fm1-gb1 mac> 9   # replace <...> with real MAC address
l2sw_bin> Press Ctrl+C to quit l2sw_bin application
```

2. Create a Shell script 'ipfwd_8g_t1040d4rdb.sh' with following content

```
pid=$1

net_pair_routes()
{
    if [ "x$5" == "x" ]
    then
        gwip=2
    else
        gwip=$5
    fi

    for net in $1 $2
    do
        ipfwd_config -P $pid -B -s 192.168.$net.2 -c $3 \
        -d 192.168.$(expr $1 + $2 - $net).$gwip -n $4 \
        -g 192.168.$(expr $1 + $2 - $net).$gwip
    done
}

net_directional_routes()
{
    ipfwd_config -P $pid -B -s 192.168.$1.2 -c $3 \
        -d 192.168.$2.2 -n $4 -g 192.168.$2.2
}

ipfwd_config -P $pid -F -a 192.168.10.1 -i 0
```

```
ipfwd_config -P $pid -F -a 192.168.20.1 -i 1
ipfwd_config -P $pid -F -a 192.168.30.1 -i 2
ipfwd_config -P $pid -F -a 192.168.40.1 -i 3
ipfwd_config -P $pid -F -a 192.168.50.1 -i 4

ipfwd_config -P $pid -G -s 192.168.10.2 -m 00:11:22:33:44:01 -r true
ipfwd_config -P $pid -G -s 192.168.10.8 -m 00:11:22:33:44:02 -r true
ipfwd_config -P $pid -G -s 192.168.10.14 -m 00:11:22:33:44:03 -r true
ipfwd_config -P $pid -G -s 192.168.20.2 -m 00:11:22:33:55:01 -r true
ipfwd_config -P $pid -G -s 192.168.20.8 -m 00:11:22:33:55:02 -r true
ipfwd_config -P $pid -G -s 192.168.20.14 -m 00:11:22:33:55:03 -r true
ipfwd_config -P $pid -G -s 192.168.30.2 -m 02:00:c0:a8:1e:02 -r true
ipfwd_config -P $pid -G -s 192.168.40.2 -m 02:00:c0:a8:28:02 -r true
ipfwd_config -P $pid -G -s 192.168.50.2 -m 02:00:c0:a8:32:02 -r true

# 1050 Flows
net_pair_routes 10 20 14 6 2        # 2 * 14 * 6 = 168
net_pair_routes 10 20 14 6 8        # 2 * 14 * 6 = 168
net_pair_routes 10 20 14 3 14       # 2 * 14 * 3 = 84
net_directional_routes 30 40 14 15  # 14 * 15 = 210
net_directional_routes 40 50 14 15  # 14 * 15 = 210
net_directional_routes 50 30 14 15  # 14 * 15 = 210

echo IPFwd Route Creation completed
```

3. Boot board with usdpaa mode, then execute commands on board:

Table 636. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	`# ifconfig eth0 192.168.1.200`	set ethernet port IP addr.
b	---	`# cd /usr/etc`	---
c	Configure PCD	`# fmc -c usdpaa_config_t1_serdes_0x66.xml -p usdpaa_policy_hash_ipv4.xml -a`	
d	Launch IP forward application which creates one thread on each core	`# ipfwd_app 0..23 -c usdpaa_config_t1_serdes_0x66.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	You may use any combination of available cores the platform supports
e	Login Linux from tftp server	`ssh root@192.168.1.200`	---
f	Run IP forward configuration script in ssh session	`# ipfwd_8g_t1040d4rdb.sh $PID`	PID is the process id of ipfwd_app and it is printed when the application starts
g	Launch TestCenter project to pump traffic	---	Now the ipfwd_app is ready

4. Refer to Step 2 & 3 in the section, *RC-ipfwd for T1040D4RDB*, except IP address assignments.

Table 637. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	210	192.168.30.2 – 192.168.30.15	192.168.40.2 – 192.168.40.16
Port 2 to Port 3	210	192.168.40.2 – 192.168.40.15	192.168.50.2 – 192.168.50.16
Port 3 to Port 1	210	192.168.50.2 – 192.168.50.15	192.168.30.2 – 192.168.30.16
Port 4 to Port 7	84	192.168.10.2 – 192.168.10.15	192.168.20.2 – 192.168.20.7
Port 5 to Port 8	84	192.168.10.2 – 192.168.10.15	192.168.20.8 – 192.168.20.13
Port 6 to Port 6	84	Streamblock 1 192.168.10.2 – 192.168.10.15 Streamblock 2 192.168.20.2 – 192.168.20.15	Streamblock 1 192.168.20.14 – 192.168.20.16 Streamblock 2 192.168.10.14 – 192.168.10.16
Port 7 to Port 4	84	192.168.20.2 – 192.168.20.15	192.168.10.2 – 192.168.10.7
Port 8 to Port 5	84	192.168.20.2 – 192.168.20.15	192.168.10.8 – 192.168.10.13
Total # Flows =	1050	n/a	n/a

5. Start TestCenter Command Sequencer to measure performance.

12.3.2.10 RC-ipfwd for LS1043ARDB

Procedure for RC-ipfwd application

1. Boot board with usdpaa mode, then execute commands on board:

Table 638. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	`# ifconfig eth0 192.168.2.200`	set ethernet port IP addr.
b	---	`# cd /usr/etc`	---
c	Configure PCD	`#fmc -c usdpaa_config_ls1043.xml -p usdpaa_policy_hash_ipv4.xml -a`	xml policy file can be found in file system.

Table continues on the next page...

Table 638. Operations on board (continued)

Step	Action	Command on Linux Partition	Notes
d	Launch IP forward application which creates one thread on each core individually	`# ipfwd_app 0..3 -c usdpaa_config_ls1043.xml -p usdpaa_policy_hash_ipv4.xml`	You may use any combination of available cores the platform supports.
e	Login to board from tftp server	`ssh root@192.168.2.20`	---
f	Run IP forward configuration script in ssh session	`# ipfwd_ls1043a_6G.sh $PID`	PID is the process id of ipfwd_app and it is printed when the application starts
g	Launch TestCenter project to pump traffic	---	Now the ipfwd_app is ready

2. Refer to Step 2 & 3 in the section, *RC-ipfwd for P2041RDB/P3041DS/P5020DS*, except IP address assignments.

Table 639. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	169	192.168.10.2 -- 192.168.10.14	192.168.20.2 -- 192.168.20.14
Port 2 to Port 1	169	192.168.20.2 -- 192.168.20.14	192.168.10.2 -- 192.168.10.14
Port 3 to Port 4	169	192.168.30.2 -- 192.168.40.14	192.168.40.2 -- 192.168.30.14
Port 4 to Port 3	169	192.168.40.2 -- 192.168.30.14	192.168.30.2 -- 192.168.40.14
Port 5 to Port 6	169	192.168.50.2 -- 192.168.60.14	192.168.60.2 -- 192.168.50.14
Port 6 to Port 5	169	192.168.60.2 -- 192.168.50.14	192.168.50.2 -- 192.168.60.14
Total # Flows =	1014	n/a	n/a

3. Start TestCenter Command Sequencer to measure performance.

12.3.2.11 LPM-ipfwd for P2041RDB/P3041DS/P5020DS

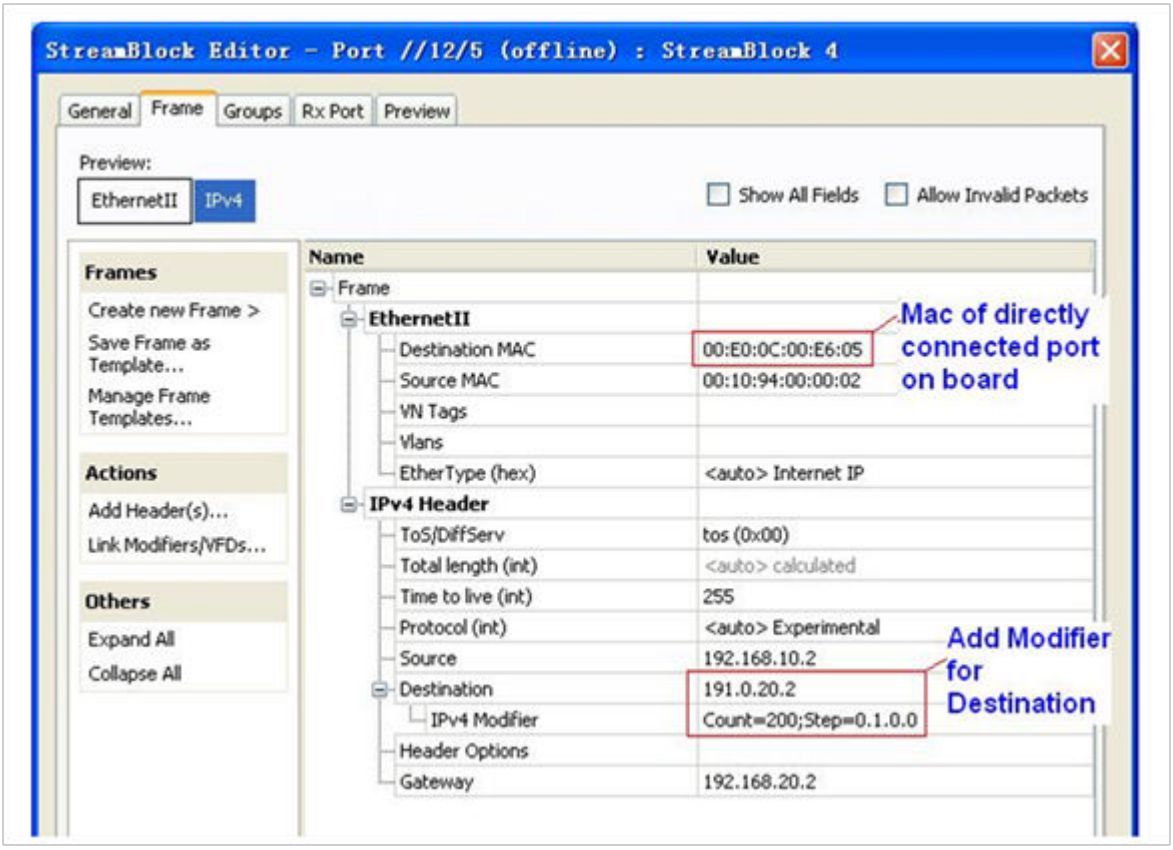
Procedure for LPM-ipfwd application

1. Boot board with usdpaa mode, then execute commands on board:

Table 640. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	#ifconfig eth0 192.168.1.200	set ethernet port IP addr.
b	---	#cd /usr/etc	---
c	Configure PCD	#fmc –c usdpaa_config_p2_p3_p5_14g.xml –p usdpaa_policy_hash_ipv4.xml –a # fmc –c usdpaa_config_ls1046.xml –p usdpaa_policy_hash_ipv4.xml –a (For LS1046ARDB)	
d	Launch IP forward application which creates one thread on core 1, 2 and 3 individually	#lpm_ipfwd_app 1..3 -c usdpaa_config_p2_p3_p5_14g.xml –p usdpaa_policy_hash_ipv4.xml –d 0x10000000 #lpm_ipfwd_app 1..3 -c usdpaa_config_ls1046.xml –p usdpaa_policy_hash_ipv4.xml –d 0x10000000	You may use any combination of available cores the platform supports
e	Login Linux from tftp server	ssh root@192.168.1.200	---
f	Run IP forward configuration script in ssh session	#lpm_ipfwd_14G.sh $PID	PID is the process id of lpm_ipfwd_app. It is printed when lpm_ipfwd_app starts
g	Launch TestCenter project to pump traffic	---	Now the lpm_ipfwd_app is ready to receive traffic

2. Open TestCenter project and reserve ports. Then create ‘Raw Stream’ in generator for each port, and compose packet as following: (change Mac/IP address, count number according to your environment)



NOTE

There is no need to add 'IPv4 Modifier' for 'Source' IP.

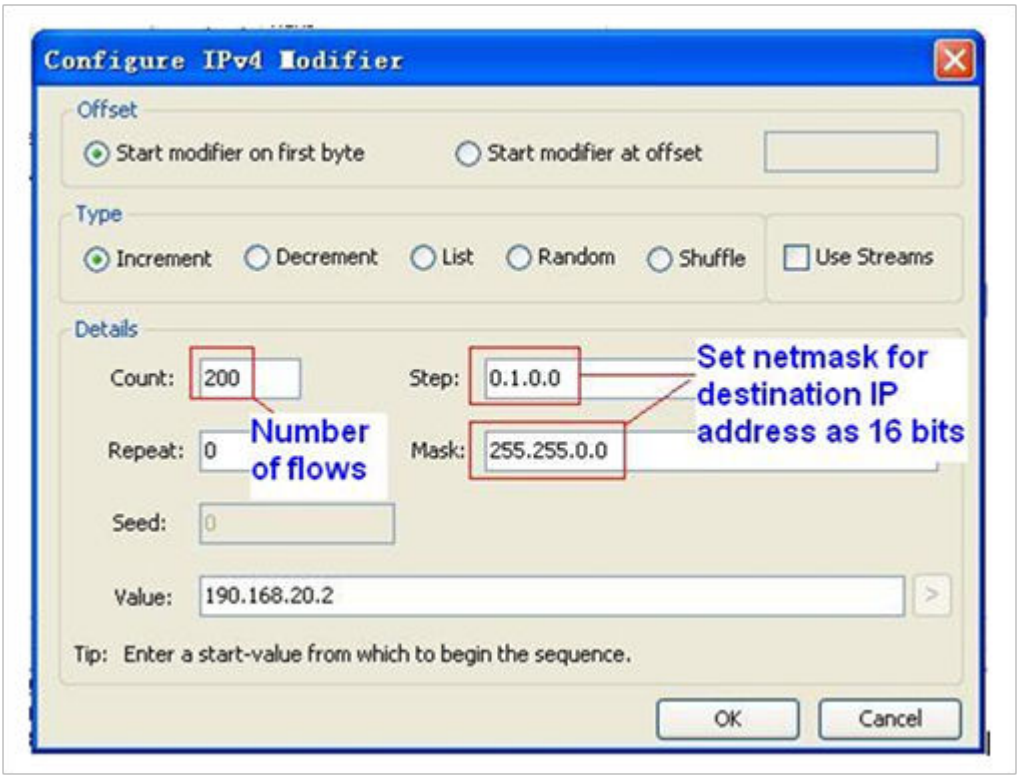


Table 641. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	204	192.168.10.2	191.0.20.2 – 191.203.20.2
Port 2 to Port 1	204	192.168.20.2	190.0.10.2 – 190.203.10.2
Port 3 to Port 4	204	192.168.40.2	194.0.50.2 – 194.203.50.2
Port 4 to Port 3	204	192.168.50.2	193.0.40.2 – 193.203.40.2
Port 5 to Port 5	204	192.168.60.2	195.0.60.2 – 195.203.60.2
Total # Flows =	1020	n/a	n/a

3. Create Command Sequencer in TestCenter and select RFC2544 Throughput template. Then set options as shown in table below.

Table 642. Command Sequencer configuration

Spirent variable	Variable Setting
Test Duration	60 seconds
Resolution	0.1%
Packet Loss Rate	0.001%
Packet Size	64, 128, 256, 512, 1024, 1280, 1518 bytes

4. Start TestCenter Command Sequencer to measure performance.

12.3.2.12 LPM-ipfwd for P4080DS/P5040DS/B4860QDS

Procedure for LPM-ipfwd application

1. Boot board with USDPAA mode, then execute the following commands on the board:

Table 643. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	#ifconfig eth0 192.168.1.200	set ethernet port IP addr.
b	---	#cd /usr/etc	---
c	Configure PCD	#fmc –c <fman_config> –p usdpaa_policy_hash_ipv4.xml –a	xml policy file can be found in file system. <fman_config> here is: for P4080DS usdpaa_config_p4_serdes_0xe.xml; P5040DS usdpaa_config_p5_serdes_0x02.xml; B4860QDS usdpaa_config_b4_serdes_0x2a_0x98.xml

Table continues on the next page...

Table 643. Operations on board (continued)

Step	Action	Command on Linux Partition	Notes
d	Launch IP forward application which creates one thread on core 1, 2 and 3 individually	#lpm_ipfwd_app 1..3 -c <fman_config> –p usdpaa_policy_hash_ipv4.xml –d 0x10000000	You may use any combination of available cores the platform supports.
e	Login Linux from tftp server	ssh root@192.168.1.200	---
f	Run IP forward configuration script in ssh session	#lpm_ipfwd_20G.sh $PID	PID is the process id of lpm_ipfwd. It is printed when lpm_ipfwd starts
g	Launch TestCenter project to pump traffic	---	Now the lpm_ipfwd_app is ready

2. Refer to Step 2 & 3 in the section, *LPM-ipfwd for P4080DS/P5040DS/B4860QDS*, except IP address assignments.

Table 644. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	512	192.168.60.2	192.0.160.2 – 193.255.160.2
Port 2 to Port 1	512	192.168.160.2	190.0.60.2 –-- 191.255.60.2
Total # Flows =	1024	n/a	n/a

3. Start TestCenter Command Sequencer to measure performance.

12.3.2.13 LPM-ipfwd for T2080QDS

Procedure for LPM-ipfwd application

1. Create a Shell script 'lpm_ipfwd_24G.sh' with following content:

```
pid=$1
board_type=$(uname -n)
if [ "$pid" == "" ]
        then
                echo "Give PID to hook up with"
                exit 1
fi

net_pair_routes()
{
        net=0
        while [ "$net" -le $5 ]
        do
                lpm_ipfwd_config -P $pid -B -c $2 -d $1.$net.24.2 -n $3 -g \
                192.168.$4.2
                net=`expr $net + 1`
        done
}

lpm_ipfwd_config -P $pid -F -a 192.168.0.1 -i 1
lpm_ipfwd_config -P $pid -F -a 192.168.1.1 -i 2
lpm_ipfwd_config -P $pid -F -a 192.168.2.1 -i 3
lpm_ipfwd_config -P $pid -F -a 192.168.3.1 -i 4
```

```
lpm_ipfwd_config -P $pid -F -a 192.168.8.1 -i 89
lpm_ipfwd_config -P $pid -F -a 192.168.9.1 -i 90

lpm_ipfwd_config -P $pid -G -s 192.168.0.2 -m 0F:11:22:33:44:00 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.1.2 -m 0F:11:22:33:44:01 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.2.2 -m 0F:11:22:33:44:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.3.2 -m 0F:11:22:33:44:03 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.8.2 -m 0F:11:22:33:44:08 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.9.2 -m 0F:11:22:33:44:09 -r true

                               # 1020
net_pair_routes 100 1 16 0 170 # 170
net_pair_routes 101 1 16 1 170 # 170
net_pair_routes 102 1 16 2 170 # 170
net_pair_routes 103 1 16 3 170 # 170
net_pair_routes 108 1 16 8 170 # 170
net_pair_routes 109 1 16 9 170 # 170
echo LPM-IPFwd Route Creation completed
```

2. Boot board with USDPAA mode, then execute the following commands on the board:

Table 645. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	#ifconfig eth0 192.168.1.200	set ethernet port IP addr.
b	---	#cd /usr/etc	---
c	Configure PCD	#fmc –c <fman_config> –p usdpaa_policy_hash_ipv4.xml –a	xml policy file can be found in file system. <fman_config> here is: usdpaa_config_t2_serdes_66_15.xml for T2080QDS
d	Launch IP forward application which creates one thread on core 1, 2 and 3 individually	#lpm_ipfwd_app 1..3 -c <fman_config> –p usdpaa_policy_hash_ipv4.xml –d 0x10000000	You may use any combination of available cores the platform supports. On T2080QDS, user could get better performance with 8 cores if user adds parameter '-s' for lpm_ipfwd_app. e.g. lpm_ipfwd_app 0..7 -c usdpaa_config_t2_serdes_66_16_22g.xml –p usdpaa_policy_hash_ipv4.xml –d 0x10000000 Note: If the performance for 8 core is less than 6 core on T2080QDS.Running lpm_ipfwd_app with "-s" option to bridge the gap between two conifguration.
e	Login Linux from tftp server	ssh root@192.168.1.200	---

Table continues on the next page...

Table 645. Operations on board (continued)

Step	Action	Command on Linux Partition	Notes
f	Run IP forward configuration script in ssh session	#lpm_ipfwd_24G.sh $PID	PID is the process id of lpm_ipfwd. It is printed when lpm_ipfwd starts
g	Launch TestCenter project to pump traffic	---	Now the lpm_ipfwd_app is ready

3. Refer to Step 2 & 3 in the section, *LPM-ipfwd for T2080QDS*, except IP address assignments.

Table 646. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values
Port 1 to Port 2	170	101.0.24.2	100.0.24.2 – 100.169.24.2
Port 2 to Port 1	170	100.0.24.2	101.0.24.2 – 101.169.24.2
Port 3 to Port 4	170	103.0.24.2	102.0.24.2 – 102.169.24.2
Port 4 to Port 3	170	102.0.24.2	103.0.24.2 – 103.169.24.2
Port 10G 1 to Port 10G 2	170	109.0.24.2	108.0.24.2 – 108.169.24.2
Port 10G 2 to Port 10G 1	170	108.0.24.2	109.0.24.2 – 109.169.24.2
Total # Flows =	1020	n/a	n/a

4. Start TestCenter Command Sequencer to measure performance.

12.3.2.14 LPM-ipfwd for T4240QDS/T4240RDB

Procedure for LPM-ipfwd application

1. Boot board with usdpaa mode, then execute commands on board:

Table 647. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	#ifconfig eth0 192.168.1.200	set ethernet port IP addr.
b	---	#cd /usr/etc	---
c	Configure PCD	#fmc –c <fman_config> –p usdpaa_policy_hash_ipv4.xml –a	<fman_config> for T4240QDS is usdpaa_config_t4_serdes_1_1_5_5.xml, for T4240RDB is usdpaa_config_t4_48g.xml

Table continues on the next page...

Table 647. Operations on board (continued)

Step	Action	Command on Linux Partition	Notes
d	Launch IP forward application which creates one thread on core 1, 2 and 3 individually	#lpm_ipfwd_app 1..3 -c <fman_config> –p usdpaa_policy_hash_ipv4.xml –d 0x10000000	You may use any combination of available cores the platform supports. Note: If the performance for 24 core is less than 16 core on T4240 board.Running lpm_ipfwd_app with "-s" option to bridge the gap between two conifguration.
e	Login Linux from tftp server	ssh root@192.168.1.200	---
f	Modify IP forwarding configuration script for T4240RDB	sed -i 's/-i 5/i 1/;s/-i 105/-i 101/' /usr/bin/lpm_ipfwd_42G.sh	Ignore this step if you are using T4240RDB
g	Run IP forwarding configuration script in ssh session	#lpm_ipfwd_42G.sh $PID	PID is the process id of lpm_ipfwd. It is printed when lpm_ipfwd starts
h	Launch TestCenter project to pump traffic	---	Now the lpm_ipfwd_app is ready

2. Refer to Step 2 & 3 in the section, *RC-ipfwd for T4240QDS/T4240RDB*, except IP address assignments.

Table 648. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values	NetMask
Port 1 to Port 2	224	192.168.50.2	192.0.24.2 – 192.223.24.2	255.255.0.0
Port 2 to Port 1	224	192.168.60.2	191.0.24.2 – 191.223.24.2	255.255.0.0
Port 3 to Port 4	64	192.168.40.2	193.0.24.2 – 193.63.24.2	255.255.0.0
Port 4 to Port 3	64	192.168.130.2	190.0.24.2 – 190.63.24.2	255.255.0.0
Port 5 to Port 6	224	192.168.140.2	195.0.24.2 – 195.223.24.2	255.255.0.0
Port 6 to Port 5	224	192.168.150.2	194.0.24.2 – 194.223.24.2	255.255.0.0
Total # Flows =	1024	n/a	n/a	n/a

3. Start TestCenter Command Sequencer to measure performance.

12.3.2.15 LPM-ipfwd for T1023RDB/T1024RDB

Procedure for LPM-ipfwd application

1. Boot two board with usdpaa mode, then execute commands on each board:

Table 649. Operations on board

Step	Action	Command on Linux Partition	Notes
a	Config ethernet port	#ifconfig eth0 192.168.1.200	set ethernet port IP addr.
b	Get into PCD file folder	#cd /usr/etc	---
c	Configure PCD	`# fmc -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -a`	xml policy file can be found in file system <fman_config> is: T1023RDB: usdpaa_config_t1023_serdes_0x77.xml T1024RDB: usdpaa_config_t1024_serdes_0x99.xml
d	Launch IP forward application which creates one thread on core 1, 2 and 3 individually	`# lpm_ipfwd_app 0..1 -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	You may use any combination of available cores the platform supports
e	Login Linux from tftp server	ssh root@192.168.1.200	---

f	Run IP forward configuration script in ssh session	`# cd /usr/bin`
		`# T1023RDB`
		`# sed -i 's/00:04:9f:01:02:05/00:e0:0c:00:76:02/g' lpm_ipfwd_t1023_4.5G.sh`
		`# sed -i 's/00:04:9f:11:12:24/00:e0:0c:00:4A:02/g' lpm_ipfwd_t1023_4.5G.sh`
		`# lpm_ipfwd_t1023_4.5G.sh $PID`
		`# T1024RDB`
		`# sed -i 's/00:04:9f:01:02:05/00:e0:0c:00:6F:00/g' lpm_ipfwd_t1024_12G.sh`
		`# sed -i 's/00:04:9f:11:12:24/00:e0:0c:00:71:00/g' lpm_ipfwd_t1024_12G.sh`
		`# lpm_ipfwd_t1024_12G.sh $PID`
			PID is the process id of lpm_ipfwd. It is printed when lpm_ipfwd starts

g	Launch TestCenter project to pump traffic	---	Now the lpm_ipfwd_app is ready

2. Refer to Step 2 & 3 in the section, *LPM-ipfwd for T1023RDB/T1024RDB*, except IP address assignments.

Table 650. IP addresses assignment

Packet Flow	Number of Flows	Source Values	Destination Values	NetMask
Board 1 Port 1 to Board 2 Port 1	256	192.168.20.2	195.0.24.2 – 195.255.24.2	255.255.0.0
Board 1 Port 2 to Board 2 Port 2	256	192.168.30.2	196.0.24.2 – 196.255.24.2	255.255.0.0
Board 2 Port 1 to Board 1 Port 1	256	192.168.20.2	193.0.24.2 – 193.255.24.2	255.255.0.0
Board 2 Port 2 to Board 1 Port 2	256	192.168.30.2	194.0.24.2 – 194.255.24.2	255.255.0.0
Total # Flows =	1024	n/a	n/a	n/a

3. Start TestCenter Command Sequencer to measure performance.

12.3.2.16 LPM-ipfwd for T1040D4RDB

Procedure for LPM-ipfwd application

1. Configure FDB entries for L2 switch through application 'l2sw_bin'

```
# l2sw_bin
l2sw_bin> mac add static 00:11:22:33:44:01 0
l2sw_bin> mac add static 00:11:22:33:44:02 1
l2sw_bin> mac add static 00:11:22:33:44:03 2
l2sw_bin> mac add static 00:11:22:33:55:03 2
l2sw_bin> mac add static 00:11:22:33:55:01 3
l2sw_bin> mac add static 00:11:22:33:55:02 4
l2sw_bin> mac add static <fm1-gb0 mac> 8   # replace <...> with real MAC address
l2sw_bin> mac add static <fm1-gb1 mac> 9   # replace <...> with real MAC address
l2sw_bin> Press Ctrl+C to quit l2sw_bin application
```

2. Create a Shell script 'lpm_ipfwd_8g_t1040d4rdb.sh' with following content:

```
pid=$1

net_pair_routes()
{
    if [ "x$6" == "x" ]
    then
        net=0
        gwip=2
    else
        net=$6
        gwip=$6
    fi

    while [ "$net" -le $5 ]
    do
        lpm_ipfwd_config -P $pid -B -c $2 -d $1.$net.24.2 -n $3 -g \
        192.168.$4.$gwip
        net=`expr $net + 1`
```

```
    done
}

lpm_ipfwd_config -P $pid -F -a 192.168.10.1 -i 0
lpm_ipfwd_config -P $pid -F -a 192.168.20.1 -i 1
lpm_ipfwd_config -P $pid -F -a 192.168.30.1 -i 2
lpm_ipfwd_config -P $pid -F -a 192.168.40.1 -i 3
lpm_ipfwd_config -P $pid -F -a 192.168.50.1 -i 4

lpm_ipfwd_config -P $pid -G -s 192.168.10.2 -m 00:11:22:33:44:01 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.10.82 -m 00:11:22:33:44:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.10.164 -m 00:11:22:33:44:03 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.20.2 -m 00:11:22:33:55:01 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.20.82 -m 00:11:22:33:55:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.20.164 -m 00:11:22:33:55:03 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.30.2 -m 02:00:c0:a8:82:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.40.2 -m 02:00:c0:a8:8c:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.50.2 -m 02:00:c0:a8:a0:02 -r true

# 1024 Flows
net_pair_routes 190 1 16 10 81            # 82
net_pair_routes 190 1 16 10 163 82        # 82
net_pair_routes 190 1 16 10 205 164       # 42
net_pair_routes 191 1 16 20 81            # 82
net_pair_routes 191 1 16 20 163 82        # 82
net_pair_routes 191 1 16 20 205 164       # 42
net_pair_routes 192 1 16 30 203           # 204
net_pair_routes 193 1 16 40 203           # 204
net_pair_routes 194 1 16 50 203           # 204

echo LPM-IPFwd Route Creation completed
```

3. Boot board with usdpaa mode, then execute commands on board:

Table 651. Operations on board

Step	Action	Command on Linux Partition	Notes
a	---	#ifconfig eth0 192.168.1.200	set ethernet port IP addr.
b	---	#cd /usr/etc	---
c	Configure PCD	#fmc –c usdpaa_config_t1_serdes_0x66.xml –p usdpaa_policy_hash_ipv4.xml –a	
d	Launch IP forward application which creates one thread on core 1, 2 and 3 individually	#lpm_ipfwd_app 1..3 -c usdpaa_config_t1_serdes_0x66.xml –p usdpaa_policy_hash_ipv4.xml –d 0x10000000	You may use any combination of available cores the platform supports
e	Login Linux from tftp server	ssh root@192.168.1.200	---
f	Run IP forward configuration script in ssh session	#lpm_ipfwd_8g_t1040d4rdb.sh $PID	PID is the process id of lpm_ipfwd. It is printed when lpm_ipfwd starts
g	Launch TestCenter project to pump traffic	---	Now the lpm_ipfwd_app is ready

4. Refer to Step 2 & 3 in the section, *LPM-ipfwd for T1040D4RDB*, except IP address assignments.

Table 652. IP addresses assignment

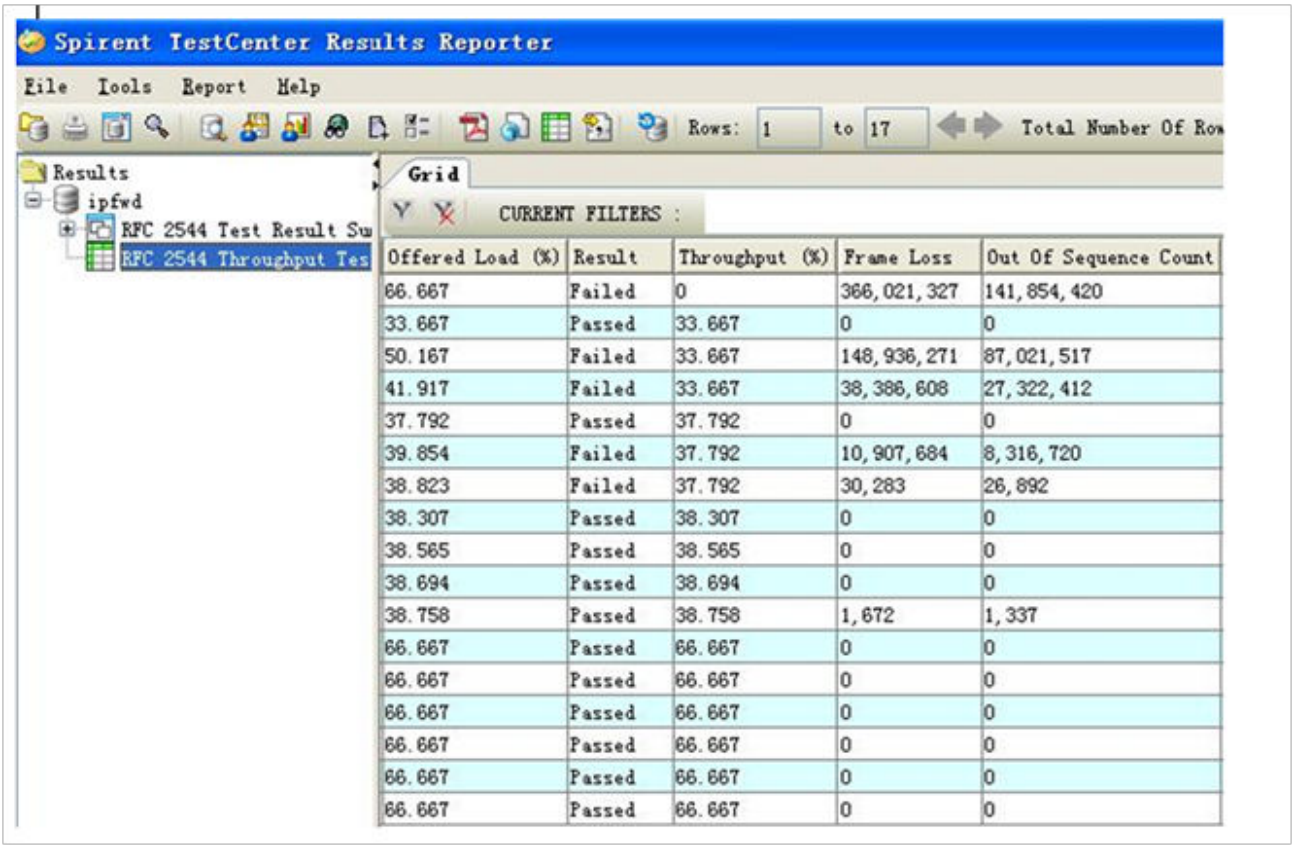
Packet Flow	Number of Flows	Source Values	Destination Values	NetMask
Port 1 to Port 2	204	192.168.30.2	193.0.24.2 – 193.203.24.2	255.255.0.0
Port 2 to Port 3	204	192.168.40.2	194.0.24.2 – 194.203.24.2	255.255.0.0
Port 3 to Port 1	204	192.168.50.2	192.0.24.2 – 192.203.24.2	255.255.0.0
Port 4 to Port 7	82	192.168.10.2	191.0.24.2 – 191.81.24.2	255.255.0.0
Port 5 to Port 8	82	192.168.10.2	191.82.24.2 – 191.163.24.2	255.255.0.0
Port 6 to Port 6	82	Streamblock 1 192.168.10.2 Streamblock 2 192.168.20.2	Streamblock 1 191.164.24.2 – 191.205.24.2 Streamblock 2 190.164.24.2 – 190.205.24.2	255.255.0.0
Port 4 to Port 7	82	192.168.20.2	190.0.24.2 – 190.81.24.2	255.255.0.0
Port 5 to Port 8	82	192.168.20.2	190.82.24.2 – 190.163.24.2	255.255.0.0
Total # Flows =	1024	n/a	n/a	n/a

5. Start TestCenter Command Sequencer to measure performance.

12.3.2.17 View of Test Result

To measure performance by TestCenter RFC 2544 Command Sequence, see the summary result using Spirent TestCenter Results Reporter.

If Order Preservation or Order Restoration is enabled, and TestCenter sends Zero Loss traffic, then count of Out of Sequence should be 0.



Offered Load (%)	Result	Throughput (%)	Frame Loss	Out Of Sequence Count
66.667	Failed	0	366,021,327	141,854,420
33.667	Passed	33.667	0	0
50.167	Failed	33.667	148,936,271	87,021,517
41.917	Failed	33.667	38,386,608	27,322,412
37.792	Passed	37.792	0	0
39.854	Failed	37.792	10,907,684	8,316,720
38.823	Failed	37.792	30,283	26,892
38.307	Passed	38.307	0	0
38.565	Passed	38.565	0	0
38.694	Passed	38.694	0	0
38.758	Passed	38.758	1,672	1,337
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0

12.3.3 DPAA IPSec

Details for reproducing USDPAA packet forwarding performance results.

12.3.3.1 Introduction

This document describes the steps to verify performance of USDPAA IPSec forwarding.

12.3.3.2 Platform Identification

Hardware Platform Identification

Depending on the platform, the application uses the setup shown in the table below.

Table 653. System Setup

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
B4860QDS	Rev B2	Rev 2.2	• CPU - 1600 MHz • CCB - 667 MHz • DDR - 933 MHz • FMan - 667 MHz	0x2A_0x8D	N_RSSS_0x2A_ 0x8D
P2041RDB	Rev B	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 667 MHz • FMan - 583 MHz	0x09	RR_PX_0x09
P3041DS	Rev X1	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 667 MHz • FMan - 583 MHz	0x36	RR_HXAPNSP_0 x36
P4080DS	Rev X3	Rev 3.0	• CPU - 1500 MHz • CCB - 800 MHz • DDR - 650 MHz • FMan - 600 MHz	0xe	R_PPSXX_0xe
P5020DS	Rev X7	Rev 2.0	• CPU - 2000 MHz • CCB - 800 MHz • DDR - 667 MHz • FMan - 600 MHz	0x36	RR_HXAPNSP_0 x36

Table continues on the next page...

Table 653. System Setup (continued)

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
P5040DS	Rev X4	Rev 2.1	• CPU - 2266 MHz • CCB - 800 MHz • DDR - 800 MHz • FMan - 600 MHz	`0x02`	`RR_XXSNSpP_0x02`
T1023RDB	Rev B	Rev 1.0	• CPU - 1400 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 700 MHz	`0x77`	`RNSS_NPP_77`
T1024RDB	Rev A	Rev 1.0	• CPU - 1400 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 700 MHz	`0x95`	`RRX_PP_95`
T1040RDB	Rev A	Rev 1.1	• CPU - 1400 MHz • CCB - 600 MHz • DDR - 800 MHz • FMan - 600 MHz	`0x66`	`RR_P_66`
T2080QDS	Rev X5	Rev 1.1	• CPU - 1800 MHz • CCB - 600 MHz • DDR - 933 MHz • FMan - 700 MHz	`6c_2d`	`RR_PNSNNR_6C_2D`

Table continues on the next page...

Table 653. System Setup (continued)

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/ FMan	SerDes Protocol	Config. Name
T4240QDS	Rev X4	Rev 2.0	• CPU - 1800 MHz • CCB - 733 MHz • DDR - 933 MHz • FMan - 733 MHz	1_1_5_5	RR_XXXXPRPR_ 1_1_5_5
T4240RDB	Rev A	Rev 2.0	• CPU - 1800 MHz • CCB - 733 MHz • DDR - 933 MHz • FMan - 733 MHz	27_55_1_9	SSFFPPH_27_5 5_1_9
LS1043ARDB	Rev A	Rev 1.1	• CPU - 1600 MHz • CCB - 400 MHz • DDR - 800 MHz • FMan - 500 MHz	1455	RR_FQPP_1455
LS1046ARDB	Rev A	Rev 1.0	• CPU - 1800 MHz • CCB - 700 MHz • DDR - 1050 MHz • FMan - 800 MHz	1133_5559	RR_FFSSPPPH_ 1133_5559

Network Simulator: Spirent TestCenter Performance Analysis System

The simulator version used was Chassis ver. 4.24.

NOTE

This particular benchmarking process used Spirent TestCenter, although other traffic generators may also be used.

Software Platform Identification

Target Development System Software:

The table below shows the different file names needed for each target development system. Files enumerated are platform specific. The rx.x string in the file names indicate the silicon revision number.

Table 654. Software Platform Identification

Target Development System Software	Image file name/Description
FMan ucode*	fsl_fman_ucode_<soc>_<silicon_rev>_<ucode_version>.bin Please refer to SDK document to use correct ucode version
RCW	P2041: rcw/p2041rdb/RR_PX_0x09/rcw_14g_1500mhz.bin P3041: rcw/p3041ds/RR_HXAPNSP_0x36/rcw_15g_1500mhz.bin P4080: rcw/p4080ds/R_PPSXX_0xe/rcw_2sgmii_1500mhz_rev3.bin P5020: rcw/p5020ds/RR_HXAPNSP_0x36/rcw_15g_2000mhz.bin P5040: rcw/p5040ds/RR_XXSNSpP_0x02/rcw_26g_2267mhz.bin B4860: rcw/b4860qds/N_RSSS_0x2A_0x8D/ rcw_4sgmii_4srio_2xfi_1600mhz_1866ddr_rev2.bin T4240QDS: rcw/t4240qds/RR_XXXXPRPR_1_1_5_5/ rcw_1_1_5_5_1666MHz_rev2.bin T4240RDB: rcw/t4240rdb/SSFFPPH_27_55_1_9/ rcw_27_55_1_9_1666MHz.bin T2080QDS: rcw/t2080qds/RR_PNSNNR_6C_2D/rcw_6c_2d_1800MHz.bin T1040RDB: rcw/t1040rdb/RR_P_66/rcw_1400MHz.bin T1023RDB: rcw/t1023rdb/RNSS_NPP_77/rcw_77_1400MHz.bin T1024RDB: rcw/t1024rdb/RRX_PPP_95/rcw_95_1400MHz.bin LS1043ARDB: rcw/ls1043ardb/RR_FQPP_1455/rcw_1600.bin LS1046ARDB: rcw/ls1046ardb/RR_FFSSPPPH_1133_5559/ rcw_1800_qspiboot.bin
Boot loader	u-boot-${PLATFORM}.bin
Extra bootargs*	bportals=s0 qportals=s0
Linux Kernel (For PowerPC)	uImage-${platform}.bin
Device Tree (For PowerPC)	uImage-${platform}-usdpaa.dtb
File system (For PowerPC)	fsl-image-core-${platform}.ext2.gz.u-boot
FIT (For ARM)	kernel-fsl-${platform}-usdpaa.itb

NOTE

*For more information about boot arguments, refer to USDPAA Boot Arguments.

12.3.3.3 Application and Traffic Flow

USDPAA IPSec forwarding supports two types of packet processes: encryption and decryption. Each process can work independently. The user must specify the process type for each flow. It Contains two applications:

- ipsecfwd_app: The application that takes charge of initializing devices, managing threads and buffers and packet processing
- ipsecfwd_config: The command line interface(CLI) that configures ipsecfwd, such as IP address assignments to each port and IP and ARP route management.

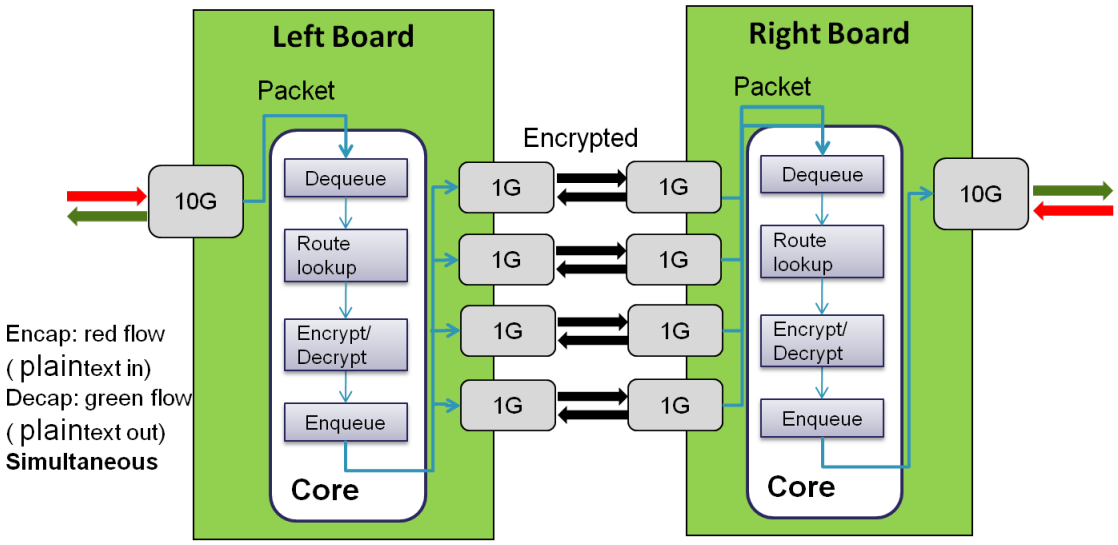
IPSec Forwarding Traffic Flow

Since Spirent TestCenter Command Sequencer (throughput measurement tool) doesn't support encrypted packets, the user must ensure that both the ingress packets to target and the egress packets from target are plain text.

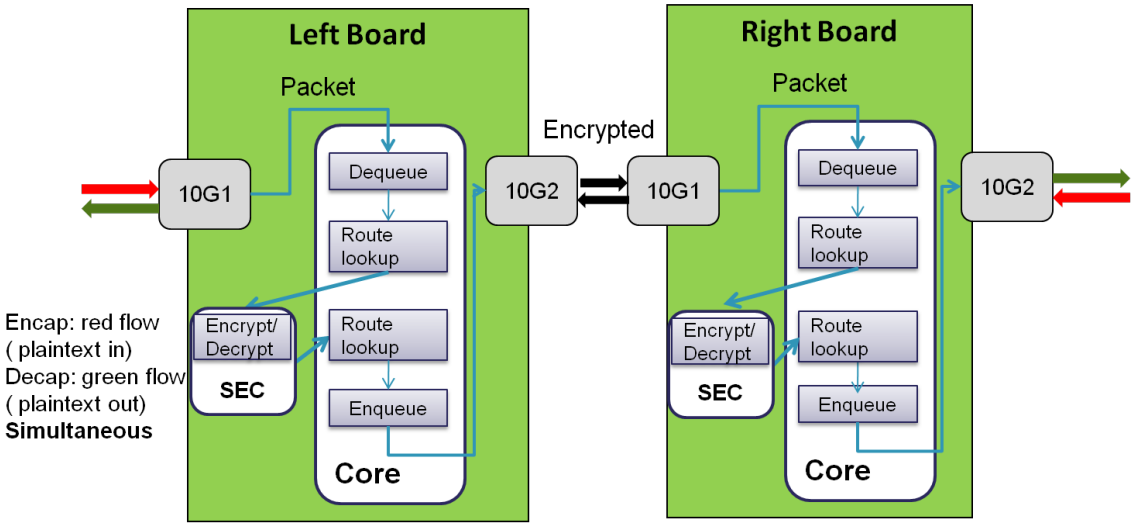
In this case, we design a traffic flow called IPSec forwarding back-to-back, wherein packets go through encryption on left target and decryption on right target in sequence, thus plaintext packets are eventually received by TestCenter.

IPSec forwarding traffic flow varies for different platforms:

On P2041RBD/P3041DS/P5020DS, 14G IPSec forwarding (aggregated throughput is 8 Gbps) is supported.



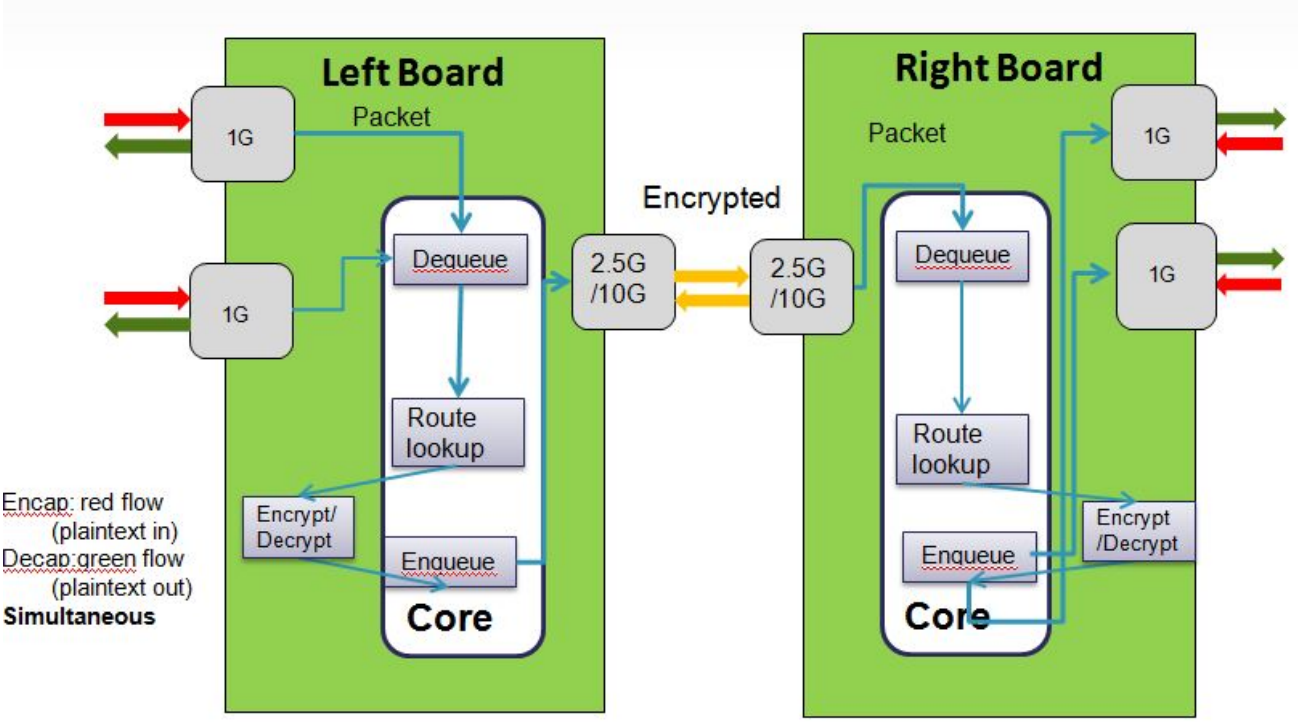
On P4080DS/P5040DS/B4860QDS/LS1046ARDB, 20G IPSec forwarding is supported; On T4240QDS/T4240RDB, 40G IPSec forwarding is supported.



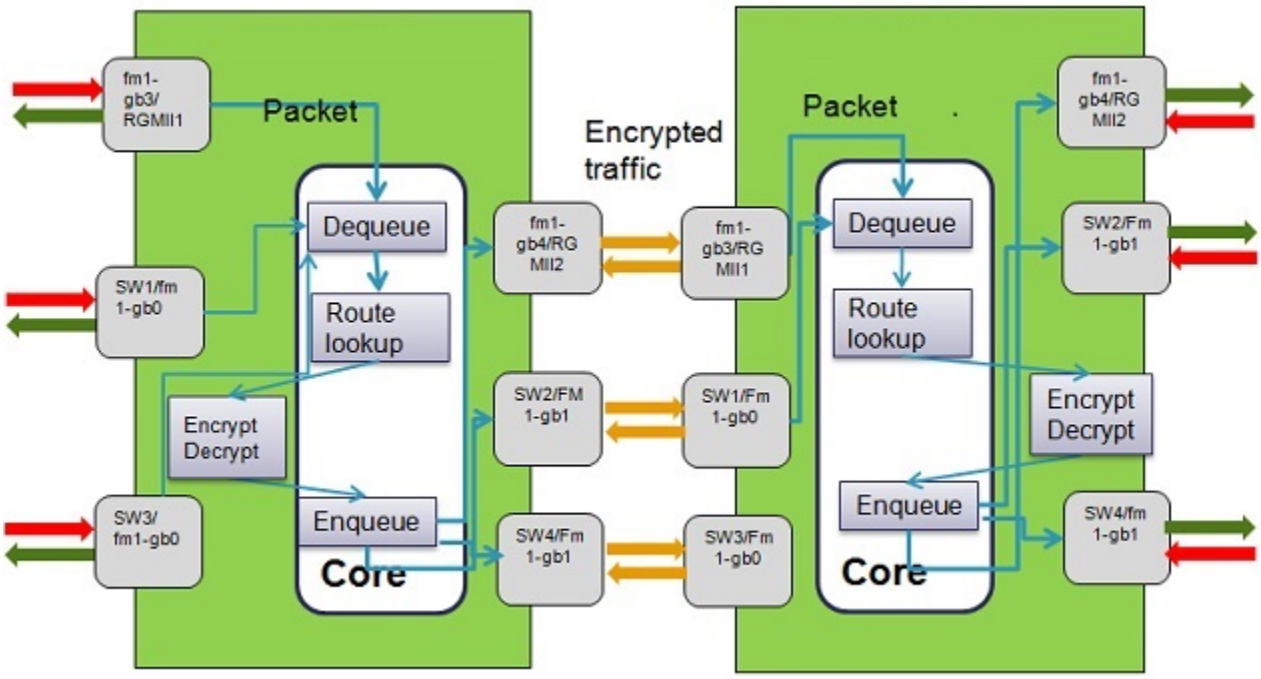
NOTE

Compared to the 20G setup, the 40G setup has two additional 10G ports on each board.

On T1023RDB/T1024RDB, 4G IPSec forwarding is supported



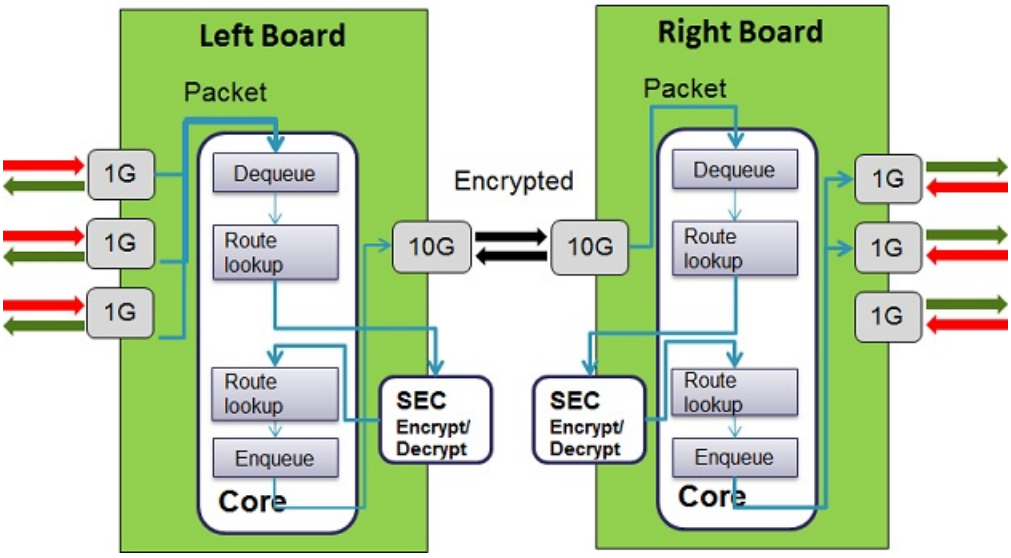
On T1040D4RDB, 6G IPSec forwarding is supported



NOTE

SW*n* is network interface on L2 switch, *n* is its index.

On LS1043ARDB, 13G IPsec forwarding is supported



DPAA Resource Configuration

Table 655. DPAA Resource Configuration for USDPAA IPSec Forwarding

DPAA Resource	USDPAA IPSec Forwarding Function

FMan	• Parsing on 32 bytes: parser result comes with every frame.
	  • Protocol parsing
	  • IP Header location
	  • IP Header checksum check
	  • Error code
	• Congestion management disabled. Fman side order restoration is on.
	• Hash is based on the SIP, DIP for encrypt; SIP, DIP, SPI for decrypt

QMan	• Push mode, Triple dequeue enable.
	• DQRR:
	  • Entry Stashing on.
	  • To L1 cache, with high priority.
	• Frame data, Annotation, Context stashing on: –Frame data = 1 line, Annotation = 1 line, Context = 1
	• Consumer Index write mode cache-inhibited: Update on every 3 frames dequeue.
	• EQRR: Access through Cache Enable Access (CENA, WIMGE=00000)

BMan	• Buffer Pool: bp9 • Buffer Size: 1728 Bytes • Number of buffers: 8192
DMA Memory	• 256MB for 32 bit platform • 256MB for 64 bit platform

12.3.3.4 IPSec Forwarding for P2041RDB/P3041DS/P5020DS

Procedure for ipsecfwd application

Assumption regarding MAC address

We assume the following regarding MAC addresses in the IPSecfwd configuration script (ipsecfwd_mix_15G.sh):

- 4 1G ports on left board: 00:e0:0c:00:e5:00/01/03/04
- 4 1G ports on right board: 00:e0:0c:00:cb:00/01/03/04

User has to change the MAC addresses in the script if the real MAC addresses are not compliant with this assumption.

If user enables promiscuous mode on these ports, this assumption could be ignored.

Step 1:

If using 10G, the user needs to set the hwconfig U-Boot variable before booting into Linux. Once hwconfig is set, boot board with usdpaa mode, then execute the following commands on board:

Table 656. Operations on board

Step Number	Action	Command on Linux Partition	Notes
1	---	`#ifconfig eth0 192.168.1.200`	Commands on Linux of **LEFT** board Ports other than eth0 can also be used
2	---	`#cd /usr/etc`	---
3	Configure PCD	`#fmc –c usdpaa_config_p2_p3_p5_14g.xml –p usdpaa_policy_hash_ipv4.xml –a`	---
4	Launch IPSec forward application which creates one thread on core 1, 2 and 3 individually	`#ipsecfwd_app <cpu_range> -c usdpaa_config_p2_p3_p5_14g.xml –p usdpaa_policy_hash_ipv4.xml –d 0x10000000`	You may use any combination of available cores the platform supports.
5	Login Linux from tftp server	`ssh root@192.168.1.200`	---
6	Run IP forward configuration script in ssh session	`# sed -i '/168.30/d' /usr/bin/ipsecfwd_mix_15G.sh` `#ipsecfwd_mix_15G.sh left $PID`	PID is the process id of ipsecfwd application. It is printed when the application starts
7	---	`# ifconfig eth0 192.168.1.100`	Commands on Linux of **RIGHT** board Ports other than eth0 can also be used
8	---	`# cd /usr/etc`	---
9	Configure PCD	`#fmc –c usdpaa_config_p2_p3_p5_14g.xml –p usdpaa_policy_hash_ipv4.xml –a`	---
10	Launch IP forward application	`#ipsecfwd_app 1..3 -c usdpaa_config_p2_p3_p5_14g.xml –p usdpaa_policy_hash_ipv4.xml –d 0x10000000`	---
11	Login Linux from tftp server	`ssh root@192.168.1.100`	---

Table continues on the next page...

Table 656. Operations on board (continued)

Step Number	Action	Command on Linux Partition	Notes
12	Run IP forward configuration script in ssh session	`# sed -i '/168.30/d' /usr/bin/ipsecfwd_mix_15G.sh` `#ipsecfwd_mix_15G.sh right $PID`	---
13	Launch TestCenter project to pump traffic	---	Now the ipsecfwd_app is ready to receive traffic

Step 2:

Create ‘Raw Stream’ in generator for each port, and compose packet as following:

- Change Mac/IP address
- Set "Count" to the appropriate number of source and destination addresses

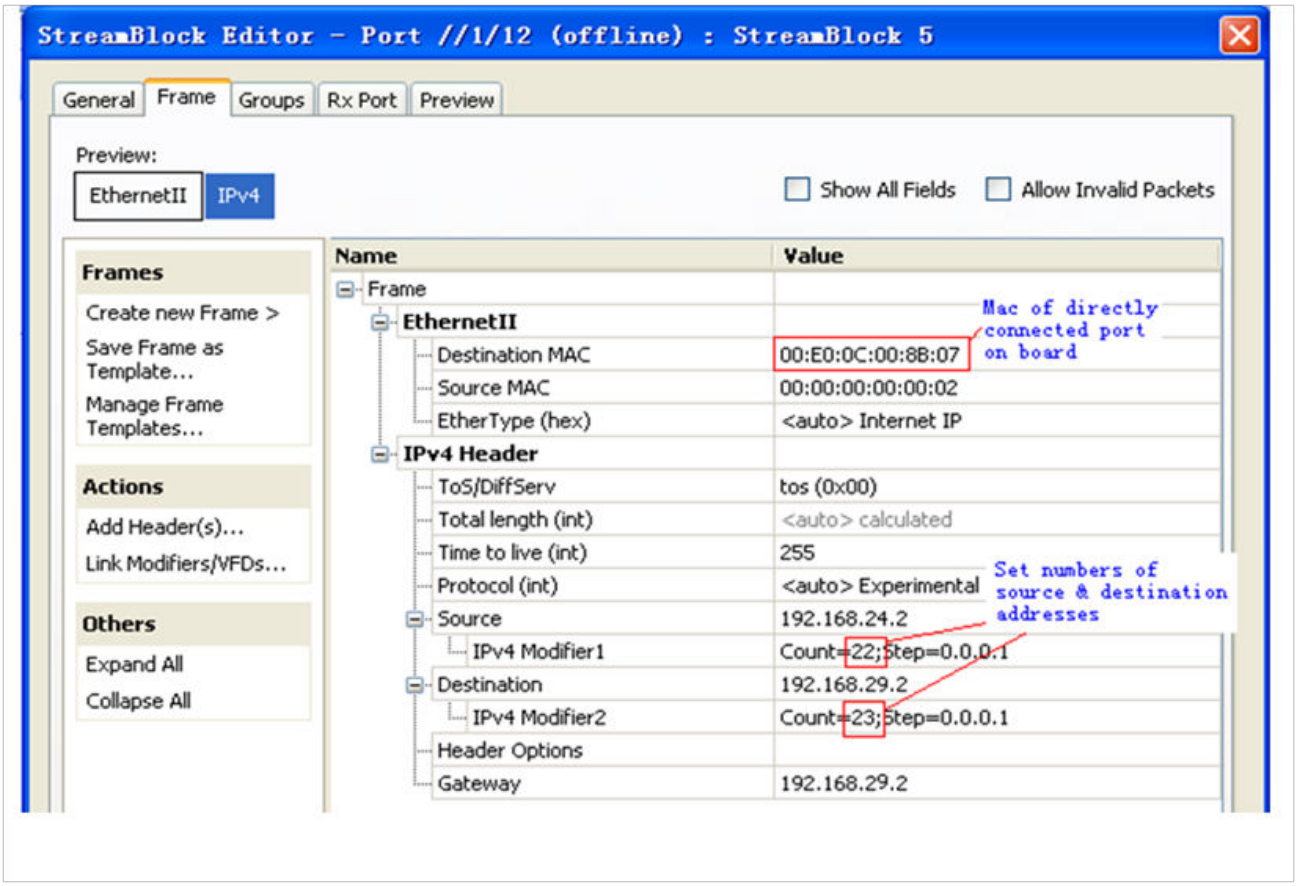
Table 657. IP addresses assignment - Left to Right (500 flows)

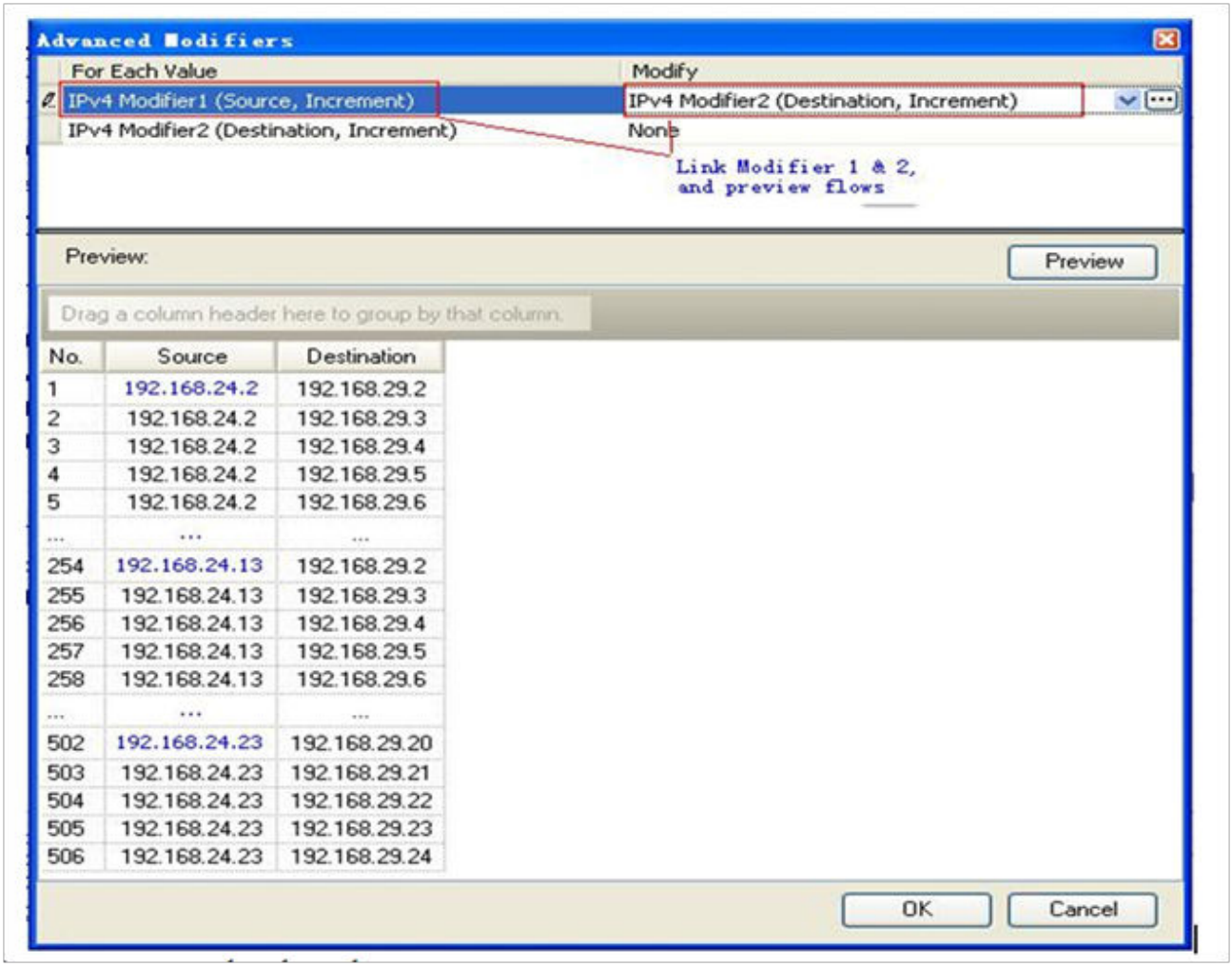
Stream #	Source Values	Destination Values
1	192.168.60.2 – 192.168.60.11	192.168.10.2 – 192.168.10.11
2	192.168.60.2 – 192.168.60.11	192.168.20.2 – 192.168.20.11
3	192.168.60.2 – 192.168.60.11	192.168.40.2 – 192.168.40.11
4	192.168.60.2 – 192.168.60.11	192.168.50.2 – 192.168.50.11

Table 658. IP addresses assignment - Right to Left (500 flows)

Stream #	Source Values	Destination Values
1	192.168.10.2 – 192.168.10.11	192.168.60.2 – 192.168.60.11
2	192.168.20.2 – 192.168.20.11	192.168.60.2 – 192.168.60.11
3	192.168.40.2 – 192.168.40.11	192.168.60.2 – 192.168.60.11
4	192.168.50.2 – 192.168.50.11	192.168.60.2 – 192.168.60.11

Total = 1000 flows


StreamBlock Editor - Port //1/12 (offline) : StreamBlock 5
General
Frame
Groups
Rx Port
Preview
Preview:
EthernetII
IPv4
Show All Fields
Allow Invalid Packets
Frames
Create new Frame >
Save Frame as Template...
Manage Frame Templates...
Actions
Add Header(s)...
Link Modifiers/VFDs...
Others
Expand All
Collapse All
Name
Value
Frame
EthernetII
Destination MAC
00:E0:0C:00:8B:07
Mac of directly connected port on board
Source MAC
00:00:00:00:00:02
EtherType (hex)
<auto> Internet IP
IPv4 Header
ToS/DiffServ
tos (0x00)
Total length (int)
<auto> calculated
Time to live (int)
255
Protocol (int)
<auto> Experimental
Source
192.168.24.2
IPv4 Modifier1
Count=22;Step=0.0.0.1
Destination
192.168.29.2
IPv4 Modifier2
Count=23;Step=0.0.0.1
Set numbers of source & destination addresses
Header Options
Gateway
192.168.29.2



Step 3:

Create Command Sequencer in TestCenter and select RFC2544 Throughput test. Then set Test Parameters for TestCenter Throughput Sequencer as shown in table below.

Table 659. Spirent TestCenter Throughput Configuration

Options	Value
Test Duration	60 seconds
Resolution	0.1%
Packet Loss Rate	0.001%
Packet Size	82, 408, 1442 bytes

Step 4:

Start TestCenter Command Sequencer to measure performance.

12.3.3.5 IPSec Forwarding for P4080DS/P5040DS/B4860QDS/ LS1046ARDB

Procedure for ipsecfwd application

Assumption regarding MAC address

We assume the following regarding MAC addresses in IPSecfwd configuration script (ipsecfwd_mix_20G.sh):

- Port 10G2 on left board: 00:e0:0c:00:8a:09
- Port 10G1 on right board: 00:e0:0c:00:e2:04

User has to change the MAC addresses in the script if the real MAC addresses are not compliant with the assumption.

If user enables promiscuous mode on these ports, this assumption could be ignored.

Step 1:

If using 10G, the user needs to set the hwconfig U-Boot variable before booting into Linux. Once hwconfig is set, boot board with usdpaa mode, then execute the following commands on board:

Table 660. Operations on board

Step Number	Action	Command on Linux Partition	Notes
1	---	`#ifconfig eth0 192.168.1.200`	Commands on Linux of **LEFT** board Ports other than eth0 can also be used
2	---	`#cd /usr/etc`	---
3	Configure PCD	`#fmc –c <fmc_config> –p usdpaa_policy_hash_ipv4.xml –a`	XML policy file can be found in file system. <fman_config> is: for P4080DS usdpaa_config_p4_serdes_0xe.xml ; P5040DS usdpaa_config_p5_serdes_0x02.xml; B4860QDS usdpaa_config_b4_serdes_0x2a_0x98.xml; LS1046ARDB usdpaa_config_ls1046.xml
4	Launch IPSec forward application which creates one thread on core 1, 2 and 3 individually	`#ipsecfwd_app 1..3 -c <fmc_config> –p usdpaa_policy_hash_ipv4.xml –d 0x10000000`	You may use any combination of available cores the platform supports.
5	Login Linux from tftp server	ssh root@192.168.1.200	---
6	Run IP forward configuration script in ssh session	`#ipsecfwd_mix_20G.sh left $PID`	PID is the process id of ipsecfwd application. It is printed when the application starts
7	---	`# ifconfig eth0 192.168.1.100`	commands on Linux of **RIGHT** board Ports other than eth0 can also be used

Table continues on the next page...

Table 660. Operations on board (continued)

Step Number	Action	Command on Linux Partition	Notes
8	---	`# cd /usr/etc`	---
9	Configure PCD	`#fmc -c <fmc_config> -p usdpaa_policy_hash_ipv4.xml -a`	See 4
10	Launch IP forward application	`#ipsecfwd_app 1..3 -c <fmc_config> -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	---
11	Login Linux from tftp server	`ssh root@192.168.1.100`	---
12	Run IP forward configuration script in ssh session	`#ipsecfwd_mix_20G.sh right $PID`	---
13	Launch TestCenter project to pump traffic	---	Now the ipsecfwd_app is ready to receive traffic

Step 2:

Create ‘Raw Stream’ in generator for each port, and compose packet as following:

- Change Mac/IP address
- Set "Count" to the appropriate number of source and destination addresses

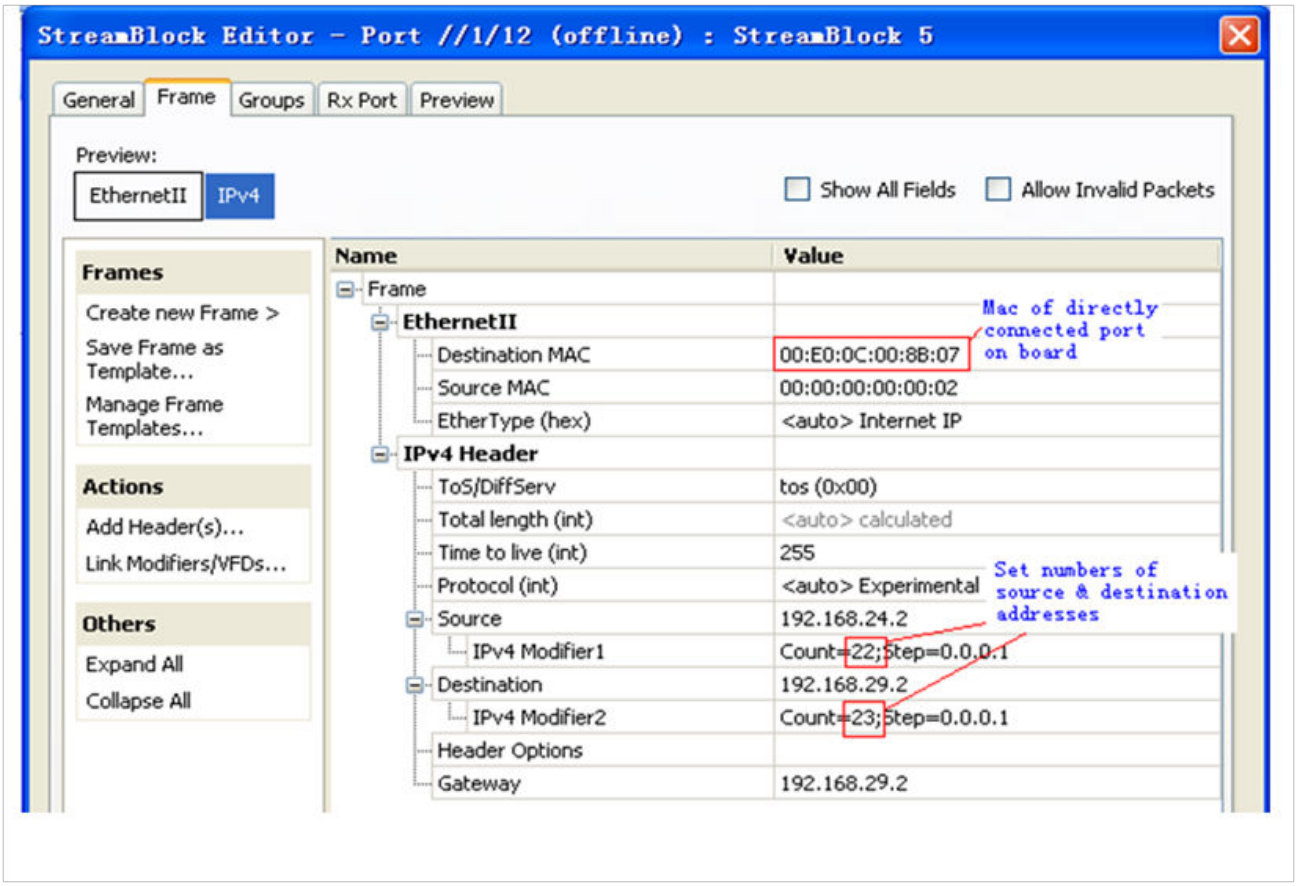
Table 661. IP addresses assignment - Left to Right (506 flows)

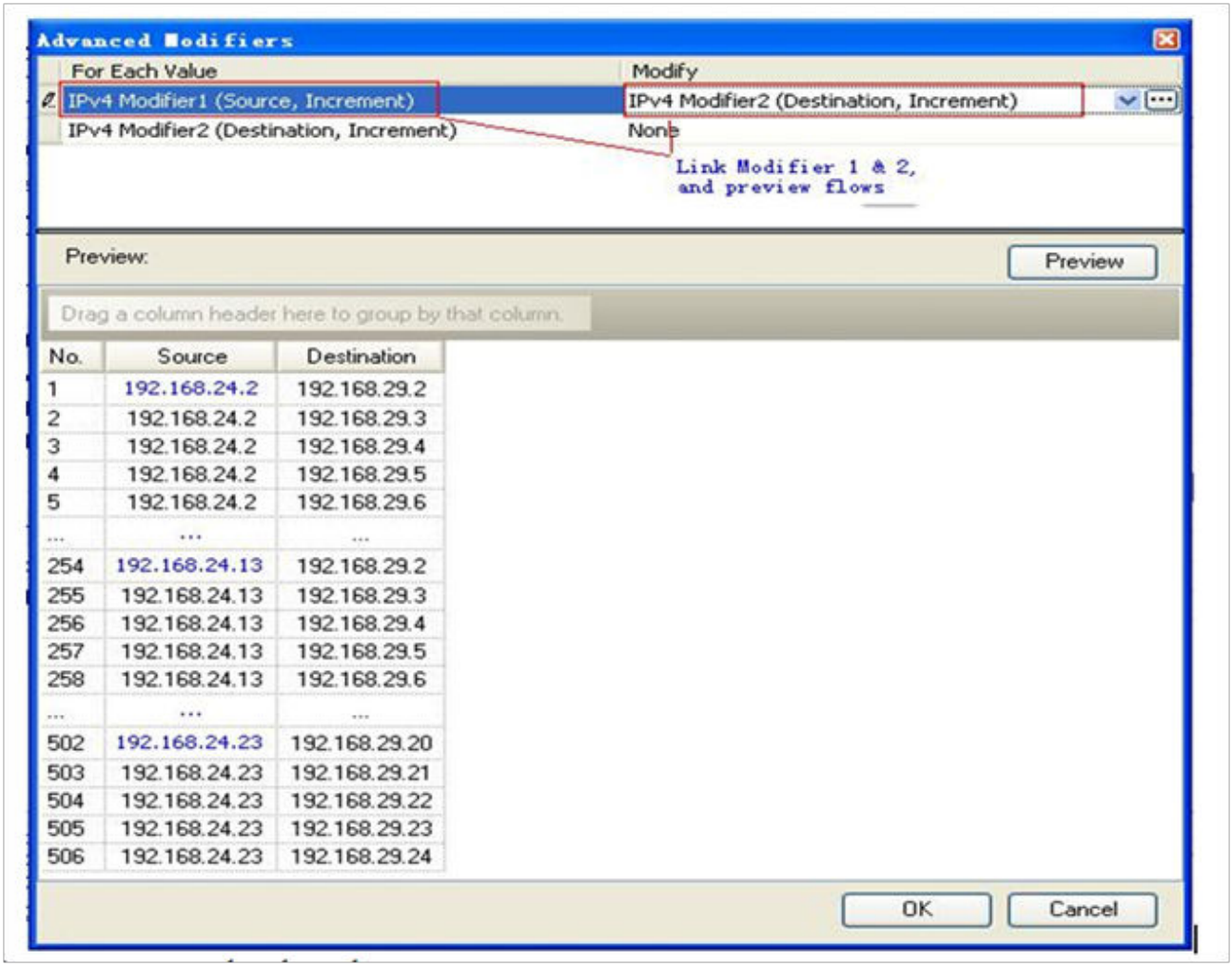
Stream #	Source Values	Destination Values
1	192.168.60.2 – 192.168.60.23	192.168.160.2 – 192.168.160.24

Table 662. IP addresses assignment - Right to Left (506 flows)

Stream #	Source Values	Destination Values
1	192.168.160.2 – 192.168.160.23	192.168.60.2 – 192.168.60.24

Total = 1012 flows


StreamBlock Editor - Port //1/12 (offline) : StreamBlock 5
General
Frame
Groups
Rx Port
Preview
Preview:
EthernetII
IPv4
Show All Fields
Allow Invalid Packets
Frames
Create new Frame >
Save Frame as Template...
Manage Frame Templates...
Actions
Add Header(s)...
Link Modifiers/VFDs...
Others
Expand All
Collapse All
Name
Value
Frame
EthernetII
Destination MAC
00:E0:0C:00:8B:07
Mac of directly connected port on board
Source MAC
00:00:00:00:00:02
EtherType (hex)
<auto> Internet IP
IPv4 Header
ToS/DiffServ
tos (0x00)
Total length (int)
<auto> calculated
Time to live (int)
255
Protocol (int)
<auto> Experimental
Source
192.168.24.2
IPv4 Modifier1
Count=22;Step=0.0.0.1
Destination
192.168.29.2
IPv4 Modifier2
Count=23;Step=0.0.0.1
Set numbers of source & destination addresses
Header Options
Gateway
192.168.29.2



Step 3:

Create Command Sequencer in TestCenter and select RFC2544 Throughput test. Then set Test Parameters for TestCenter Throughput Sequencer as shown in table below.

Table 663. Spirent TestCenter Throughput Configuration

Spirent variable	Variable Setting
Test Duration	60 seconds
Resolution	0.1%
Packet Loss Rate	0.001%
Packet Size	82, 408, 1442 bytes

Step 4:

Start TestCenter Command Sequencer to measure performance.

12.3.3.6 IPSec Forwarding for T4240QDS

Procedure for ipsecfwd application

Assumption regrading MAC address

We assume the following regarding MAC addresses in IPSecfwd configuration script (ipsecfwd_mix_40G.sh):

- Port 10G2 & 10G4 on left board: 00:e0:0c:00:8a:09/0a
- Port 10G1 & 10G3 on right board: 00:e0:0c:00:e2:04/05

User has to change the MAC addresses in the script if the real MAC addresses are not compliant with the assumption.

If user enables promiscuous mode on these ports, this assumption could be ignored.

Step 1:

If using 10G, the user needs to set the hwconfig U-Boot variable before booting into Linux. Once hwconfig is set, boot board with usdpaa mode, then execute the following commands on board:

Table 664. Operations on board

Step Number	Action	Command on Linux Partition	Notes
1	---	`#ifconfig eth0 192.168.1.200`	Commands on Linux of **LEFT** board Ports other than eth0 can also be used
2	---	`#cd /usr/etc`	---
3	Configure PCD	`#fmc -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -a`	<fman_config> for T4240QDS is usdpaa_config_t4_serdes_1_1_5_5.xml, for T4240RDB is usdpaa_config_t4_48g.xml
4	Launch IPSec forward application which creates one thread on core 1, 2 and 3 individually	`#ipsecfwd_app 1..3 -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	You may use any combination of available cores the platform supports.
5	Login Linux from tftp server	`ssh root@192.168.1.200`	---
6	Run IP forward configuration script in ssh session	`#ipsecfwd_mix_40G.sh left $PID`	PID is the process id of ipsecfwd application. It is printed when the application starts
7	---	`# ifconfig eth0 192.168.1.100`	Commands on Linux of **RIGHT** board Ports other than eth0 can also be used
8	---	`# cd /usr/etc`	---

Table continues on the next page...

Table 664. Operations on board (continued)

Step Number	Action	Command on Linux Partition	Notes
9	Configure PCD	`#fmc -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -a`	See 3
10	Launch IP forward application	`#ipsecfwd_app 1..3 -c <fman_config> -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	---
11	Login Linux from tftp server	`ssh root@192.168.1.100`	---
12	Run IP forward configuration script in ssh session	`#ipsecfwd_mix_40G.sh right $PID`	---
13	Launch TestCenter project to pump traffic	---	Now the ipsecfwd_app is ready to receive traffic

Step 2:

Create ‘Raw Stream’ in generator for each port, and compose packet as following:

- Change Mac/IP address
- Set "Count" to the appropriate number of source and destination addresses

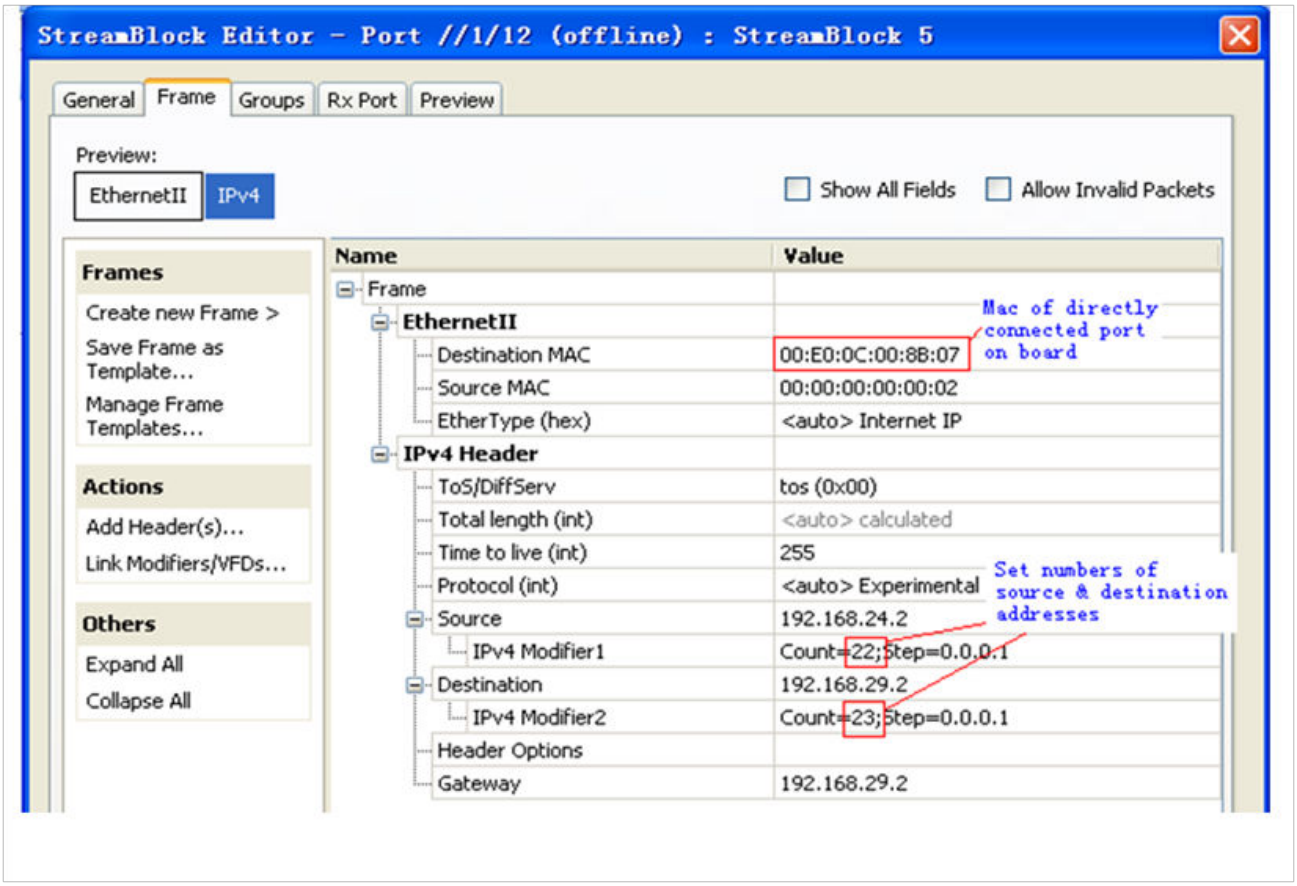
Table 665. IP addresses assignment - Left to Right (512 flows)

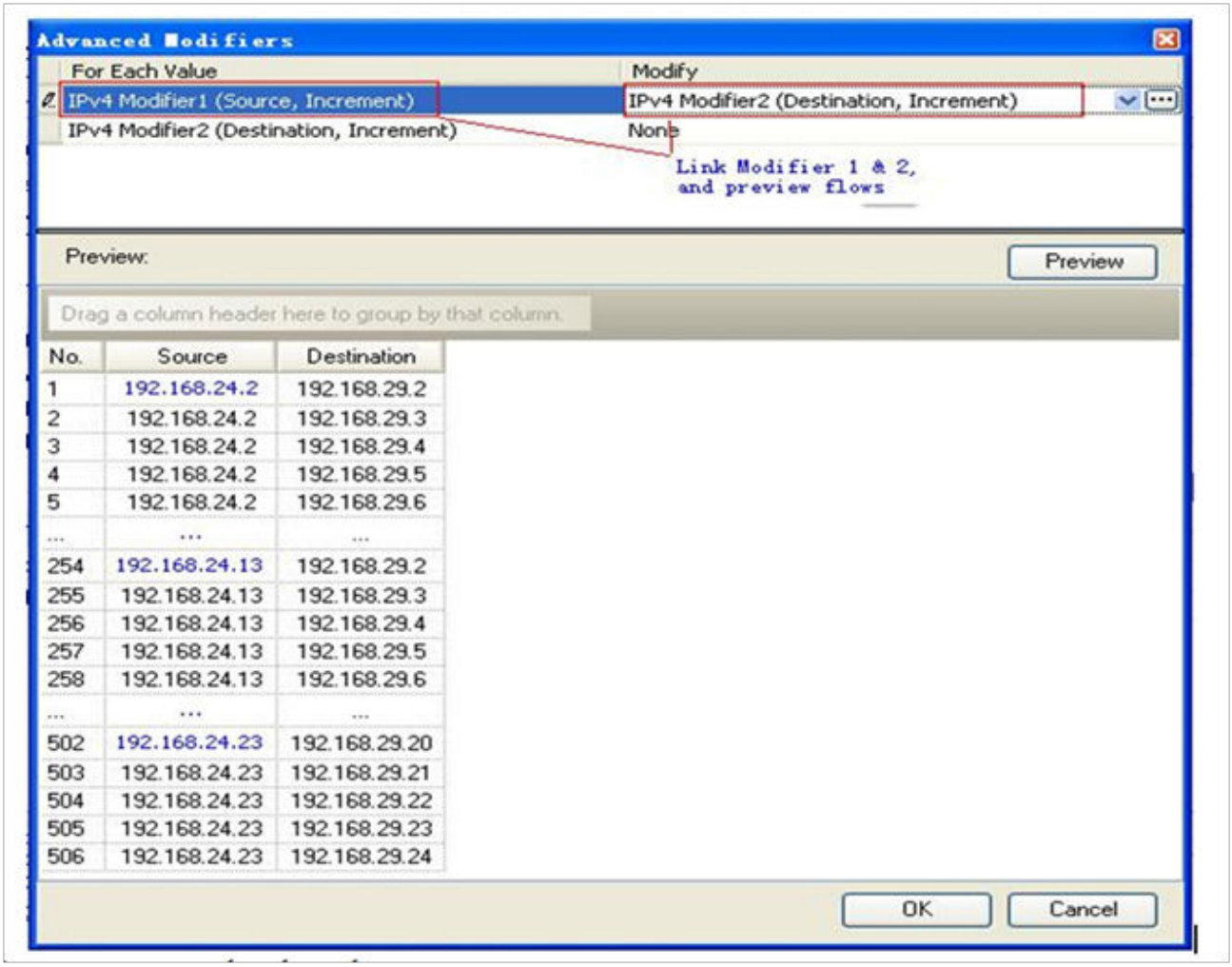
Stream #	Source Values	Destination Values
1	192.168.60.2 – 192.168.60.17	192.168.160.2 – 192.168.160.17
2	192.168.70.2 – 192.168.70.17	192.168.170.2 – 192.168.170.17

Table 666. IP addresses assignment - Right to Left (512 flows)

Stream #	Source Values	Destination Values
1	192.168.160.2 – 192.168.160.17	192.168.60.2 – 192.168.60.17
2	192.168.170.2 – 192.168.170.17	192.168.70.2 – 192.168.70.17

Total = 1024 flows


StreamBlock Editor - Port //1/12 (offline) : StreamBlock 5
General
Frame
Groups
Rx Port
Preview
Preview:
EthernetII
IPv4
Show All Fields
Allow Invalid Packets
Frames
Create new Frame >
Save Frame as Template...
Manage Frame Templates...
Actions
Add Header(s)...
Link Modifiers/VFDs...
Others
Expand All
Collapse All
Name
Value
Frame
EthernetII
Destination MAC
00:E0:0C:00:8B:07
Mac of directly connected port on board
Source MAC
00:00:00:00:00:02
EtherType (hex)
<auto> Internet IP
IPv4 Header
ToS/DiffServ
tos (0x00)
Total length (int)
<auto> calculated
Time to live (int)
255
Protocol (int)
<auto> Experimental
Source
192.168.24.2
IPv4 Modifier1
Count=22;Step=0.0.0.1
Destination
192.168.29.2
IPv4 Modifier2
Count=23;Step=0.0.0.1
Set numbers of source & destination addresses
Header Options
Gateway
192.168.29.2



Step 3:

Create Command Sequencer in TestCenter and select RFC2544 Throughput test. Then set Test Parameters for TestCenter Throughput Sequencer as shown in table below.

Table 667. Spirent TestCenter Throughput Configuration

Spirent variable	Variable Setting
Test Duration	60 seconds
Resolution	0.1%
Packet Loss Rate	0.001%
Packet Size	82, 408, 1442 bytes

Step 4:

Start TestCenter Command Sequencer to measure performance.

12.3.3.7 IPSec Forwarding for T1023RDB/T1024RDB

Procedure for ipsecfwd application

Assumption regrading MAC address

We assume the following regarding MAC addresses in IPSecfwd configuration script (ipsecfwd_t1023_mix_4.5G.sh, ipsecfwd_t1024_mix_12G.sh):

- Port fm1-mac3 and fm1-mac4 on left board: 00:e0:0c:00:71:02/03
- Port fm1-mac3 and fm1-mac4 on right board: 00:e0:0c:00:6f:02/03

Note: For T1023RDB, used fm1-mac1 instead of fm1-mac3, port id is 0

User has to change the MAC addresses in the script if the real MAC addresses are not compliant with the assumption.

If user enables promiscuous mode on these ports, this assumption could be ignored.

Step 1:

If using 10G, the user needs to set the hwconfig U-Boot variable before booting into Linux. Once hwconfig is set, boot board with usdpaa mode, then execute the following commands on board:

Table 668. Operations on board

Step Number	Action	Command on Linux Partition	Notes
1	---	`#ifconfig eth0 192.168.1.200`	Commands on Linux of **LEFT** board Ports other than eth0 can also be used
2	---	`#cd /usr/etc`	---
3	Configure PCD	`#fmc -c usdpaa_config_t1024_serdes_0x99.xml -p usdpaa_policy_hash_ipv4.xml -a`	---
4	Launch IPSec forward application which creates one thread on core 0 and 1 individually	`#ipsecfwd_app 0..1 -c usdpaa_config_t1024_serdes_0x99.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	Input 0 or 1 instead of 0..1 to launch single data processing thread on core 0 or core 1
5	Login Linux from tftp server	`ssh root@192.168.1.200`	---

Table continues on the next page...

Table 668. Operations on board (continued)

Step Number	Action	Command on Linux Partition	Notes

6	Modify IPSec forwarding configuration script	`sed -i 's/e2:1e/6F:02/g;t;s/8a:1e/71:02/g' <ipsecfwd_config_script>`
		`sed -i 's/e2:28/6F:03/g;t;s/8a:28/71:03/g' <ipsecfwd_config_script>`
		`sed -i 's/00:04:9f:01:02:05/00:e0:0c:00:6F:00/g;t;s/00:04:9f:11:12:24/00:e0:0c:00:71:00/g' <ipsecfwd_config_script>`
			<ipsecfwd_config_script> for T1023RDB is ipsecfwd_t1023_mix_4.5G.sh, for T1024RDB is ipsecfwd_t1024_mix_12G.sh.

7	Run IP forward configuration script in ssh session	`#ipsecfwd_t1024_mix_12G.sh left $PID`	PID is the process id of ipsecfwd application. It is printed when the application starts
8	---	`# ifconfig eth0 192.168.1.100`	Commands on Linux of **RIGHT** board Ports other than eth0 can also be used
9	---	`# cd /usr/etc`	---
10	Configure PCD	`#fmc -c usdpaa_config_t1024_serdes_0x99.xml -p usdpaa_policy_hash_ipv4.xml -a`	---
11	Launch IP forward application	`#ipsecfwd_app 0..1 -c usdpaa_config_t1024_serdes_0x99.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	---
12	Login Linux from tftp server	`ssh root@192.168.1.100`	---

Table continues on the next page...

Table 668. Operations on board (continued)

Step Number	Action	Command on Linux Partition	Notes

13	Modify IPSec forwarding configuration script	`sed -i 's/e2:1e/6F:02/g;t;s/8a:1e/71:02/g' <ipsecfwd_config_script>`
		`sed -i 's/e2:28/6F:03/g;t;s/8a:28/71:03/g' <ipsecfwd_config_script>`
		`sed -i 's/00:04:9f:01:02:05/00:e0:0c:00:6F:00/g;t;s/00:04:9f:11:12:24/00:e0:0c:00:71:00/g' <ipsecfwd_config_script>`
			<ipsecfwd_config_script> for T1023RDB is ipsecfwd_t1023_mix_4.5G.sh, for T1024RDB is ipsecfwd_t1024_mix_12G.sh.

14	Run IP forward configuration script in ssh session	`#ipsecfwd_t1024_mix_12G.sh right $PID`	PID is the process id of ipsecfwd application. It is printed when the application starts
15	Launch TestCenter project to pump traffic	---	Now the ipsecfwd_app is ready to receive traffic

Step 2:

Create ‘Raw Stream’ in generator for each port, and compose packet as following:

- Change Mac/IP address
- Set "Count" to the appropriate number of source and destination addresses

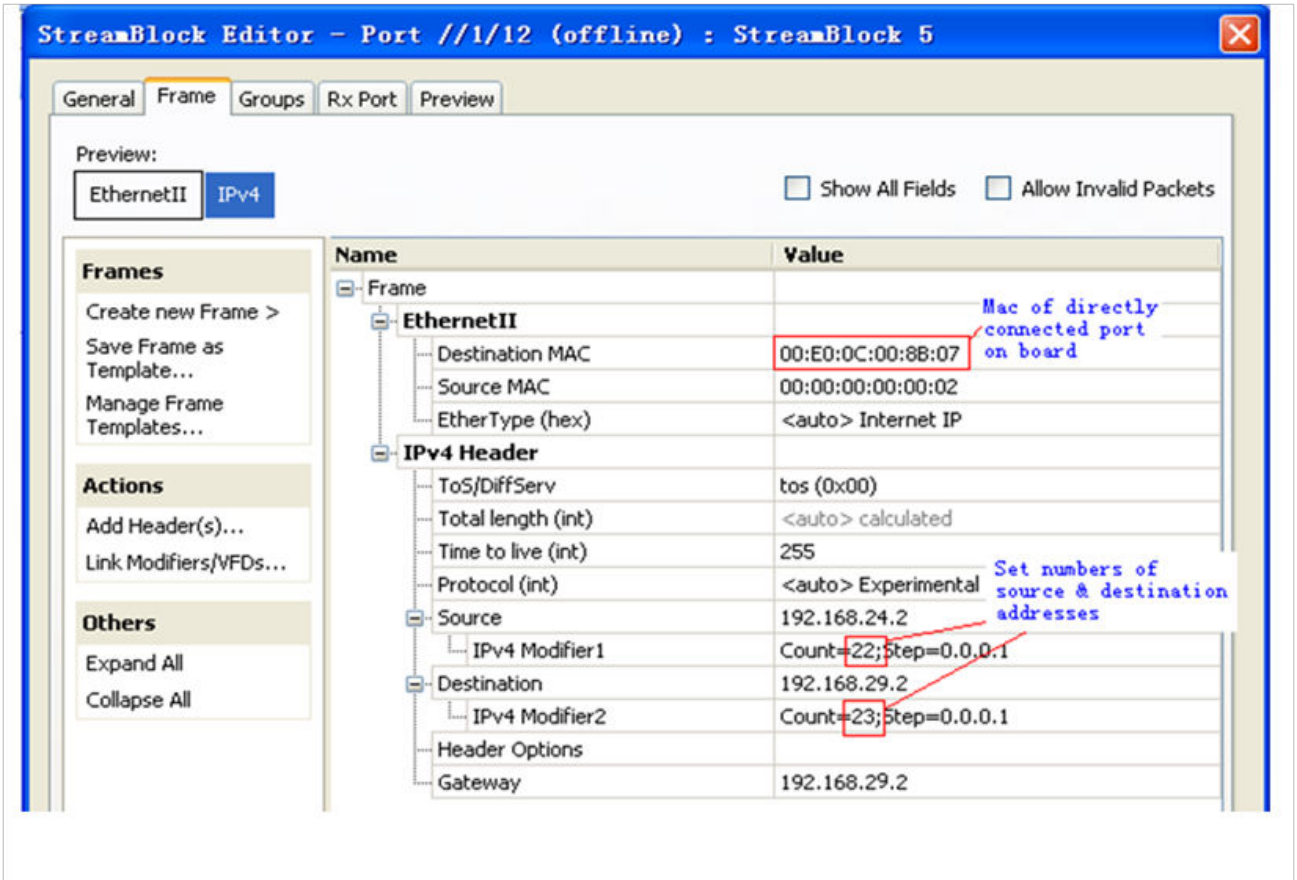
Table 669. IP addresses assignment - Left to Right (512 flows)

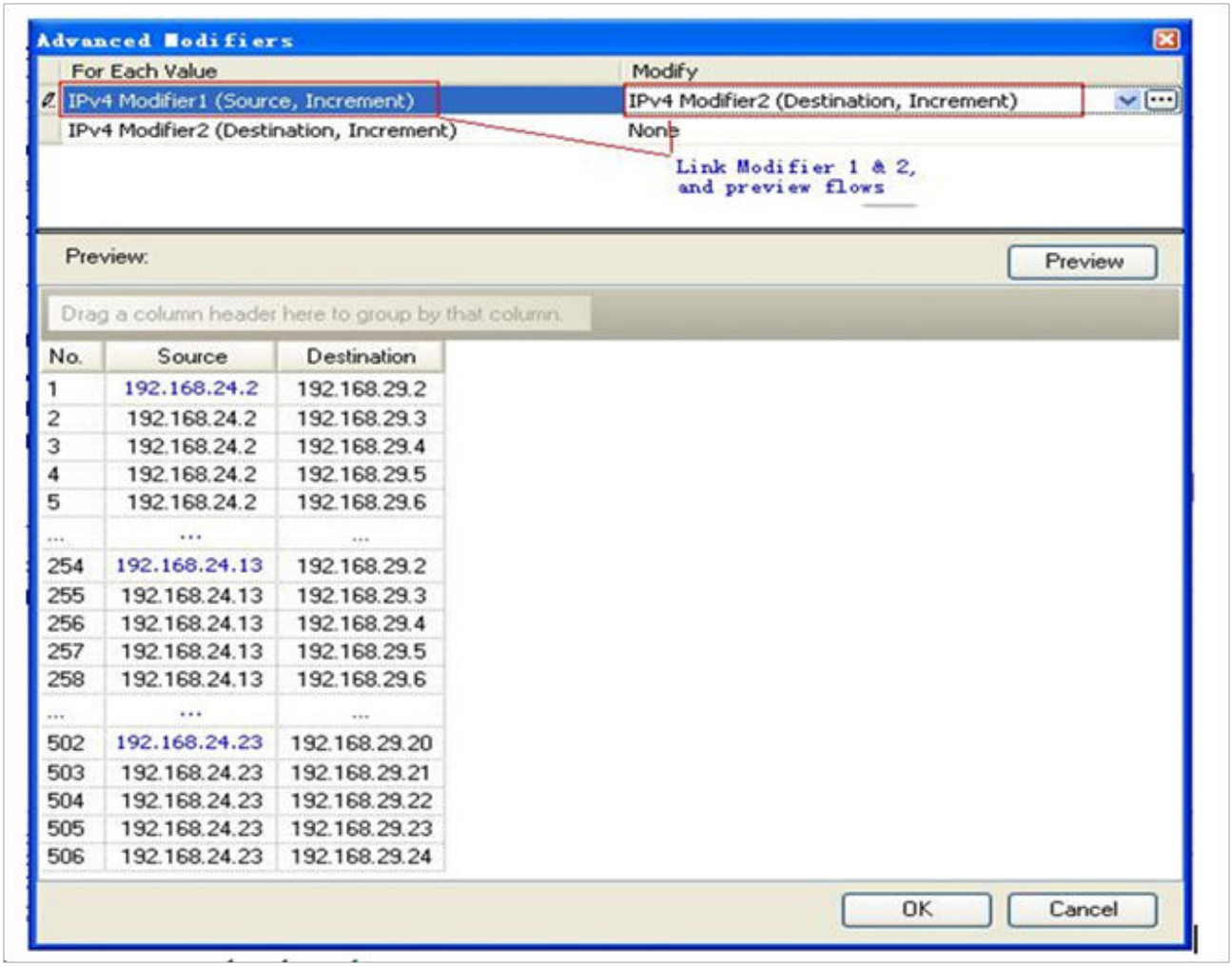
Stream #	Source Values	Destination Values
1	192.168.20.2 – 192.168.20.17	192.168.40.2 – 192.168.40.17
2	192.168.30.2 – 192.168.30.17	192.168.50.2 – 192.168.50.17

Table 670. IP addresses assignment - Right to Left (512 flows)

Stream #	Source Values	Destination Values
1	192.168.40.2 – 192.168.40.17	192.168.20.2 – 192.168.20.17
2	192.168.50.2 – 192.168.50.17	192.168.30.2 – 192.168.30.17

Total = 1024 flows


StreamBlock Editor - Port //1/12 (offline) : StreamBlock 5
General
Frame
Groups
Rx Port
Preview
Preview:
EthernetII
IPv4
Show All Fields
Allow Invalid Packets
Frames
Create new Frame >
Save Frame as Template...
Manage Frame Templates...
Actions
Add Header(s)...
Link Modifiers/VFDs...
Others
Expand All
Collapse All
Name
Value
Frame
EthernetII
Destination MAC
00:E0:0C:00:8B:07
Mac of directly connected port on board
Source MAC
00:00:00:00:00:02
EtherType (hex)
<auto> Internet IP
IPv4 Header
ToS/DiffServ
tos (0x00)
Total length (int)
<auto> calculated
Time to live (int)
255
Protocol (int)
<auto> Experimental
Source
192.168.24.2
IPv4 Modifier1
Count=22;Step=0.0.0.1
Destination
192.168.29.2
IPv4 Modifier2
Count=23;Step=0.0.0.1
Set numbers of source & destination addresses
Header Options
Gateway
192.168.29.2



Step 3:

Create Command Sequencer in TestCenter and select RFC2544 Throughput test. Then set Test Parameters for TestCenter Throughput Sequencer as shown in table below.

Table 671. Spirent TestCenter Throughput Configuration

Spirent variable	Variable Setting
Test Duration	60 seconds
Resolution	0.1%
Packet Loss Rate	0.001%
Packet Size	82, 408, 1442 bytes

Step 4:

Start TestCenter Command Sequencer to measure performance.

12.3.3.8 IPSec Forwarding for T1040D4RDB

Procedure for ipsecfwd application

Assumption regrading MAC address

We assume the following regarding MAC addresses in IPSecfwd configuration script (ipsecfwd_mix_40G.sh):

- Port 1G1, 1G2, 1G4 & 1G5 on left board: 00:e0:0c:00:8a:00/01/03/04
- Port 1G1, 1G2, 1G4 & 1G5 on right board: 00:e0:0c:00:e2:00/01/03/04

User has to change the MAC addresses in the script if the real MAC addresses are not compliant with the assumption.

If user enables promiscuous mode on these ports, this assumption could be ignored.

Step 1:

Configure FDB entries for L2 switch through application 'l2sw_bin'

```
Left board:
# l2sw_bin
l2sw_bin> mac add 00:11:22:33:44:01 0
l2sw_bin> mac add 00:11:22:33:55:01 1
l2sw_bin> mac add 00:11:22:33:44:02 2
l2sw_bin> mac add 00:11:22:33:55:02 3
l2sw_bin> mac add 00:e0:0c:00:8a:00 8
l2sw_bin> mac add 00:e0:0c:00:e2:01 9
l2sw_bin> mac add 00:11:22:33:22:01 9
l2sw_bin> mac add 00:11:22:33:22:02 9
l2sw_bin> Press Ctrl+C to quit l2sw_bin application

Right board:
l2sw_bin> mac add 00:e0:0c:00:e2:00 8
l2sw_bin> mac add 00:e0:0c:00:e2:01 9
l2sw_bin> mac add 00:11:22:33:55:01 8
l2sw_bin> mac add 00:11:22:33:55:02 8
l2sw_bin> mac add 00:11:22:33:22:01 0
l2sw_bin> mac add 00:11:22:33:33:01 1
l2sw_bin> mac add 00:11:22:33:22:02 2
l2sw_bin> mac add 00:11:22:33:33:02 3
l2sw_bin> Press Ctrl+C to quit l2sw_bin application
```

Step 2:

Create a Shell script 'ipsecfwd_mix_6G.sh' on both boards with following content

```
pid=$2
if [ "$pid" == "" ]
    then
        echo "Give PID to hook up with"
        exit 1
fi

ipsecfwd_config -P $pid -F -a 192.168.60.1 -i 0
ipsecfwd_config -P $pid -F -a 192.168.70.1 -i 3
ipsecfwd_config -P $pid -F -a 192.168.160.1 -i 1
ipsecfwd_config -P $pid -F -a 192.168.170.1 -i 4

if [ "$1" == "left" ]
then
    i=2
    while [ "$i" -le 14 ]
    do
#set the mac address of the right board here for creating ARP entry
```

```
      ipsecfwd_config -P $pid -G -s 192.168.70.$i -m 00:e0:0c:00:2a:05 -r true
      ipsecfwd_config -P $pid -G -s 192.168.170.$i -m 00:e0:0c:00:e2:03 -r true
      i=`expr $i + 1`
    done

    i=2
    while [ "$i" -le 19 ]
    do
        if [ "$i" -le 10 ]
        then
            ipsecfwd_config -P $pid -G -s 192.168.60.$i -m 00:11:22:33:44:01 -r true
            ipsecfwd_config -P $pid -G -s 192.168.160.$i -m 00:11:22:33:55:01 -r true
        else
            ipsecfwd_config -P $pid -G -s 192.168.60.$i -m 00:11:22:33:44:02 -r true
            ipsecfwd_config -P $pid -G -s 192.168.160.$i -m 00:11:22:33:55:02 -r true
        fi
        i=`expr $i + 1`
    done

    f=out
    s=in
fi

if [ "$1" == "right" ]
then
    i=2
    while [ "$i" -le 14 ]
    do
#set the mac address of the left board here for creating ARP entry
      ipsecfwd_config -P $pid -G -s 192.168.70.$i -m 00:e0:0c:00:8a:04 -r true
      ipsecfwd_config -P $pid -G -s 192.168.170.$i -m 00:e0:0c:00:4b:05 -r true
      i=`expr $i + 1`
    done

    i=2
    while [ "$i" -le 19 ]
    do
        if [ "$i" -le 10 ]
        then
            ipsecfwd_config -P $pid -G -s 192.168.60.$i -m 00:11:22:33:22:01 -r true
            ipsecfwd_config -P $pid -G -s 192.168.160.$i -m 00:11:22:33:33:01 -r true
        else
            ipsecfwd_config -P $pid -G -s 192.168.60.$i -m 00:11:22:33:22:02 -r true
            ipsecfwd_config -P $pid -G -s 192.168.160.$i -m 00:11:22:33:33:02 -r true
        fi
        i=`expr $i + 1`
    done

    f=in
    s=out
fi


if [ "$f" == "" ]
then
    exit
fi

w=1
i=2
```

```
while [ "$i" -le 20 ]
do
    j=2
    while [ "$j" -le 19 ]
    do
        ipsecfwd_config -P $pid -A -s  192.168.60.$i -d  192.168.160.$j \
            -g 192.168.60.$i -G 192.168.160.$j -i $w -r $f
        w=`expr $w + 1`
        ipsecfwd_config -P $pid -A -s  192.168.160.$i -d  192.168.60.$j \
            -g 192.168.160.$i -G 192.168.60.$j -i $w -r $s
        w=`expr $w + 1`

        j=`expr $j + 1`
    done
    i=`expr $i + 1`
done

i=2
while [ "$i" -le 14 ]
do
    j=2
    while [ "$j" -le 14 ]
    do
        ipsecfwd_config -P $pid -A -s  192.168.70.$i -d  192.168.170.$j \
            -g 192.168.70.2 -G 192.168.170.2 -i $w -r $f
        w=`expr $w + 1`
        ipsecfwd_config -P $pid -A -s  192.168.170.$i -d  192.168.70.$j \
            -g 192.168.170.2 -G 192.168.70.2 -i $w -r $s
        w=`expr $w + 1`

        j=`expr $j + 1`
    done
    i=`expr $i + 1`
done
echo IPSecfwd CP initialization complete
```

Step 3:

Boot board with usdpaa mode, then execute the following commands on board:

Table 672. Operations on board

Step Number	Action	Command on Linux Partition	Notes
1	---	`#ifconfig eth0 192.168.1.200`	Commands on Linux of **LEFT** board Ports other than eth0 can also be used
2	---	`#cd /usr/etc`	---
3	Configure PCD	`#fmc -c usdpaa_config_t1_serdes_0x66.xml -p usdpaa_policy_hash_ipv4.xml -a`	---

Table continues on the next page...

Table 672. Operations on board (continued)

Step Number	Action	Command on Linux Partition	Notes
4	Launch IPSec forward application which creates one thread on core 1, 2 and 3 individually	`#ipsecfwd_app 1..3 -c usdpaa_config_t1_serdes_0x66.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	You may use any combination of available cores the platform supports.
5	Set fm1-gb1 to promiscuous mode under prompt of ipsecfwd_app	> promisc enable f:0 p:1	---
6	Login Linux from tftp server	`ssh root@192.168.1.200`	---
7	Run IP forward configuration script in ssh session	`#ipsecfwd_mix_6G.sh left $PID`	PID is the process id of ipsecfwd application. It is printed when the application starts
8	---	`# ifconfig eth0 192.168.1.100`	Commands on Linux of **RIGHT** board Ports other than eth0 can also be used
9	---	`# cd /usr/etc`	---
10	Configure PCD	`#fmc -c usdpaa_config_t1_serdes_0x66.xml -p usdpaa_policy_hash_ipv4.xml -a`	See 4
11	Launch IP forward application	`#ipsecfwd_app 1..3 -c usdpaa_config_t1_serdes_0x66.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000`	---
12	Set fm1-gb0 to promiscuous mode under prompt of ipsecfwd_app	> promisc enable f:0 p:0	---
13	Login Linux from tftp server	`ssh root@192.168.1.100`	---
14	Run IP forward configuration script in ssh session	`#ipsecfwd_mix_6G_t1040.sh right $PID`	---
15	Launch TestCenter project to pump traffic	---	Now the ipsecfwd_app is ready to receive traffic

Step 4:

Create ‘Raw Stream’ in generator for each port, and compose packet as following:

- Change Mac/IP address
- Set "Count" to the appropriate number of source and destination addresses

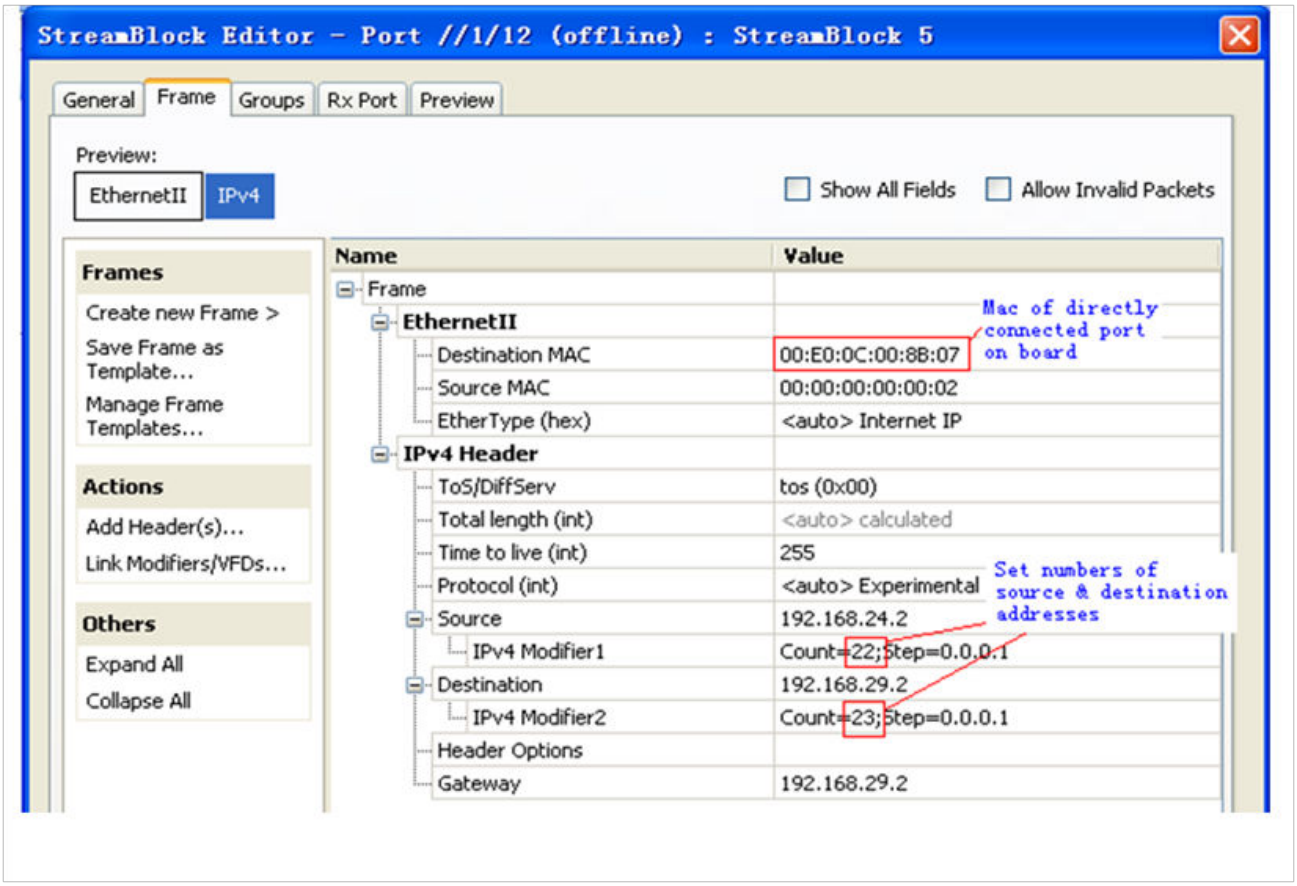
Table 673. IP addresses assignment - Left to Right (511 flows)

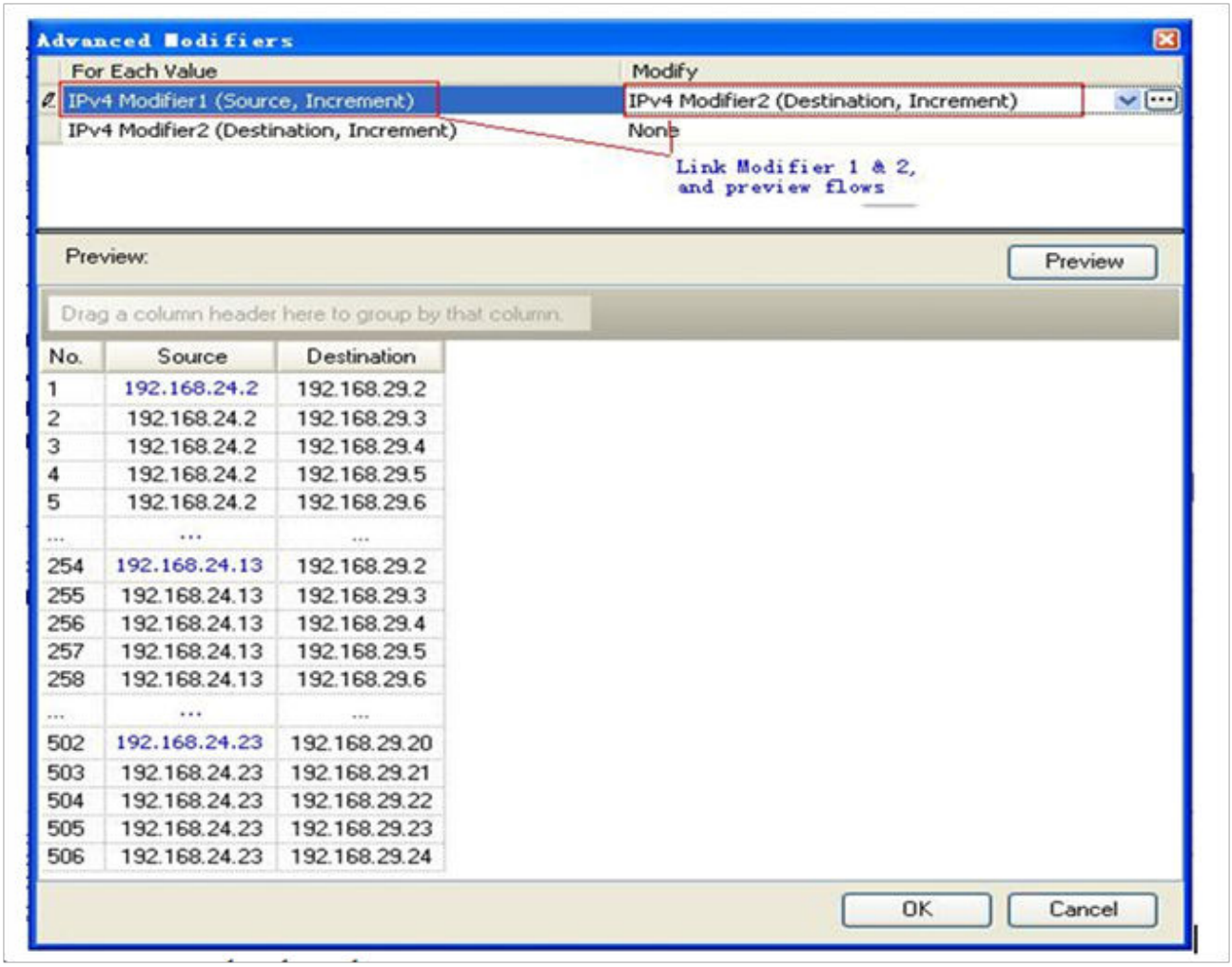
Stream #	Ingress -> Egress Ports	Source Values	Destination Values
1	fm1-gb3(L) -> fm1-gb4(R)	192.168.70.2 – 192.168.70.14	192.168.170.2 – 192.168.170.14
2	SW1(L) -> SW2(R)	192.168.60.2 – 192.168.60.20	192.168.160.2 – 192.168.160.10
3	SW3(L) -> SW4(R)	192.168.60.2 – 192.168.60.20	192.168.160.11 – 192.168.160.19

Table 674. IP addresses assignment - Right to Left (511 flows)

Stream #	Ingress -> Egress Port	Source Values	Destination Values
1	fm1-gb4(R) -> fm1-gb3(L)	192.168.170.2 – 192.168.170.14	192.168.70.2 – 192.168.70.14
2	SW2(R) -> SW1(L)	192.168.160.2 – 192.168.160.20	192.168.60.2 – 192.168.60.10
3	SW4(R) -> SW3(L)	192.168.160.2 – 192.168.160.20	192.168.60.11– 192.168.60.19

Total = 1022 flows


StreamBlock Editor - Port //1/12 (offline) : StreamBlock 5
General
Frame
Groups
Rx Port
Preview
Preview:
EthernetII
IPv4
Show All Fields
Allow Invalid Packets
Frames
Create new Frame >
Save Frame as Template...
Manage Frame Templates...
Actions
Add Header(s)...
Link Modifiers/VFDs...
Others
Expand All
Collapse All
Name
Value
Frame
EthernetII
Destination MAC
00:E0:0C:00:8B:07
Mac of directly connected port on board
Source MAC
00:00:00:00:00:02
EtherType (hex)
<auto> Internet IP
IPv4 Header
ToS/DiffServ
tos (0x00)
Total length (int)
<auto> calculated
Time to live (int)
255
Protocol (int)
<auto> Experimental
Source
192.168.24.2
IPv4 Modifier1
Count=22;Step=0.0.0.1
Destination
192.168.29.2
IPv4 Modifier2
Count=23;Step=0.0.0.1
Set numbers of source & destination addresses
Header Options
Gateway
192.168.29.2



Step 5:

Create Command Sequencer in TestCenter and select RFC2544 Throughput test. Then set Test Parameters for TestCenter Throughput Sequencer as shown in table below.

Table 675. Spirent TestCenter Throughput Configuration

Spirent variable	Variable Setting
Test Duration	60 seconds
Resolution	0.1%
Packet Loss Rate	0.001%
Packet Size	82, 408, 1442 bytes

Step 6:

Start TestCenter Command Sequencer to measure performance.

12.3.3.9 IPSec Forwarding for LS1043

Procedure for ipsecfwd application

Assumption regarding MAC address

We assume the following regarding MAC addresses in IPSecfwd configuration script (ipsecfwd_ls1043a_mix_13G.sh):

- Port 10G on left board: 00:04:9f:01:02:05
- Port 10G on right board: 00:04:9f:11:12:24

User has to change the MAC addresses in the script if the real MAC addresses are not compliant with the assumption.

If user enables promiscuous mode on these ports, this assumption could be ignored.

Step 1:

If using 10G, the user needs to set the hwconfig U-Boot variable before booting into Linux. Once hwconfig is set, boot board with usdpaa mode, then execute the following commands on board:

Table 676. Operations on board

Step Number	Action	Command on Linux Partition	Notes
1	---	`#ifconfig eth0 192.168.2.65`	Commands on Linux of **LEFT** board
2	---	`#cd /usr/etc`	---
3	Configure PCD	`#fmc –c usdpaa_config_ls1043.xml –p usdpaa_policy_hash_ipv4.xml –a`	XML policy file and config can be found in file system.
4	Launch IPSec forward application which creates one thread on core 0, 1, 2 and 3 individually	`#ipsecfwd_app 0..3 -c usdpaa_config_ls1043.xml –p usdpaa_policy_hash_ipv4.xml`	You may use any combination of available cores the platform supports.
5	Login to the board using ssh	ssh root@192.168.2.65	---
6	Run IPsec forward configuration script in ssh session	`#ipsecfwd_ls1043a_mix_13G.sh left $PID`	PID is the process id of ipsecfwd application. It is printed when the application starts
7	---	`#ifconfig eth0 192.168.2.65`	commands on Linux of **RIGHT** board Ports other than eth0 can also be used
8	---	`# cd /usr/etc`	---
9	Configure PCD	`#fmc –c usdpaa_config_ls1043.xml –p usdpaa_policy_hash_ipv4.xml –a`	See 3

Table continues on the next page...

Table 676. Operations on board (continued)

Step Number	Action	Command on Linux Partition	Notes
10	Launch IPSec forward application	`#ipsecfwd_app 0..3 -c usdpaa_config_ls1043.xml -p usdpaa_policy_hash_ipv4.xml`	---
11	Login Linux from tftp server	`ssh root@192.168.2.65`	---
12	Run IPSec forward configuration script in ssh session	`#ipsecfwd_ls1043a_mix_13G.sh right $PID`	---
13	Launch TestCenter project to pump traffic	---	Now the ipsecfwd_app is ready to receive traffic

Step 2:

Create ‘Raw Stream’ in generator for each port, and compose packet as following:

- Change Mac/IP address
- Set "Count" to the appropriate number of source and destination addresses

Table 677. IP addresses assignment - Left to Right (507 flows:169*3)

Stream #	Source Values	Destination Values
1	192.168.10.2 – 192.168.10.14	192.168.20.2 - 192.168.20.14
2	192.168.30.2 – 192.168.40.14	192.168.40.2 - 192.168.30.14
3	192.168.50.2 – 192.168.60.14	192.168.60.2 - 192.168.50.14

Table 678. IP addresses assignment - Right to Left (507 flows: 169*3)

Stream #	Source Values	Destination Values
1	192.168.20.2 - 192.168.20.14	192.168.10.2 – 192.168.10.14
2	192.168.40.2 - 192.168.30.14	192.168.30.2 – 192.168.40.14
3	192.168.60.2 - 192.168.50.14	192.168.50.2 – 192.168.60.14

Total = 1014 flows

Step 3:

Create Command Sequencer in TestCenter and select RFC2544 Throughput test. Then set Test Parameters for TestCenter Throughput Sequencer as shown in table below.

Table 679. Spirent TestCenter Throughput Configuration

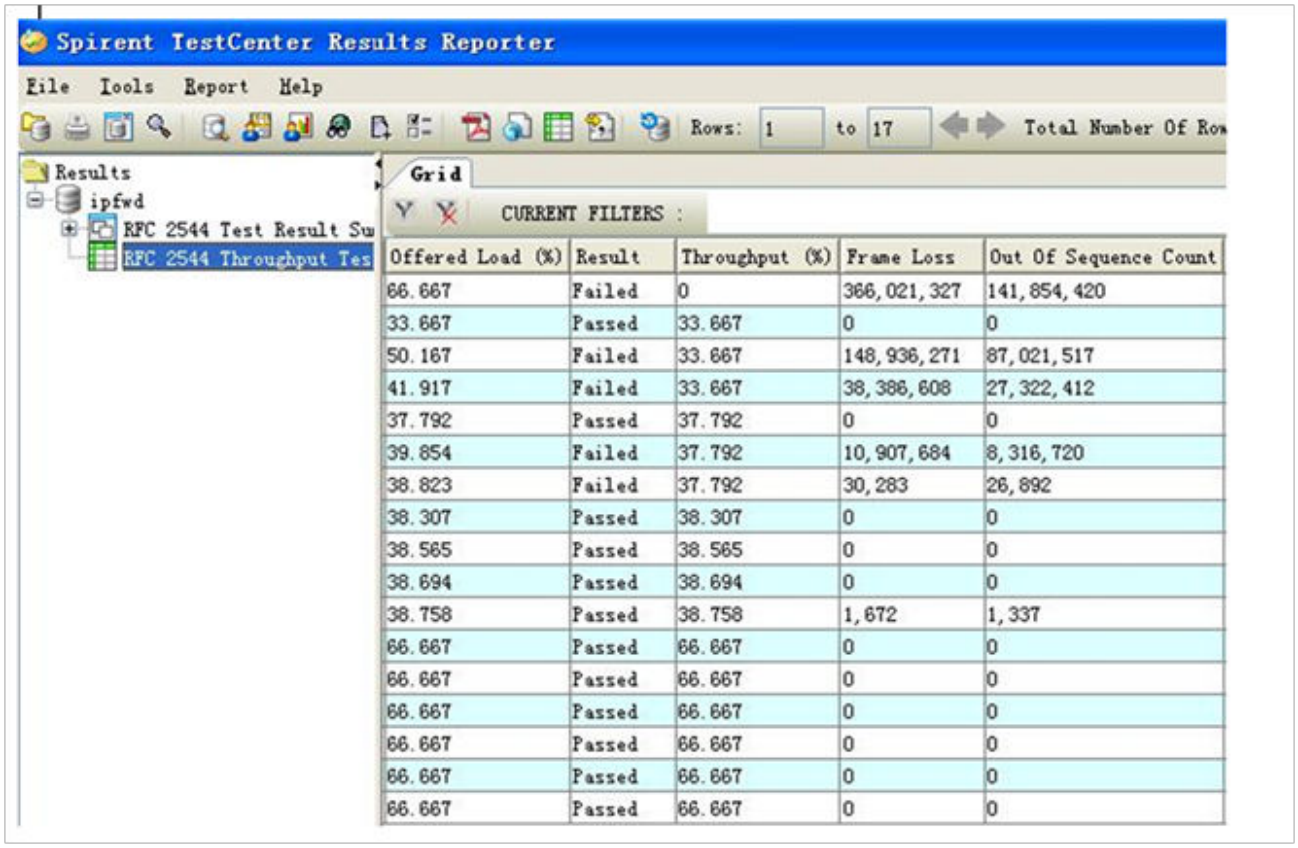
Spirent variable	Variable Setting
Test Duration	60 seconds
Resolution	0.1%
Packet Loss Rate	0.001%
Packet Size	82, 408, 1442 bytes

Step 4:

Start TestCenter Command Sequencer to measure performance.

12.3.3.10 View of Test Result

When measuring performance using TestCenter RFC 2544 Command Sequence, the result summary can be viewed using the Spirent TestCenter Results Reporter. To prevent any "out of sequence" packets when measuring with Zero Loss Throughput, enable Order Preservation or Order Restoration.



Offered Load (%)	Result	Throughput (%)	Frame Loss	Out Of Sequence Count
66.667	Failed	0	366,021,327	141,854,420
33.667	Passed	33.667	0	0
50.167	Failed	33.667	148,936,271	87,021,517
41.917	Failed	33.667	38,386,608	27,322,412
37.792	Passed	37.792	0	0
39.854	Failed	37.792	10,907,684	8,316,720
38.823	Failed	37.792	30,283	26,892
38.307	Passed	38.307	0	0
38.565	Passed	38.565	0	0
38.694	Passed	38.694	0	0
38.758	Passed	38.758	1,672	1,337
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0

12.3.4 RMan

This document describes the steps to verify performance of USDPAA RMan packet forwarding by use of the USDPAA NXP RMan Application (FRA). Follow this document to reproduce the performance results.

12.3.4.1 Platform Identification

Hardware Platform Identification

Depending on the platform, the application uses the setup shown in the table below.

Table 680. Devices

Platform	Platform Rev.	Silicon Rev.	Default frequencies	SerDes Protocol	Configuration Name
P3041DS	Rev X1	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 1333 MHz • FMan - 583 MHz	0x33	RR_HXAPNRP_0x33
P5020DS	Rev X7	Rev 2.0	• CPU - 2000 MHz • CCB - 800 MHz • DDR - 1333 MHz • FMan - 600 MHz	0x33	RR_HXAPNRP_0x33
B4860QDS	Rev B2	Rev 2.1	• CPU - 1600 MHz • CCB - 667 MHz • DDR - 1867 MHz • FMan - 667 MHz	0x2A_0x8D	N_RSSS_0x2A_0x8D
T4240QDS	Rev X4	Rev 2.0	• CPU - 1667 MHz • CCB - 733 MHz • DDR - 1866 MHz • FMan - 733 MHz	1_1_5_5	RR_XXXXPRPR_1_1_5_5

Network Simulator: Spirent TestCenter Performance Analysis System

Other traffic generators may be used, but the procedures provided are specifically for Spirent Chassis ver. 4.24

Software Platform Identification: Target Development System Software

Table 681. Software Platform Identification

System	Image file name/Description
FMan ucode	fsl_fman_ucode_<soc>_<silicon_rev>_<ucode_version>.bin Please refer to SDK document to use correct ucode version

Table continues on the next page...

Table 681. Software Platform Identification (continued)

System	Image file name/Description
RCW	P3041: rcw/RR_HXAPNRP_0x33/rcw_12g_1500mhz.bin P5020: rcw/RR_HXAPNRP_0x33/rcw_12g_2000mhz.bin B4860: rcw/N_RSSS_0x2A_0x8D/ rcw_4sgmii_4srio_2xfi_1600mhz_1866ddr.bin T4240: rcw/RR_XXXXPRPR_1_1_5_5/rcw_1_1_5_5_1666MHz_rev2.bin
Boot loader	u-boot-${PLATFORM}.bin
Extra bootargs	bportals=s0 qportals=s0
Linux Kernel	uImage-${platform}.bin
Device Tree	uImage-${platform}-usdpaa.dtb
File system	fsl-image-core-${platform}.ext2.gz.u-boot

12.3.4.2 Application configuration

Software Configuration

Table 682. Software configuration

Application	Description

Linux kernel change	The SDK includes source code for Linux. 'It is necessary to enable the RapidIO UIO driver and RapidIO Message Manager driver in the kernel module configuration.
	`Device Drivers  --->`
	`<*> Userspace I/O drivers  --->`
	`<*>  Freescale Serial RapidIO support`
	`[*] Staging drivers  --->`
	`[*]  Freescale RapidIO Message Manager support`

USDPAA FRA Application Suite	FRA is a multi-thread USDPAA application that runs simultaneously on one or more boards which are connected with RapidIO cable. FRA application forwards IPv4 packets from one Ethernet interface to another boards using RapidIO transaction.

DPAA resource configuration

Table 683. DPAA resource configuration

DPAA Resource	USDPAA FRA Function
FMan	• Hash is based on the SIP,DIP

QMan	• Push mode, Triple dequeue enable.
	• DQRR:
	  • Entry Stashing on.
	  • To L1 cache, with high priority.
	• Frame data, Annotation, Context stashing on: –Frame data = 1 line, Annotation = 1 line, Context = 1
	• Consumer Index write mode cache-inhibited: Update on every 3 frames dequeue.
	• EQRR: Access through Cache Enable Access (CENA, WIMGE=00000)

BMan	• Buffer Pool: bp10
	  • Store doorbell messages
	  • Buffer Size: 80 Bytes
	  • Number of buffers: 256
	• Buffer Pool: bp11
	  • Store data-streaming/mailbox messages
	  • Buffer Size: 1600 Bytes
	  • Number of buffers: 8192
	• Buffer Pool: bp12
	  • Store s/g tables
	  • Buffer Size: 64 Bytes
	  • Number of buffers: 8192

DMA Memory	• 256MB for 32-bit platform • 256MB for 64-bit platform
RMan	• Enable error frame queue (only for T4240QDS) • Enable Outbound Segmentation Interleaving
sRIO port	• 4x LP-Serial link interfaces • Transmission rates of 5 Gbps (data rates of 4.0 Gbps) per lane • 4 x lane mode

12.3.4.3 Test Procedure

RMan performance is measure by USDPAA FRA.

FRA Working Flows

The FRA application can run on two boards at the same time. The two boards acting as host and agent are connected together with a RapidIO cable. FRA first parses the configuration, and then creates threads to do packet processing. FRA provides two packet processing flows: Processing1 and Processing2.

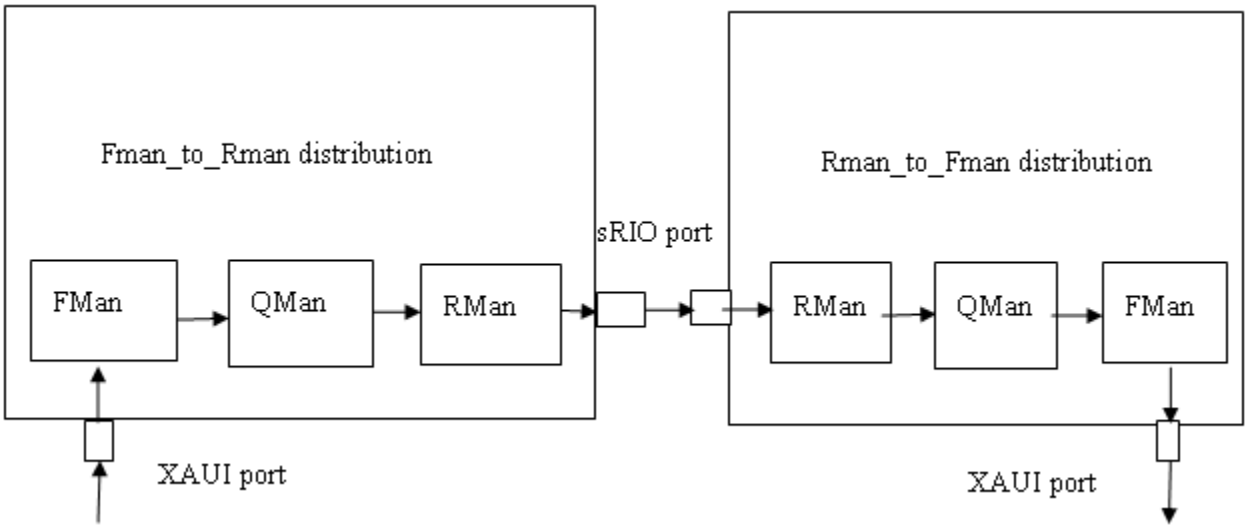


Figure 419. Packet Processing - Processing1

In the Processing1 flow, the RMan directly enqueues the packets to FMan dedicated transmission channel. Then, the FMan directly enqueues the packets to RMan dedicated transmission channel, and the core does not need to participate in the processing.

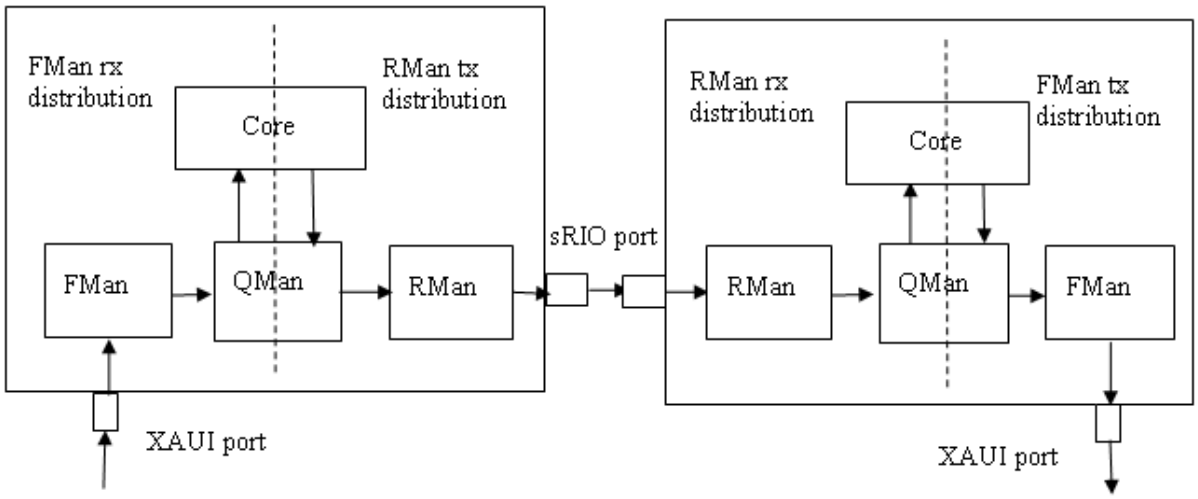


Figure 420. Packet Processing - Processing2

In the Processing2 flow, the FMan and RMan enqueue the packets to a frame queue of pool channel. Then, the core will dequeue and process the packets, and enqueue them to the FMan or RMan dedicated transmission channel to be sent out.

Procedure for FRA

1. Boot board in USDPAA mode
2. Execute the following commands on the board:

```
# cd /usr/etc
```

3. Configure PCD using the command:

```
# fmc -c usdpaa_config_p3_p5_serdes_0x33.xml -p usdpaa_policy_hash_ipv4.xml -a
```

   XML files can be found in the SDK. Use a different configuration file for each platform.

4. Launch FRA using the command:

```
# fra -c usdpaa_config_p3_p5_serdes_0x33.xml -p usdpaa_policy_hash_ipv4.xml -f
fra_config_dstr_processing1.xml
```

   FRA uses the following configuration files:

   - Packet processing using a data-streaming transaction:
     - `fra_config_dstr_processing1.xml`
     - `fra_config_dstr_processing1_port2.xml`
     - `fra_config_dstr_processing2.xml`
     - `fra_config_dstr_processing2_port2.xml`
     - `fra_config_dstr_port1_port2_loopback.xml`
     - `fra_config_dstr_testspeed_rxpoint.xml`
     - `fra_config_dstr_testspeed_txpoint.xml`
   - Packet processing using a mailbox transaction:
     - `fra_config_mbox_processing1.xml`
     - `fra_config_mbox_processing1_port2.xml`
     - `fra_config_mbox_processing2.xml`
     - `fra_config_mbox_processing2_port2.xml`

5. Launch TestCenter project to pump traffic, and the FRA is ready to receive traffic.
6. Create ‘Raw Stream’ in the generator for each port.
7. Compose a packet by changing the Mac/IP address and IPv4 Modifier Count according to your environment.

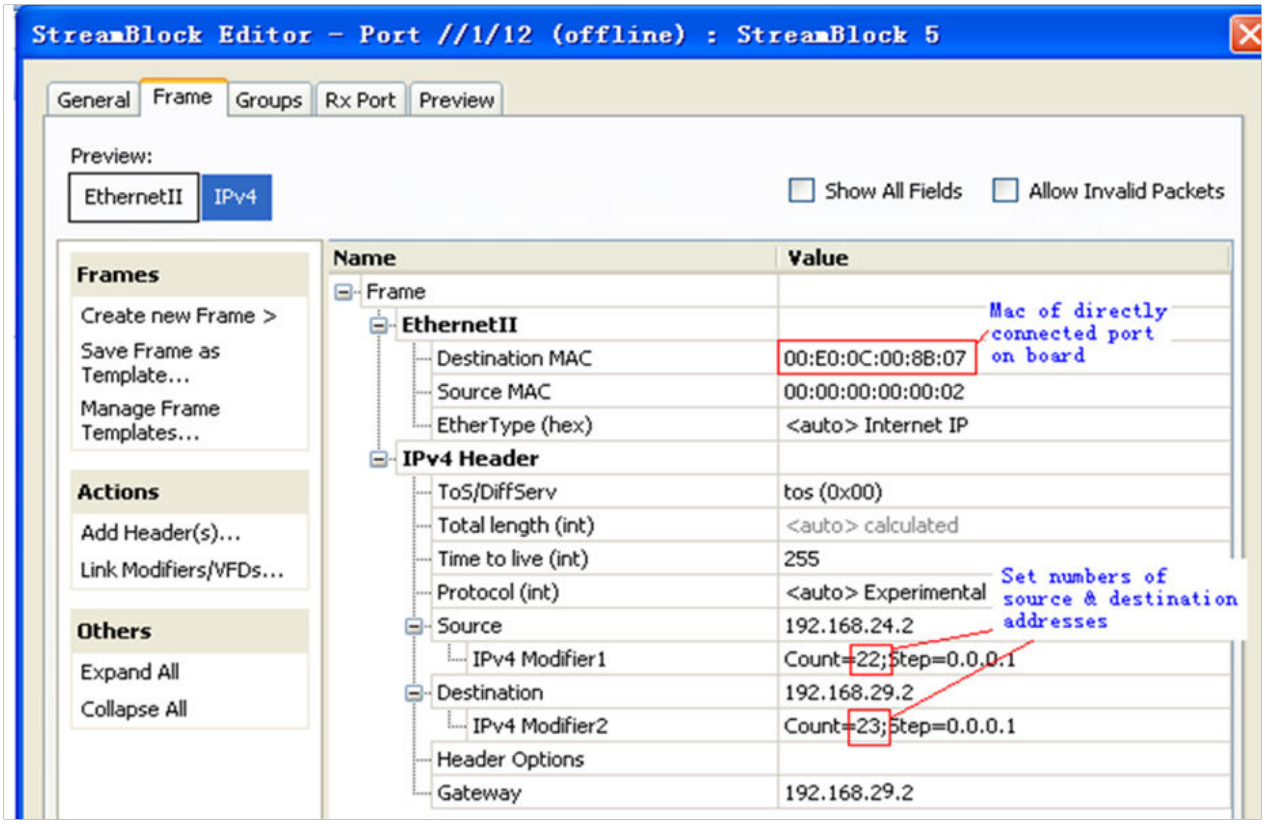


Figure 421. StreamBlock Editor

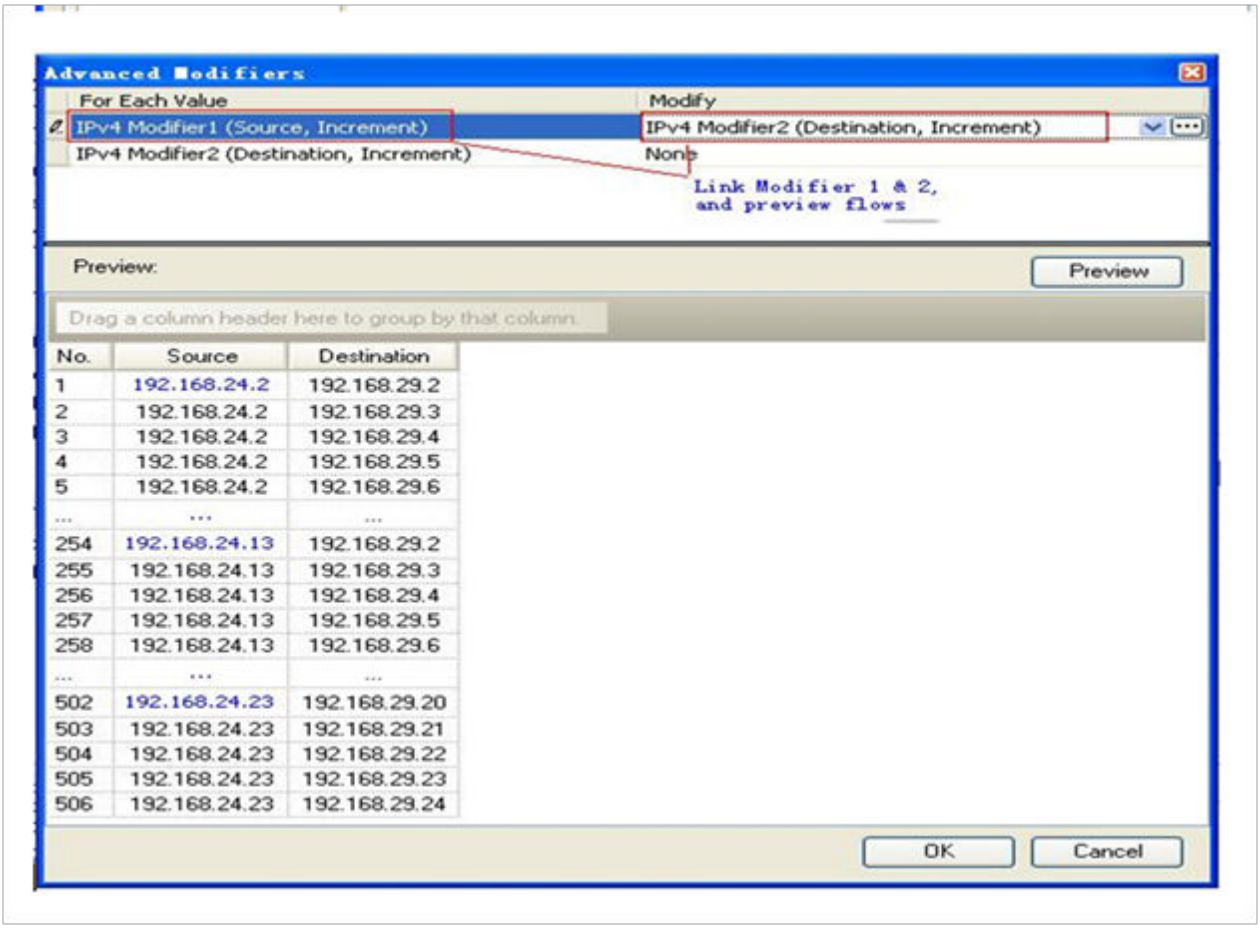
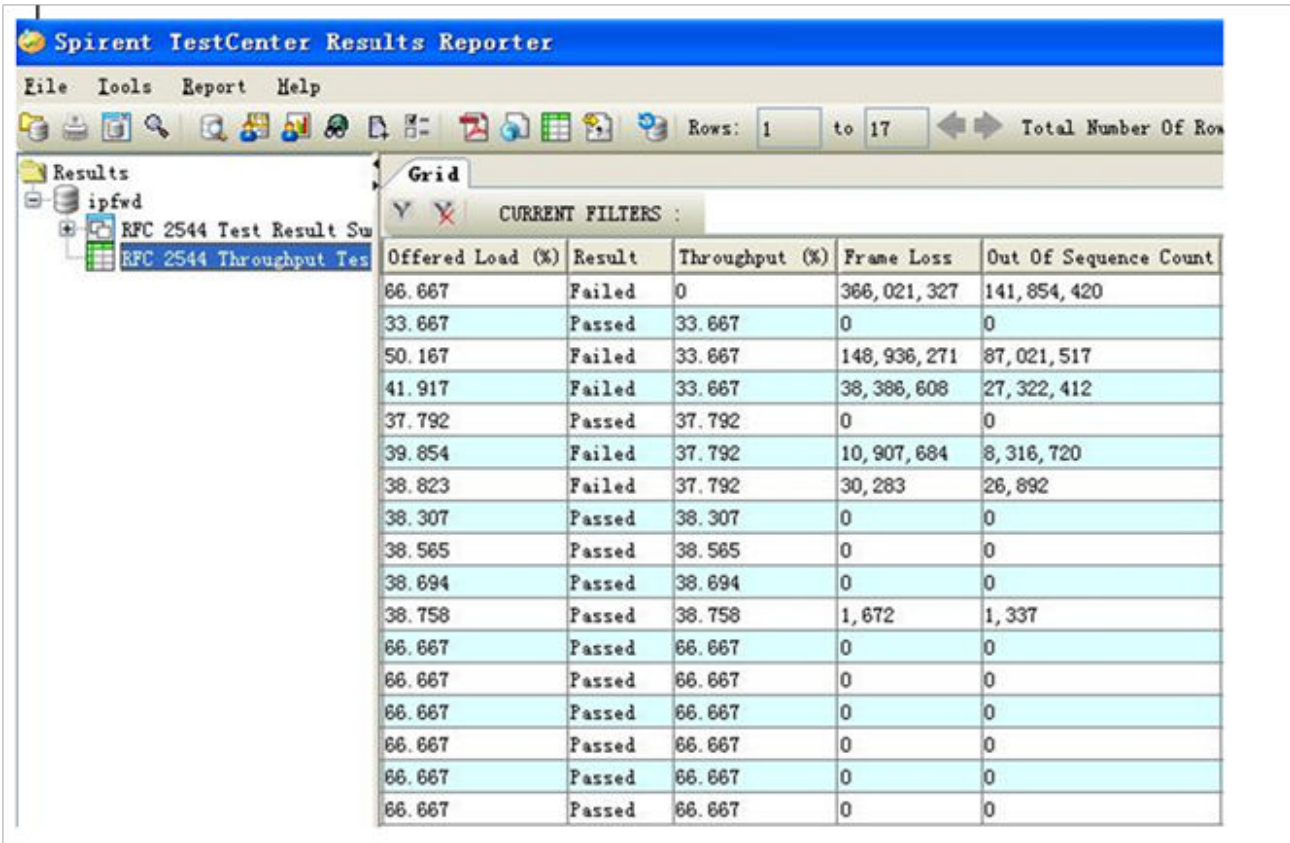


Figure 422. Advanced Modifiers

8. Create the 'Command Sequencer' in the Spirent TestCenter and select the 'RFC2544 Throughput' template.
9. Make the following variable settings:
   - Test Duration: 60 seconds
   - Resolution: 0.1%
   - Packet Loss Rate: 0.001%
   - Packet size: 64, 260, 1024 bytes
10. Start the Command Sequencer to measure performance.

12.3.4.4 View of Test Result

The following shows the results of the Test Center Results Reporter, when measuring the performance of the RFC 2544 Command Sequence:



Offered Load (%)	Result	Throughput (%)	Frame Loss	Out Of Sequence Count
66.667	Failed	0	366,021,327	141,854,420
33.667	Passed	33.667	0	0
50.167	Failed	33.667	148,936,271	87,021,517
41.917	Failed	33.667	38,386,608	27,322,412
37.792	Passed	37.792	0	0
39.854	Failed	37.792	10,907,684	8,316,720
38.823	Failed	37.792	30,283	26,892
38.307	Passed	38.307	0	0
38.565	Passed	38.565	0	0
38.694	Passed	38.694	0	0
38.758	Passed	38.758	1,672	1,337
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0
66.667	Passed	66.667	0	0

Figure 423. Spirent Test Center Report

12.3.5 sRIO

Instructions for executing benchmarking code for sRIO performance measurements of the P3041DS.

12.3.5.1 Introduction

Instructions for executing benchmarking code for USDPAA sRIO performance measurements.

The USDPAA sRIO performance benchmark is a specific instance from a suite of embedded SoC benchmarks. The goal of this exercise, summarized in the list below, is to provide the user with a range of performance options from which to select an SoC performance target that matches end product requirements.

The USDPAA sRIO performance benchmark can be implemented on the P2041/P3041/P4080/P5020/T4240/B4860 platforms. This document gives an instance of the steps based on the P3041 platform, although other platfroms may also refer to this document.

12.3.5.1.1 Platform Identification

Hardware Platform Identification

Depending on the platform, the application uses the setup shown in the table below.

Table 684. System Setup

Platform	Platform Rev.	Silicon Rev.	Default frequency Core/CCB/DDR/FMan	SerDes Protocol	Configuration Name
P2041RDB	Rev B	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 1333 MHz • FMan - 583 MHz	0x09	RR_PX_0x09
P3041DS	Rev X1	Rev 2.0	• CPU - 1500 MHz • CCB - 750 MHz • DDR - 1333 MHz • FMan - 583 MHz	0x36	RR_HXAPNSP_0x36
P4080DS	Rev X3	Rev 3.0	• CPU - 1500 MHz • CCB - 800 MHz • DDR - 1300 MHz • FMan - 600 MHz	0xe	R_PPSXX_0xe
P5020DS	Rev X7	Rev 2.0	• CPU - 2000 MHz • CCB - 800 MHz • DDR - 1333 MHz • FMan - 600 MHz	0x36	RR_HXAPNSP_0x36
P5040DS	Rev X4	Rev 2.1	• CPU - 2266 MHz • CCB - 800 MHz • DDR - 1600 MHz • FMan - 600 MHz	0x02	RR_XXSNSpP_0x02
B4860QDS	Rev B2	Rev 2.1	• CPU - 1600 MHz • CCB - 667 MHz • DDR - 1867 MHz • FMan - 667 MHz	0x2A_0x8D	N_RSSS_0x2A_0x8D
T4240QDS	Rev X4	Rev 2.0	• CPU - 1667 MHz • CCB - 733 MHz • DDR - 1866 MHz • FMan - 733 MHz	1_1_5_5	RR_XXXXPRPR_1_1_5_5

Software Platform Identification: Host development System Software with the latest version

- QorIQ DPAA SDK 1.8 Release
- Compiler from SDK 1.5

- GCC v4.7.3, GLIBC 2.15
- Linux Kernel v3.8.13

12.3.5.1.2 Setting Up the Test Environment

Details of the operational variables and product characteristics for both the hardware and the software are presented. For information on the P3041 SoC and P3041DS board hardware configurations, please refer to:

- *P3041 Reference Manual*
- *P3041DS Reference Manual*

NOTE

The following instructions are specific to P3041DS, though this guide may also be used for other platforms.

Hardware Environment

The following figure shows a high-level hardware setup of the sRIO application.

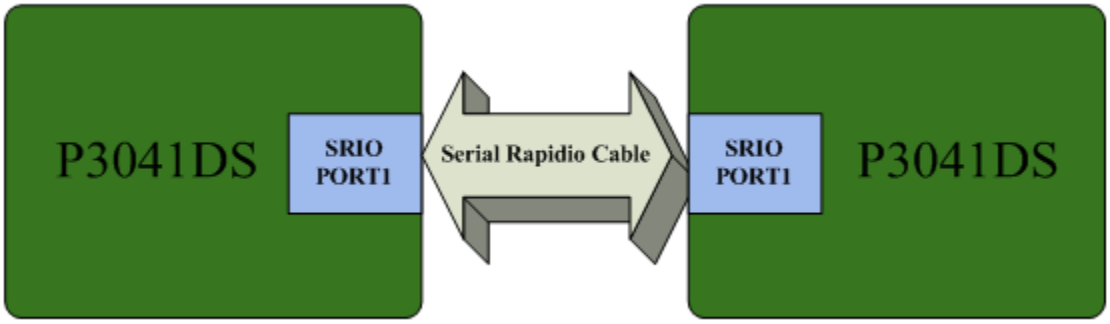


Figure 424. Test Configuration

Identifying the Hardware Version for P3041 platform

The details of the SoC and Board hardware are shown below:

Table 685. SoC Supported Hardware Configuration

Characteristic	Description
CPU Core	Rev 1.0, e500mc Core version 2.2
Cache	• L1: 32KB • L2: 128KB SPLRU with aging • CPC: 1MB SPLRU with aging
Operating Frequency	• CPU - 1500 MHz • CCB - 750 MHz • DDR3 - 667 MHz (1333 MT/s) • FMan - 583 MHz

Table continues on the next page...

Table 685. SoC Supported Hardware Configuration (continued)

Characteristic	Description
Board	P3041DS, Sys ID: 0x1c, Sys Ver: 0x01, FPGA Ver: 0x04
DDR3	DDR3 667MHz Asynchronous (transaction rate 1333MT/s) • 1T timing, No ECC • 2 Chip select cache line interleaving per controller.
Chip Select	Each covers 1024MB memory = total 2GB
Chip Select	Each covers 1024MB memory = total 2GB
SGMII Card	1 card with 3 Ethernet ports and 2 RGMII ports
XAUI	1 card
PCIe	1G NIC

Identifying the Software Platform Version

All software was built from SDK ISO. The details of the Software components are shown below:

Target Development System Software

The table below shows a description of each software components:

Table 686. Target Development System Software Detail

Component	Description
Boot loader	The U-Boot used is distributed with the SDK package.
RCW	The RCWs used is distributed with the SDK package. For sRIO test, Serdes 0x33 (1 XAUI ports, 2 sRIO ports and 2 RGMII port) is choosen: • rcw_0x33_1500mhz.bin
FMan ucode	The ucode used is distributed with the SDK package.
Kernel and dtb	Enable the RapidIO UIO driver and the DMA UIO driver in the kernel module configuration: Device Drivers --->\ <*> Userspace I/O drivers --->\ <*> Freescale Serial RapidIO support\ <*> Freescale DMA support
root filesystem considerations	The tests procedure for sRIO performance based on standard rootfile system.

12.3.5.2 Configuring and Running the Benchmark Test

NXP Serial RapidIO user space driver is based on NXP Linux SDK. It is composed of Linux UIO driver in Linux kernel space and sRIO driver/application in Linux user space. There are DMA and sRIO UIO drivers, which map DMA/sRIO registers and sRIO window to linux user space for user space driver usage. User space DMA and sRIO drivers provide driver interfaces for application, while timer driver provides means to measure the performance. Applications can use driver interface to accomplish the final srio function. Currently, the sRIO driver supports two rapidio ports, and the DMA driver supports two controllers and eight channels based on basic direct mode and basic chain mode . The demo SRA application can implement SRIO SWRITE, NWRITE, NWRITE_R, NREAD, ATOMIC_INC, ATOMIC_DEC, ATOMIC_CLR, ATOMIC_SET type protocols based on window/segment/subsegment, and give the user sRIO performance data to evaluate NXP rapidIO IP block.

12.3.5.2.1 Instructions for Booting Up

Table 687. Target Development System Software Detail

Step	Description
`setenv ipaddr=<board_ipaddress>`	Gives the board a static IP.
`setenv netmask=<netmask>`	Network masking required to contact the tftpserver.
`setenv serverip=<server_ipaddress>`	The IP Address of the tftp server.
`editenv bootargs`	USDPAA process driver accesses DMA memory. Append "usdpaa_mem=64M" to existing bootargs.
Deployment of the Linux kernel and File system to the P3041DS board.	
Login to the P3041DS board.	Use login ID = “root”

NOTE

For more information refer to each specific platform quickstart guide.

12.3.5.2.2 Test Commands and Scenarios

SRA (Serial RapidIO Application) Help

```
#sra
sra> sra
```

Then SRA help will display. Error command will cause SRA to print help information. Following are the help information:

```
-----------------SRIO APP CMD FORMAT-----------------
Set window attribute
sra -attr [port_id] [fun] [fun_id]([write_attr] [read_attr])
sra -attr port1/2 device_id [id]
sra -attr port1/2 target_id [id]
sra -attr port1/2 seg_num [num]
sra -attr port1/2 subseg_num [num]
sra -attr port1/2 subseg_tdid [seg_id] [tdid]
sra -attr port1/2 accept_all [id]
sra -attr port1/2 irq [id]
sra -attr port1/2 win_attr [id] [write_attr] [read_attr]
sra -attr port1/2 seg_attr [id] [write_attr] [read_attr]
```

```
Notes:
        [id] for command accept_all: 0 - disable; 1 - enable
        [id] for irq: 0 - disable; 1 - enable
                [write_attr]    : swrite/nwrite/nwrite_r
        [read_attr]     :
 nread/atomic_inc/atomic_dec/atomic_set/atomic_clr

Do sra operation
sra -op [port_id] [win_id] [seg_id] [subseg_id] [operation] [data_len]
sra -op port1/2 [win_id] [seg_id] [subseg_id] w/r/s/p [data_len]
        [data_len]      : should be less than the window/segment/subsegment's size, max size is 2M
                        data_len should be 1/2/4 for ATOMIC operation

Do SRIO test and print performance result
sra -test [case] [port] [task] [srio] [number]
 sra -test srio port1/2 dma/core [srio_type] payload_size
        [srio_type] should be swrite/nwrite/nwrite_r/nread
        payload_size should be less than 2M bytes
 sra -test dma_chain
 sra -test show_perf port1/2 times
 sra -test show_task
 sra -test free_task port1/2
-----------------------------------------------------
```

SRA Test Operation

This command uses sRIO port1 or port2 to implement RapidIO performance test, or do DMA basic chain mode test based on the [case] parameter input. In RapidIO performance test, if you just give the parameters [case], [port], [task], it will measure the sRIO performance under different transmission protocol type, with different transmission data size, and with different dma BWC (bandwith control), and print out the performance result. If you give all the parameters [case], [port], [task], [srio], [number], it will create a task for the specific sRIO transmission performance test. When you want to get the performance result from the task, you can start the calculation by the other command. In DMA basic chain mode test, SRA will copy lower region data to the upper region.

SRA Test under DMA mode

This will measure the sRIO port1 performance using DMA under different transmission protocol type with different transmission data size, and with different dma BWC (bandwith control). It will print out the performance result at the end.

```
sra> sra -test srio port1 dma
```

SRA Test under core mode

This will measure the sRIO port1 performance using core under different transmission protocol type with different transmission data size. It will print out the performance result at the end.

Note: The cache will impact the results of the performance.

```
sra> sra -test srio port1 core
```

SRA Test to create a transmission task with DMA mode

This will create a sRIO transmission task with the port1, DMA mode, the swrite protocol type, and the 4096 bytes payload size. It can print out the performance result for the specific transmission at the start of the performance calculation with the other command.

Note: The payload size should be less then 2M bytes.

```
sra> sra -test srio port1 dma swrite 4096
```

SRA Test to create a transmission task with core mode

This will create a sRIO transmission task with the port2, core mode, the nread protocol type, and the 2048 bytes payload size. It can print out the performance result for the specific transmission at the start of the performance calculation with the other command.

Note: The payload size should be less then 2M bytes, and the cache will impact the results of the performance.

```
sra> sra -test srio port2 core nread 2048
```

SRA Test for DMA chain

The SRA will implement the DMA chain mode test.

```
sra> sra -test dma_chain
```

SRA Test to start the task's performance calculation

When there is a sRIO task on port1, this command can start the performance calculation with the 10000 transmission times. And it will print out the performance result at the end.

```
sra> sra -test show_perf port1 10000
```

SRA Test to show the existing tasks

When there are sRIO tasks, this command can show the all tasks on port1 and port2, and print out the task's parameters.

```
sra> sra -test show_task
```

SRA Test to free the task

When there is a sRIO task on port1, this command can free the task.

```
sra> sra -test free_task port1
```

12.3.5.2.3 Running the Test

After two boards boot up, login to Linux and run the test commands in the Test Command and Scenarios section, so you can implement the sRIO performance. The figure below gives a more detailed description for the NWRITE/NREAD transactions.

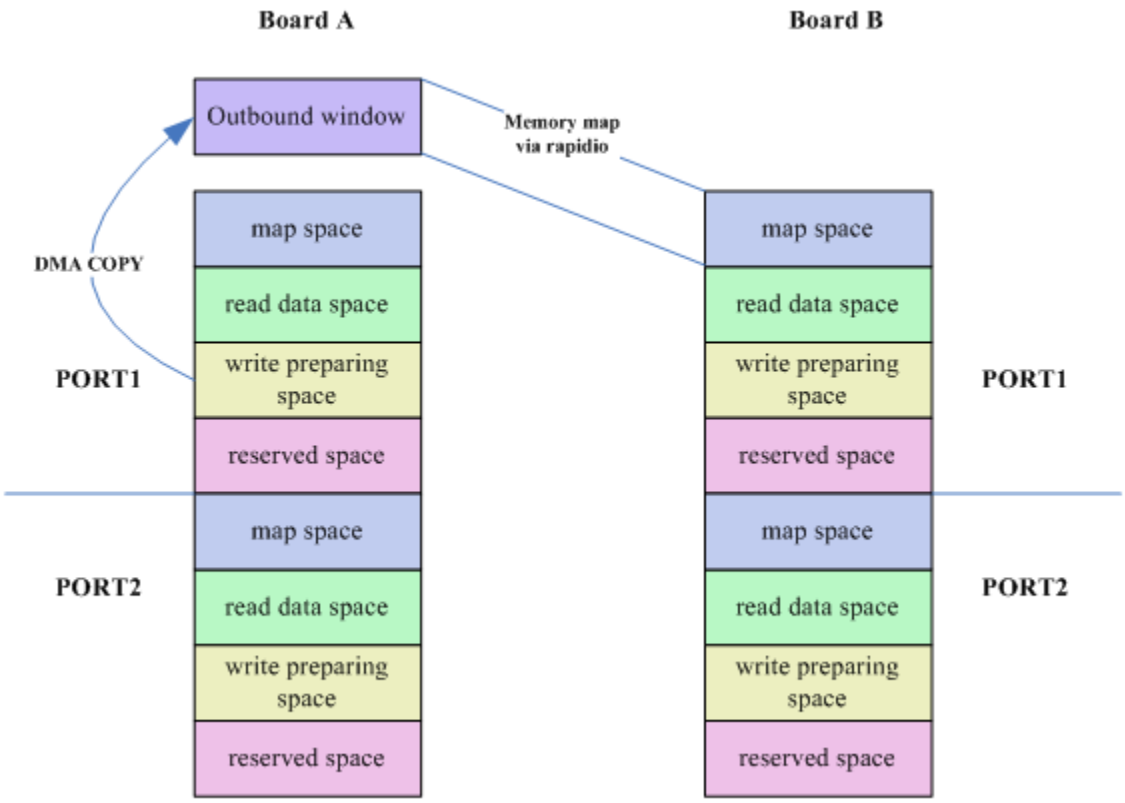


Figure 425. SRIO NWRITE Operation with Two Boards

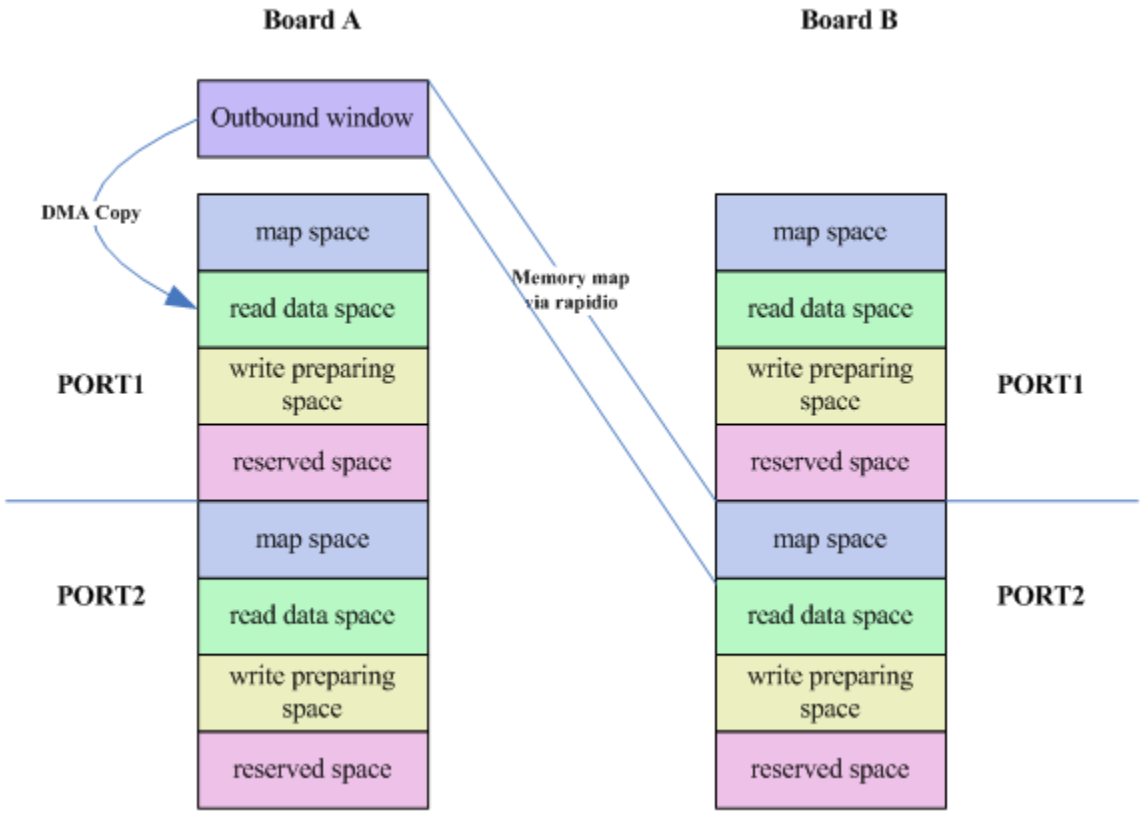


Figure 426. SRIO NREAD Operation with Two Boards

12.3.5.3 Test Result Analysis

The expected result is best when benchmarking is performed from the 1 DMA 1 exclusive channel configuration.

Theoretical Data Throughput:

The maximum transfer rate is 10 Gbps (3.125Gbps link rate, 4x, each direction, after 8B/10B encoding).

sRIO Protocol Efficiency

Table 688. sRIO Protocol Efficiency

Transaction Type	Payload	Overhead	Efficiency
SWRITE	256 Bytes	18 Bytes	93.4%
NWRITE / NWRITE_R	256 Bytes	20 Bytes	92.8%

Theoretical Payload Throughput Calculation:

sRIO Payload Throughput = (Theoretical Data Throughput) x (sRIO Protocol Efficiency)

- NWRITE / NWRITE_R = 10G x 92.8% = 9.28 Gbps
- Assumes 100% Line Utilization

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

QORIQSDK21703
Rev. 0
30 Mar 2017