AN12553

S32R274 initialization Process - from MCU Powerup to Main Function Entry

Rev. 0 — August, 2019 Application Note

by: NXP Semiconductors

Contents

1 Introduction

The S32R274 is a 32-bit Power Architecture® based Microcontroller (MCU) unit for automotive applications. It extends the MPC5775K family by a value device optimized for surround RADAR sensors and midrange front RADAR sensors. The MCU family is designed to address advanced RADAR signal processing capabilities and merge it with microcontroller capabilities for generic software tasks and bus interfacing.

This application note describes the procedure that needs to be followed during powerup to main function entry on the S32R274. It also describes the software requirements for initializing the device and starting code execution on multiple cores.

1.1 Objective

After reading this application note you can understand the following:

- The reset and boot procedure of the MCU and the transition to execute user software
- How to select the boot mode
- How to search boot location and configure them
- How to use DCF records to control initial device configurations
- How to process serial boot by BAM
- How to create, understand and configure the startup code for software requirement
- How to initialize SRAM memories
- How to start Z7 core

1.2 Register instantiations in C

C code in Application note based on MCU header files, which use unions to define MCU memory-mapped registers and to provide a structure that puts all register definitions for a particular hardware component together under one structure name.

There are two different instantiations that could be used in a C program:

```
<MODULE>.<REGISTER>.R = 0x00000001

<MODULE>.<REGISTER>.B.<BIT> = 1
```

2 Reset and boot procedure

The reset process is a sequence of several reset phases. Each reset phase has a specific entry condition, a specific exit condition, and a specific device reset behavior, which is different among the reset phases. The reset phases are executed in an order, thus building the actual reset process.

2.1 Relevant module

2.1.1 Reset Generation Module

The reset generation module (MC_RGM) generates and manages the reset process sequence of the chip, which ensures that the relevant parts of the chip are reset based on the reset source event. It provides a register interface and a reset sequencer. Various registers are available to monitor and control the chip reset sequence .

2.1.2 System Status and Configuration Module

The System Status and Control Module (SSCM) is enabled by RGM. SSCM reads the Device Configuration Format (DCF) records during boot from the flash memory and distributes the read values via an internal DCF bus to the related modules to configure certain registers in the chip during system boot, while the reset signal is asserted. It parses this data to see if valid boot code exists.

2.1.3 Boot Assist Module

The Boot Assist Module (BAM) is a block of read-only memory containing VLE code which is executed according to the boot mode of the device. The code stored in the BAM is not executed when booting in single chip mode. Except when entering the "static mode", in case no valid bootable section has been found, in case the lifecycle is failure analysis, if invalid address or password is downloaded through serial boot or in case of any communication error over serial interface.

2.1.4 Other modules

There are some other modules (PMC, STUC2, MC_ME) related to reset and boot, see the S32R274 Reference Manual for detailed description.

2.2 Reset state machine

The reset sequence is comprised of five phases managed by a state machine, which ensures that all phases are correctly processed through waiting for a minimum duration, until all processes that needs to occur during that phase have been completed before proceeding to the next phase. The state machine is used to produce the reset sequence as shown in the following figure.

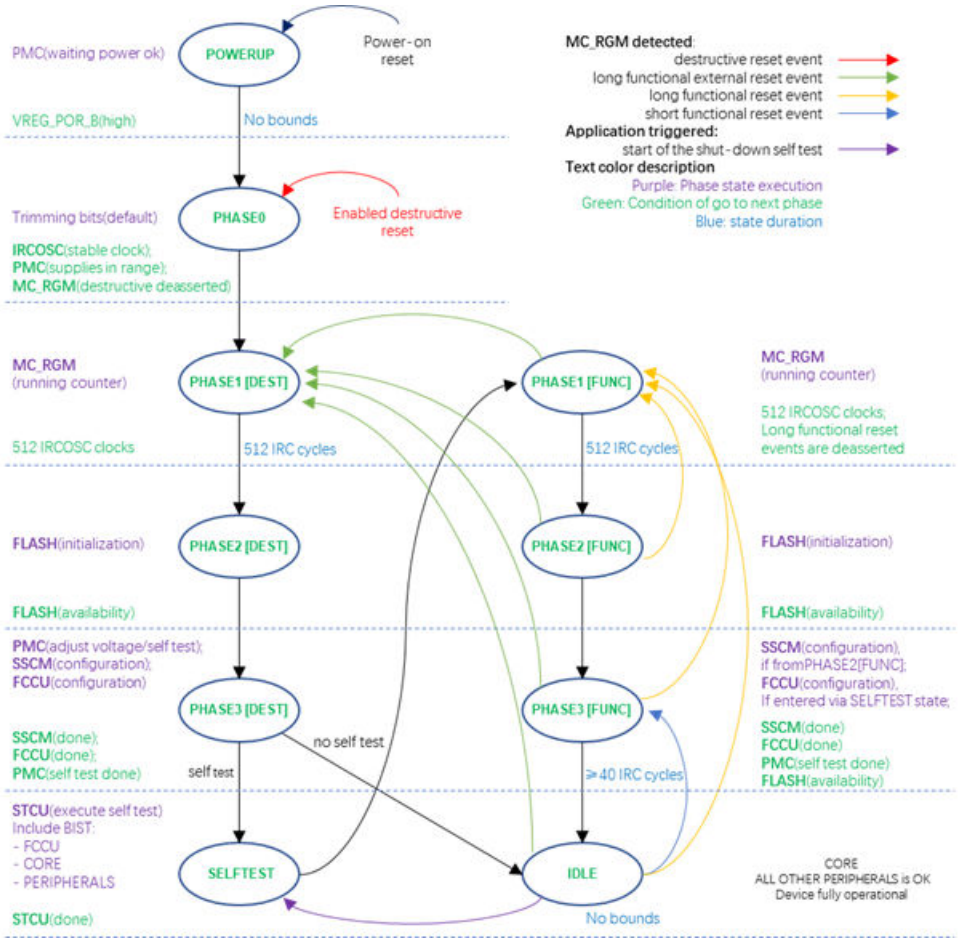

Figure 1. Reset state machine diagram

You can observe that in Figure1 there are different types of reset that may be generated when the device has reached an operational state.These different levels of reset (destructive, functional, and short) allow some system resources to be maintained in their last state during the reset event.

2.3 Boot sequence overview

The following is a high-level summary of the boot sequence.

1. When power is applied to a properly programmed device, the Power Management Controller (PMC) takes control of the device.
2. After the system power supplies have reached predefined levels, the PMC signals the RGM to begin the boot sequence.
3. During the boot sequence (reset phase two), flash is initialized is by the hardware.
4. RGM enables the SSCM to read the DCF records.
5. SSCM writes the configuration information to the specified registers.
6. In a single chip boot mode SSCM searches for the boot location.
7. After registers are initialized, the SSCM passes the control of the boot sequence back to the RGM, which directs the STCU2 to start the memory and logic built in self tests (MBIST and LBIST).
8. After MBIST and LBIST are complete, RGM releases reset and device starts code execution from reset vector.

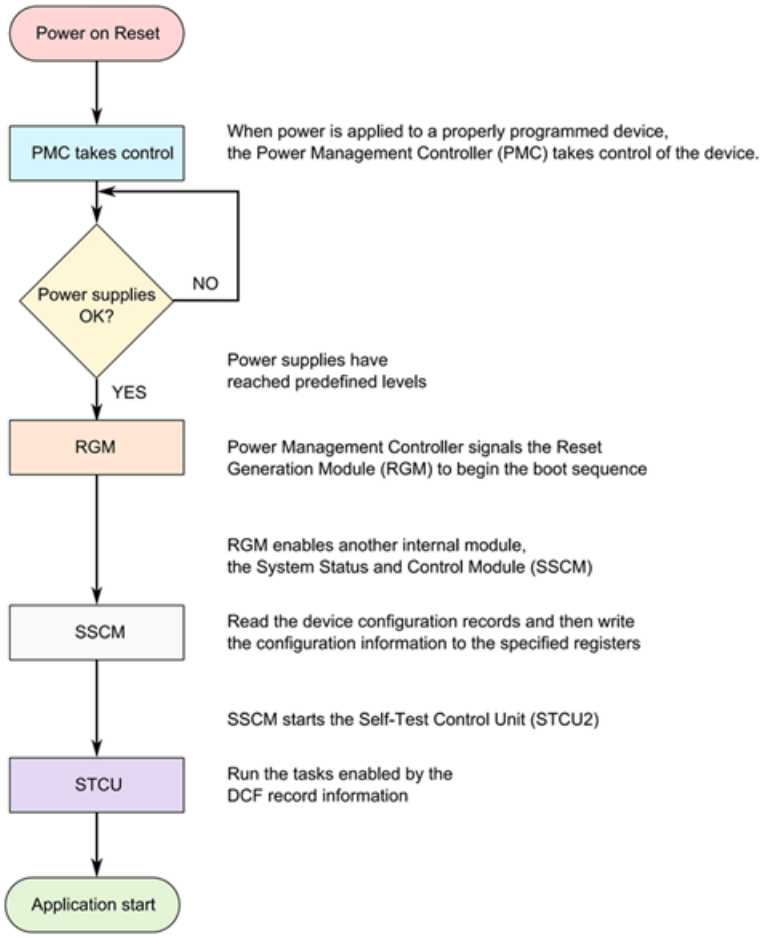

Figure 2. Boot sequence flow diagram

2.4 External signal

The following pins are associated with reset:

- VREG_POR_B
- RESET_B

2.4.1 VREG_POR_B

The VREG_POR_B is connected to P16 pin. In reset POWERUP state, VREG_POR_B pin is driven low. When VREG_POR_B pin is high, the reset state is allowed to enter into PHASE 0.

2.4.2 RESET_B

The RESET_B is connected to T17 pin, which is a bidirectional pin. The voltage level on this pin can either be driven low by an external reset generator or by the device internal reset circuitry. A high level on this pin can only be generated by an external pullup resistor which should be strong enough to overdrive the weak internal pulldown resistor.

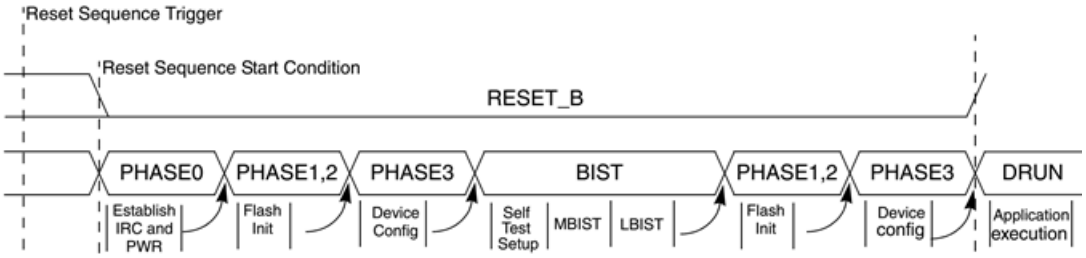

Figure 3. Reset sequence

2.5 Module status during reset process

The following figure shows the status of device modules during the reset process.

Module	POWER UP	PHASE0	PHASE 1[DEST]	PHASE 2[DEST]	PHASE 3[DEST]	SELFTEST	PHASE 1[FUNC]	PHASE 2[FUNC]	PHASE 3[FUNC]	IDLE
PMC	ON	ON	ON	ON	ON	ON	ON	ON	ON	ON
IRCOSC	RST	ON	ON	ON	ON	ON	ON	ON	ON	ON
MC_RGM	RST	ON	ON	ON	ON	ON	ON	ON	ON	ON
FLASH	RST	RST	RST	ON	ON	ON	RST	ON	ON	ON
SSCM	RST	RST	RST	RST	ON	ON	RST	RST	ON	ON
FCCU	RST	RST	RST	ON	ON	BIST	ON	ON	ON	ON
STCU	RST	RST	RST	ON	ON	ON	ON	ON	ON	ON
CORE	RST	RST	RST	RST	RST	BIST	RST	RST	RST	ON
OTHERS	RST	RST	RST	RST	RST	BIST	RST	RST	RST	ON

Figure 4. Module Status during reset states

RST: Module is held in reset by the global reset process

BIST: Module is being tested or held in a non-functional state during self-test execution as controlled by the STCU.

ON : Module is functional.

3 Boot mode

The device supports the following boot modes for the main boot core:

- Single Chip (SC) - The device boots from the first bootable section of the flash memory main array.
- Serial Boot Loader (SBL) - The device downloads boot code from either SCI or CAN interface and then execute it.

3.1 Boot mode selection

If booting is not possible with the selected configuration then the device will enter static mode. Boot mode is selected depending on:

- Life Cycle (LC):
  - Serial boot only possible in NXP production (MCU_PROD) or customer delivery (CUST_DEL) LC
  - No boot in failure analysis LC (requires test mode)
- FAB and ABS pin states

Table 1. Module status during reset states

LC	FAB pin	ABS pin	SSCM_STATUS(BMODE)	Functionality
MCU_PROD, CUST_DEL	0	x	Single chip/Flash boot	Flash boot
MCU_PROD, CUST_DEL	1	0	SCI serial boot loader	UART boot (BAM)
MCU_PROD, CUST_DEL	1	1	CAN serial boot loader	CAN boot (BAM)
OEM_PROD, IN_FIELD	x	x	Singlechip/Flash boot	Flash boot
FA	x	x	-	Static mode

3.2 External signal pin

The pins associated with boot mode select are:

- FAB
- ABS

3.3 Boot mode select flow

The following is the S32R274 boot mode select flow:

1. To boot either from FlexCAN or LINFlex, the chip must be forced into an Alternate Boot Loader mode via the FAB pin which must be asserted before initiating the reset sequence. The type of alternate boot mode is selected according to the Alternate Boot Selector (ABS) pin. Boot from FlexCAN or LinFLEX is only available when LifeCycle is MCU_PROD or CUST_DEL.
2. If FAB is not asserted and LifeCycle is MCU_PROD or CUST_DEL, the device boots from the first flash-memory sector which contains a valid boot signature.
3. If LifeCycle is OEM_PROD or IN_FIELD, then FAB and ABS pins are ignored and device always boots from Flash.
4. If no flash memory sector contains a valid boot signature, the device will go into static mode.
5. If Life cycle (LC) = Failure Analysis (FA), then device will go into static mode.

NOTE

Static mode means the chip enters the low power SAFE mode and the processor executes a wait instruction.

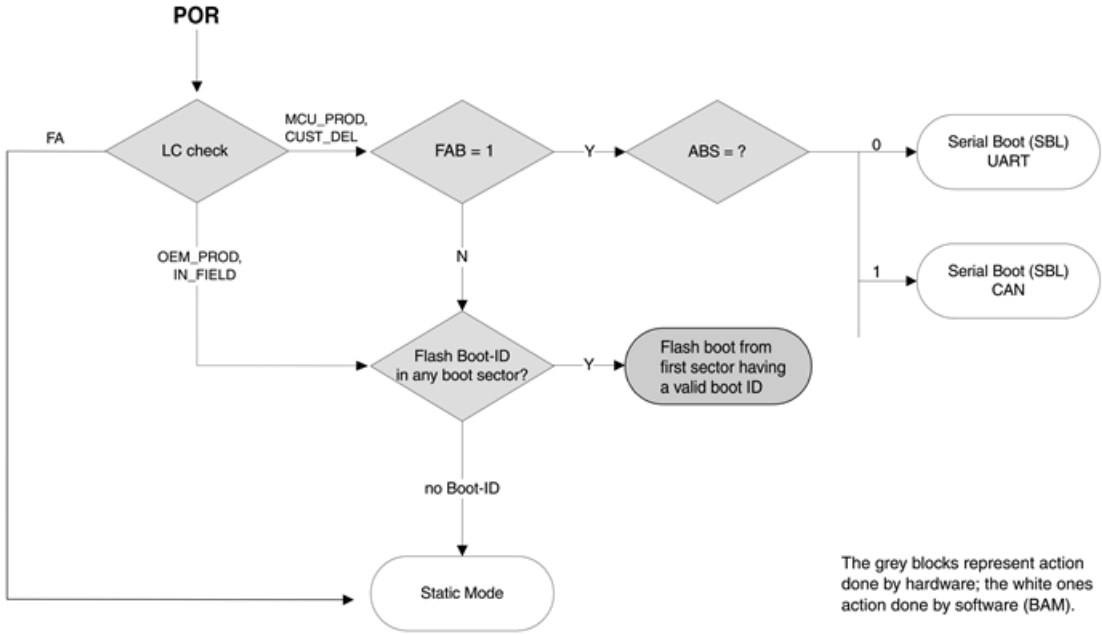

Figure 5. Boot mode selection

3.4 Life cycle status

The SSCM determines the Life Cycle (LC) of the device by reading the LC slots. The read operation is done during the reset phase with normal timings and is protected by operating monitors and ECC check. In addition, a set of sanity checks executed over the LC read data guarantees the integrity of the final LC value. At the end of the reset phase, the LC can have one of the following values:

- MCU production
- Customer delivery
- OEM production
- In field
- Failure analysis

4 Device configuration

SSCM controls the device configuration. DCF records are used by the SSCM to configure certain registers in the chip during system boot while the reset signal is asserted. The DCF records are intended to be programmed by user and contain various configurations for chip boot up. Some UTEST DCF records are written in factory and programmed during production testing. Others are written by the end user and programmed at the same time when application code is programmed into the flash memory.

4.1 DCF records structure

A DCF record is a double-word (64-bit) wide data field consisting of the following:

- Data - 32 bits of data to be written to the DCF client
- Chip select - For each DCF record, one chip select bit is asserted
- Address - Address of the DCF client within the selected module
- Parity - Parity Bit for the DCF record
- Stop - Stop bit indicates the end of the list of DCF records.

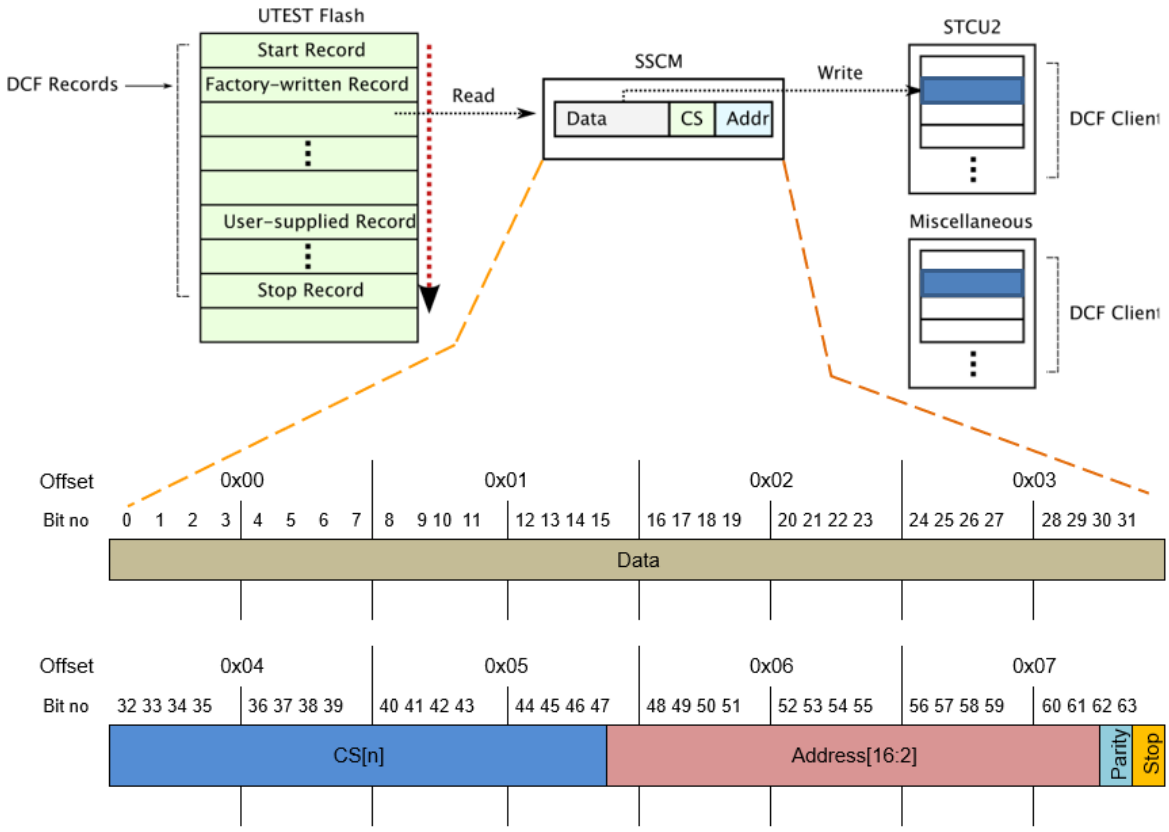

Figure 6. DCF record structure

4.2 DCF memory map

Factory-written DCF records start at the first address in the UTEST flash memory area. Initial records are programmed by NXP.

Table 2. Module Status during reset states

Start address	End address	Size(byte)	Description
0x00400300	0x00400307	8	DCF start record(0x05AA55AF00000000)
0x00400308	0x00400AFF	2040	UTEST DCF records
0x00400B00	0x00403FFF	13568	Reserved for customer OTP data

UTEST DCF records may be added at the next location in the UTEST memory map immediately following the factory (NXP) written UTEST DCF records to the end of the list by the customer.

4.3 DCF client

DCF clients are 32-bit wide hardware registers inside a module that receive and store the data from a DCF record. This stored data is used to initialize registers and configure features.

Table 3. DCF client modules

CS[14:0] Assignment	DCF Client Target module
CS[0]	SSCM
CS[1]	IPS
CS[2]	STCU
CS[3]	RAM
CS[4]	TAMPER DETECT
CS[5]	TSENS/RCOSC
CS[6]	PASS
CS[7]	Miscellaneous
CS[8]	Reserved
CS[9]	PMC/AFE

4.4 Programming DCF records

The developer must maintain a history of DCF which had been programmed. The flash programming tool used in the development environment must be configured to program the DCF record memory for new records only, compared to a standard erase/program sequence. Attempting to erase or overprogram OTP flash may result in a program failure, or other unpredictable behavior, and could leave the UTEST flash in an indeterminate state. In some extreme cases, this may even leave the device inaccessible by the debugger. Please consult with your tool vendor before programming DCF records to ensure that the DCF records are programmed as expected and without errors.

During a typical application development, the way DCF records are used may be changed once or more before the final software release.

4.5 Overwriting existing DCF records

As the UTEST flash memory is OTP (one time programable) user cannot simply rewrite already programmed data. For this purpose, S32R274 implements overwrite function.

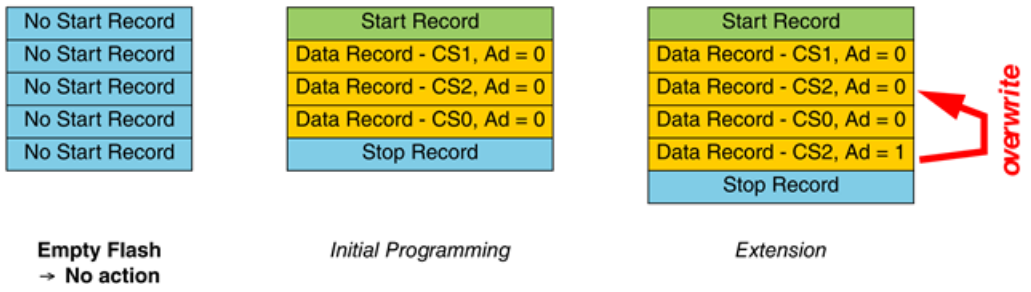

Figure 7. Overwriting DCF records

More than one DCF record can be written to the same DCF client. In this case the later record usually overrides a DCF client value set by a previous record. However, not all DCF clients allow overwrites, this depends on the DCF client implementation.

5 Boot location

For getting a vlaid boot ID a lot locations are searched in the chip. The lowest sector that starts with a valid boot ID is used to boot the device. For the flash memory locations that are searched on the chip, please refer to the chip-specific details about boot locations in flash memory in the chip reference manual.

5.1 Potential boot sectors

S32R274 series MCU boots from eight specified flash locations:

1. 0x00F98000,Flash block 0 -boot location 0, Partition 2 -16KB
2. 0x00F9C000,Flash block 1 -boot location 1, Partition 3 -16KB
3. 0x00FA0000,Flash block 2 -boot location 2, Partition 2 -64KB
4. 0x00FB0000,Flash block 3 -boot location 3, Partition 2 -64KB
5. 0x01000000,Flash block 8 -boot location 4, Partition 6 -256KB
6. 0x01040000,Flash block 9 -boot location 5, Partition 6 -256KB
7. 0x01080000,Flash block 10 -boot location 6, Partition 6 -256KB
8. 0x010C0000,Flash block 11 -boot location 7, Partition 7 -256KB

SSCM scans these eight locations during boot in reset state.

5.2 Reset configuration half-word

Each boot sector in flash memory contains the reset configuration half-word (RCHW) at offset 00h. If the RCHW field BOOT_ID holds the value 5Ah (0X01011010b), then the sector is considered bootable. In addition, there is a flag which indicates that the code is a VLE code, all other bits are reserved.

Table 4. Reset configuration half-word

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	VLE	Boot identifier							
Reserved							1	0	1	0	1	1	0	1	0

When the chip detects that it needs to boot from flash memory and also needs to find a valid BOOT_ID, the device boots from the application start address at offset 04h within the boot sector.

5.3 Boot and alternate boot sectors

Some applications require an alternate boot sector so that the main boot can be erased and reprogrammed in the field. The user can create two bootable sectors when an alternate boot is needed. The lowest sector is the main boot sector and the highest shall be the alternate boot sector. The alternate boot sector does not needs to be together with the main boot sector. This scheme ensures that there is always one active boot sector by erasing one of the boot sectors only.

- Sector is activated (that is, program a valid BOOT_ID instead of FFh as initially programmed).
- Sector is deactivated by writing 0 to some bits of the BOOT_ID field (bit 1 and/or bit 3, and/or bit 4, and/or bit 6).

5.4 Configure boot location

The RCHW occupies the most significant 16 bits of the first 32-bit internal memory word at the boot location. The next 32 bits contain the boot vector address. After applying the RCHW, the SSCM branches to this boot vector. During software initialization, reserve space for both of these 32-bit locations in the linker directive file are as follows:

```
MEMORY
{
    flash_rchw        : org = 0x00FA0000,        len = 0x4
    cpu_reset_vector  : org = 0x00FA0000+0x04,   len = 0x4
……
}
```

In the initialization code file, these two locations are generated with a valid RCHW encoding and the start address symbol for code entry point.

```
SECTIONS
{
.rchw  : { } > flash_rcw
.reset_vector : { } > cpu_reset_vector
…
}
```

6 Serial boot by BAM

SBL boot mode is managed by BAM. If some conditions are true, the device fetches code at location 0xFFFF_C000 and the BAM application starts.

6.1 Cases for executing BAM

Single chip boot mode (selecting the first bootable Flash sector) is managed by hardware and BAM do not participate in it.

BAM is executed in any of the following cases:

- Serial boot mode has been selected by FAB pin
- Hardware has not found a valid Boot ID in any flash memory boot locations
- If lifecycle is Failure Analysis.

If one of these conditions is true, the chip fetches code at location 0xFFFF_C000 and the BAM application starts.

6.2 BAM boot flow

The SSCM_STATUS[BMODE] field indicates which boot has to be executed. BMODE reset value depends on the device status after leaving reset.

BMODE:

- 001 FlexCAN (FlexCAN_0) Serial Boot Loader
- 010 LINFlex-UART (LINFlex_1) Serial Boot Loader
- Other values are reserved

If BMODE field shows the reserved values, the Boot mode is not considered valid and the BAM pushes the device into Static mode.

In all other cases data is downloaded in Serial Boot mode and saved in the correct SRAM location.

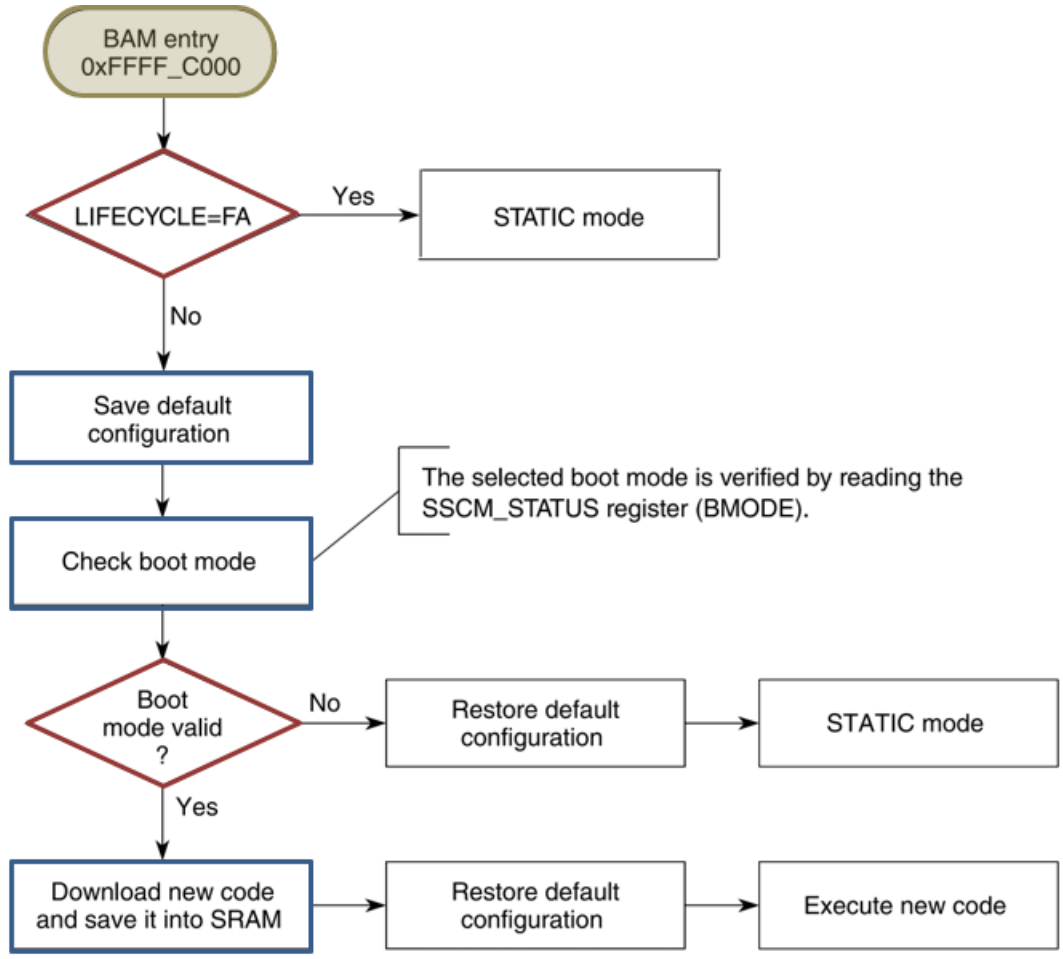

Figure 8. BAM process flow

The first action is to save the initial device configuration. In this way it is possible to restore the initial configuration after downloading the new code before executing it. This allows the new code to be executed as the device was just coming out of reset.

The initial device configuration is then restored and the code jumps to the address provided from the downloaded code. At this point BAM has just finished its task. If there is any error (that is, communication error, wrong boot mode selected, etc.), BAM restores the default configuration and puts the device into Static mode. It is needed only when the chip is unable boot in the mode which was selected.

During and after the BAM execution, the mode reported by ME_GS[S_CURRENT_MODE] in the module MC_ME Module is "DRUN".

6.3 Serial boot sequence

From high level perspective, the download protocol follows the steps shown in the following figure.

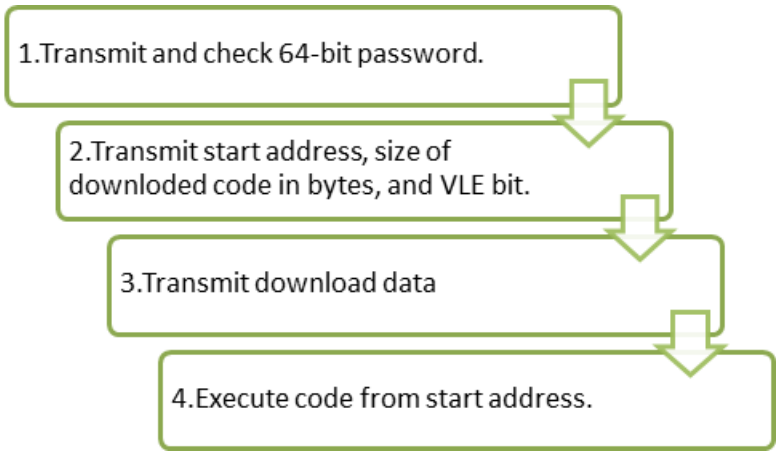

Figure 9. Serial boot flow

Each step must be completed before the next step starts. A more detailed description of these steps is given in the following sub sections.

6.3.1 Transmission mode

The communication is done in half duplex manner, any transmission from host is followed by the MCU transmission:

- Host sends data to MCU and waits
- MCU echoes to host, the data has been received
- Host verifies if echoes are correct
  - if data is correct, the host can continue to send data
  - if data is not correct, the host stops transmission and MCU must be reset.

All multi-byte data structures are sent with MSB first.

6.3.2 Download 64-bit password and password check

The first 64-bits received represent the password. This password is compared with PUBLIC password *0xFEEDFACECAFEBEEF*. If password matches, the download continues otherwise BAM puts device into Safe mode.

6.3.3 Download start address VLE bit and code size

The next 8 bytes received by the MCU contain a 32-bit Start Address, the Variable Length Instruction (VLE) bit and a 31-bit code length.

The VLE bit is used to indicate the instruction set for which the code has been compiled. The BAM supports the download of VLE code.

The Start Address defines the location at which the received data will be stored and location at which the MCU will branch after the download is complete. The two LSB bits of the start address are ignored by the BAM program, such that the loaded code should be 32-bit word aligned.

The Code Length defines the number of data bytes to be loaded. BAM code allows code length of up to 150 KB.

6.3.4 Download data

Each byte of data received is stored in the chip SRAM, starting from the address specified in the previous protocol step. It is not verified whether the provided address in SRAM is a valid address or is writable.

The address increments until the number of bytes of data received matches the number of bytes specified in the previous protocol step.

Since the SRAM is protected by a 64-bit wide Error Correction Code (ECC). for the downloaed data, the BAM performs a series of 64 bit writes to initialize the SRAM. The actual writes are performed in chunks of 32-bit. If the last byte received does not fall onto a 32-bit boundary, the BAM fills it with 0 bytes.

Finally a "dummy" word (0x0000_0000) is written to avoid a possible ECC error during core prefetch.

6.3.5 Execute code

The BAM program waits for the last echo message transmission to get completed.

Then it restores the initial MCU configuration and jumps to the code loaded at Start Address which was received in step 2 of the protocol. At this point BAM has finished its tasks and MCU is controlled by new code executed from SRAM.

6.3.6 Serial interface

The BAM downloads code into internal SRAM through the following serial protocols and executes it afterwards:

- FlexCAN (CAN_0)
- LINFlex-UART (LINFlex_1)

Depending on the selected boot mode, any download is performed with a fixed baud rate.

6.3.6.1 UART Baud rate

The LINFlexD controller is configured to operate at a baud rate of f XOSC / 833, using 8 bit data frame without parity bit and 1 stop bit.

6.3.6.2 CAN Baud rate

Boot from FlexCAN uses the system clock driven by the external oscillator. The FlexCAN controller is configured to operate at a baud rate = system clock frequency/40.

6.3.7 Inhibiting BAM operation

Under certain circumstances, you may want to inhibit BAM operation. To do this, set the SSCM_ERROR[RAE]. The default value of RAE bit is 0, which means allow access to BAM memory block.

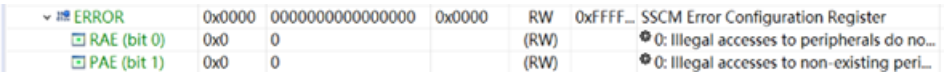

Figure 10. BAM memory block

The example C code is used to inhibit BAM operation.

```
{……
SSCM.ERROR.B.RAE = 0x1;  //set SSCM_ERROR_RAE value;
……}
```

Any attempt to access the memory range occupied by the BAM will then result in an access error.

7 Software startup

Started from reset vector or program's entry point, the initialization procedure executes and performs the minimal setup for preparation of C code execution later.

7.1 Software execution conditions

For the user application the software needs to be executed and it is necessary to achieve the following conditions when releasing reset (RESET_B pin is high):

- The related module finish reset operations:
  - The SSCM module indicates device configuration is done.
  - The FCCU module has completed its configuration sequence.
  - The PMC module indicates it has finished its internal self-test.
  - The FLASH module indicates the availability of array accesses.
- Reset status transfers to Idle, turn on the boot core
- The following must be properly programed to the respective flash memories before releasing reset:
  - User application code.
  - Reset vectors for CPU.
  - DCF records.
  - Life cycle records.

7.2 Generating example code

This typical examples code file (startup.s) is generated through S32 DS.Power wizard taking certain initialization steps.

1. Install S32DS.Power.
2. Choose S32DS Application Project.

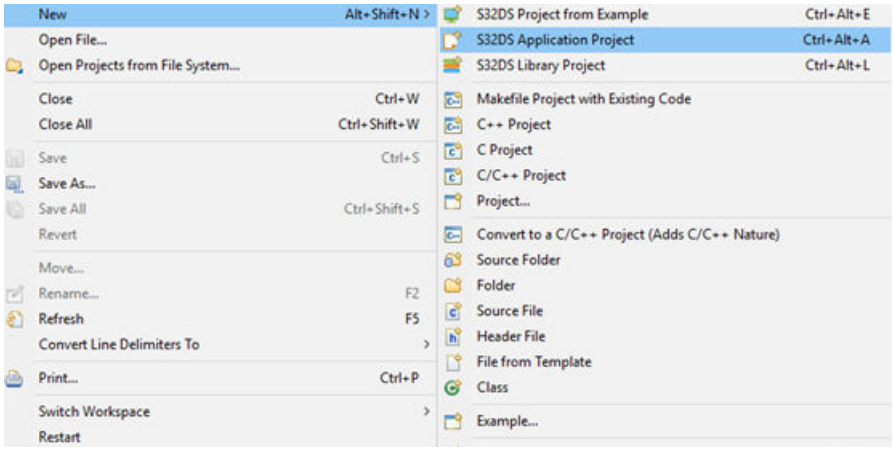

Figure 11. New project

3. Choose S32R274 Processors.

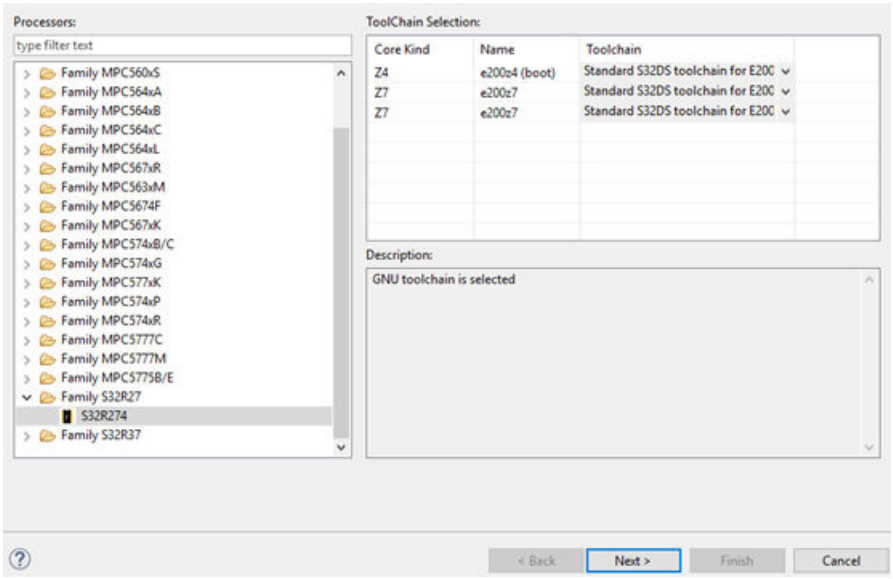

Figure 12. Select S32R274 processor

4. Choose Flash and SRAM start address and size as per the application demands, it can be adjusted here or adjusted in the link file.

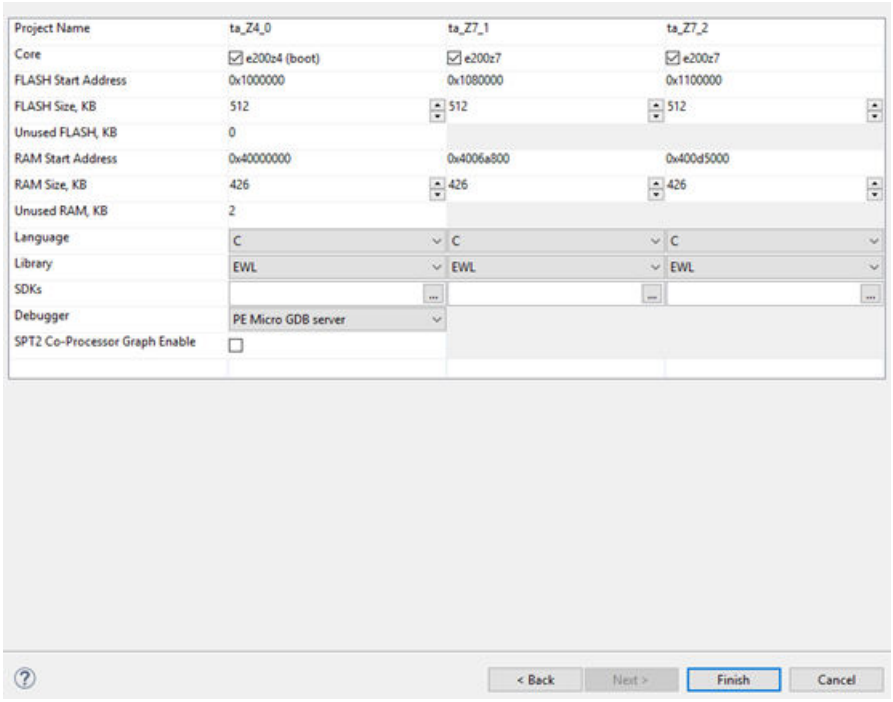

Figure 13. SRAM and Flash selection

5. Click Finish to generate the project file.
6. Startup.s file can be found under the /Project_Setting/Startup_code/ folder.

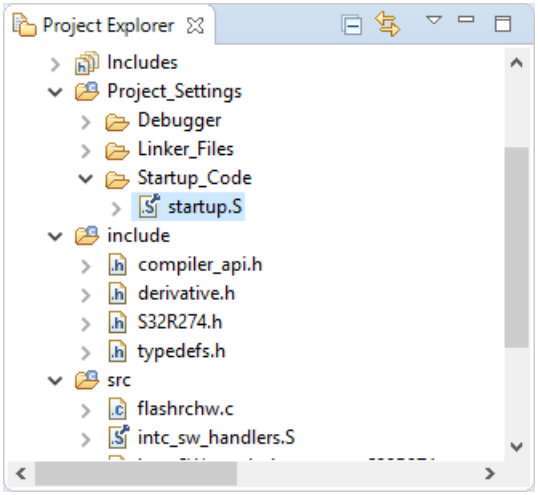

Figure 14. Startup code location

7.3 Assembly language initialization sequence

The initialization sequence implemented in the examples code is written in assembler, and the initialization flow is shown in the following figure.

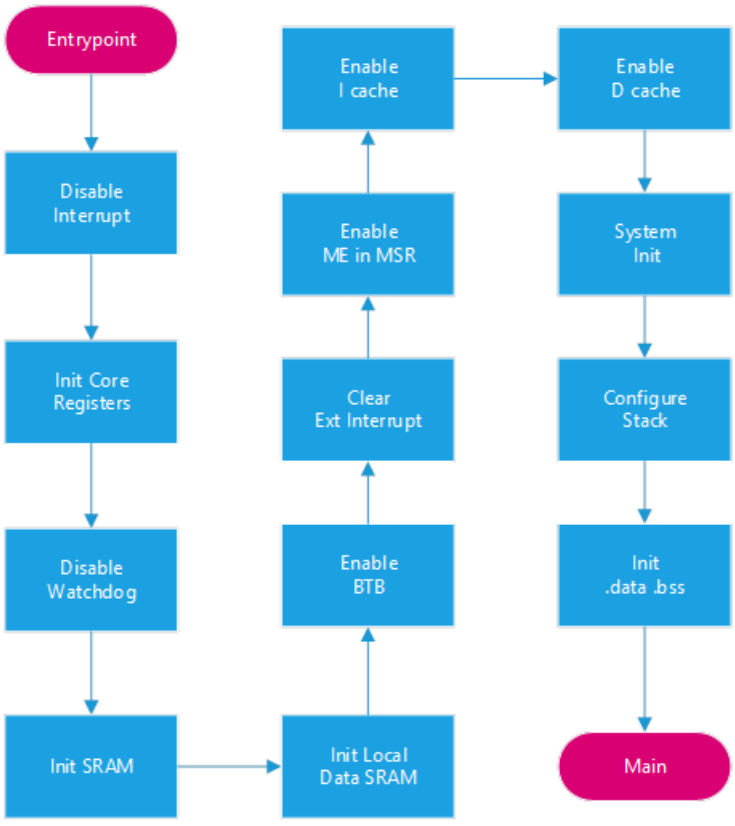

Figure 15. Initialization sequence diagram

7.4 Inhibit interrupt

7.4.1 Disable interrupts

Before software initialization interrupts need to be disabled.

```
wrteei 0 ;# Disable interrupts
```

7.4.2 Clear reservations on external interrupt

```
;#************** Clear reservations on external interrupt *****************;
# Set ICR in HID0
    e_lis      r3, 0x2
    mtspr     1008, r3
    se_isync
```

7.4.3 Enable interrupts

Before jump to main() interrupts can be enabled.

```
wrteei 1 ;# Enable interrupts
```

7.5 Core configuration

7.5.1 Initialize core registers

The two lock-steps e200z4 cores needs to initialize their registers before they are used otherwise, two lockstep cores will contain different random data. If this is the case when a value is stored to memory (e.g. stacked) it will cause a lock step error.

```
#-------------------------------------------------------#
# Initialize Core Registers #
#-------------------------------------------------------#
# GPRs 0-31
        e_li    r0, 0
        e_li    r1, 0
        e_li    r2, 0
        e_li    r3, 0
        e_li    r4, 0
        e_li    r5, 0
        e_li    r6, 0
        e_li    r7, 0
        e_li    r8, 0
        e_li    r9, 0
        e_li    r10, 0
        e_li    r11, 0
        e_li    r12, 0
        e_li    r13, 0
        e_li    r14, 0
        e_li    r15, 0
        e_li    r16, 0
        e_li    r17, 0
        e_li    r18, 0
        e_li    r19, 0
        e_li    r20, 0
        e_li    r21, 0
        e_li    r22, 0
        e_li    r23, 0
        e_li    r24, 0
        e_li    r25, 0
        e_li    r26, 0
        e_li    r27, 0
```

```
        e_li    r28, 0
        e_li    r29, 0
        e_li    r30, 0
        e_li    r31, 0
```

```
# Init any other CPU register which might be stacked (before being used).
        mtspr     1, r1            ;#XER
        mtcrf     0xFF, r1
        mtspr     CTR, r1
        mtspr     272, r1          ;#SPRG0
        mtspr     273, r1          ;#SPRG1
        mtspr     274, r1          ;#SPRG2
        mtspr     275, r1          ;#SPRG3
        mtspr     58, r1           ;#CSRR0
        mtspr     59, r1           ;#CSRR1
        mtspr     570, r1          ;#MCSRR0
        mtspr     571, r1          ;#MCSRR1
        mtspr     61, r1           ;#DEAR
        mtspr     63, r1           ;#IVPR
        mtspr     256, r1          ;#USPRG0
        mtspr     62, r1           ;#ESR
        mtspr     8, r31           ;#LR
```

7.5.2 Enable BTB

To resolve branch instructions and improve the accuracy of branch predictions, Z4 implements a dynamic branch prediction mechanism using 8-entry Branch Target Buffer (BTB). BTB is enabled via SPR1013 BUCSR (Branch Unit Control and Status Register), which is used for general control and status of BTB.

```
;#******************************** Enable BTB ******************************
;#Flush & Enable BTB - Set BBFI bit in BUCSR
    e_li       r3, 0x201
    mtspr      1013, r3
    se_isync
```

7.5.3 Enable ME bit in MSR

The Machine State Register defines the state of the processor. The e200z4201n3 MSR is shown in the following figure.

Figure 16. e200z4201n3 MSR

```
;#************************** Enable ME Bit in MSR *************************
    mfmsr     r6
    e_or2i    r6, 0x1000
    mtmsr     r6
```

7.5.4 Disable watchdog

Usually the Software Watchdog Timer (SWT) is disabled so that it does not interfere with application debug sessions. If the SWT is enabled, this should be pointed within the initialization procedure that require watchdog service, depending on the timeout period of the watchdog.

```
#************************ DISABLE WATCHDOG ********************
        e_lis   r4, SWT_BASE_ADDR@h     ;# Initialize the base address of SWT_0
        e_or2i  r4, SWT_BASE_ADDR@l

        e_li     r5, SWT_COUNT@l
        mtctr    r5                      ;# Move to counter number of SWT instances

disable_swt:
        e_li     r3, 0xC520
        e_stw    r3, 0x10(r4)            ;# Write the watchdog unlock value 0xc520

        e_li     r3, 0xD928
        e_stw    r3, 0x10(r4)            ;# Write the watchdog unlock value 0xD928
        e_lis    r3, 0xFF00
        e_or2i   r3, 0x0102
        e_stw    r3, 0(r4)
        e_addi   r4,r4, 0x4000           ;# Increase the pointer to the next instance of SWT
        e_bdnz   disable_swt             ;# Loop for all instance of SWT
```

7.5.5 SRAM initialization

The internal SRAMs feature Error Correcting Code (ECC). These ECC bits may contain random data after the chip is turned on, all SRAM locations must be initialized before being read by application code. Initialization is done by executing 64-bit writes to the entire SRAM block. The value written does not matter at this point, so the Store Multiple Word instruction will be used to write 32 general-purpose registers with each loop iteration.

```
#********************* Initialise SRAM ECC *********************
# Store number of 128Byte (32GPRs) segments in Counter
    e_lis        r5, __SRAM_SIZE@h  ;# Initialize r5 to size of SRAM (Bytes)
    e_or2i       r5, __SRAM_SIZE@l
    e_srwi       r5, r5, 0x7        ;# Divide SRAM size by 128
    mtctr        r5                 ;# Move to counter for use with "bdnz"
```

```
;# Base Address of the internal SRAM
    e_lis        r5, __SRAM_BASE_ADDR@h
    e_or2i       r5, __SRAM_BASE_ADDR@l
;# Fill SRAM with writes of 32GPRs
sram_loop:
    e_stmw       r0, 0(r5)       ;# Write all 32 registers to SRAM
    e_addi       r5, r5, 128     ;# Increment the RAM pointer to next 128bytes
    e_bdnz       sram_loop       ;# Loop for all of SRAM
#*************** Initialise Local Data SRAM ECC *****************
# Store number of 128Byte (32GPRs) segments in Counter
    e_lis        r5, __LOCAL_DMEM_SIZE@h    ;#Initialize r5 to size of SRAM (Bytes)
    e_or2i       r5, __LOCAL_DMEM_SIZE@l
    e_srwi       r5, r5, 0x7      ;#Divide SRAM size by 128
    mtctr        r5               ;#Move to counter for use with "bdnz"
;# Base Address of the Local SRAM
    e_lis        r5, __LOCAL_DMEM_BASE_ADDR@h
    e_or2i       r5, __LOCAL_DMEM_BASE_ADDR@l
;# Fill Local SRAM with writes of 32GPRs
ldmem_loop:
```

```
    e_stmw       r0,0(r5)          ;# Write all 32 registers to SRAM
    e_addi       r5,r5,128         ;#Increment the RAM pointer to next 128bytes
    e_bdnz       ldmem_loop        ;# Loop for all of SRAM
```

7.5.6 Enable cache

The core instruction and data caches are enabled through the L1 Cache Control and Status Registers 0 & 1 (L1CSR0 and L1CSR1). The instruction cache is invalidated and enabled by setting the L1CSR1[ICINV] and L1CSR1[ICE]. The data cache is enabled by setting L1CSR0[DCINV] and L1CSR0[DCE]. The cache invalidation operation takes some time and can be interrupted or aborted. Because nothing else is going on in the boot-up procedure at this point, it is not interrupted or aborted. User can set the bits and move on.

```
#************* Invalidate and enable caches *********************
# Instruction cache (I-CACHE)
    e_li r5, 0x3      # Start instruction cache invalidation and enable
    mtspr 1011,       r5 # Set L1CSR1.ICINV & ICE bits
# Data cache (D-CACHE)
    e_li r5, 0x3      # Start data cache invalidation and enable
    mtspr 1010,       r5  # Set L1CSR0.DCINV & DCE bits
*************************************************************
```

The following code represents a more robust cache enable routine that may be used if desired. This code checks to ensure the invalidation is successfully completed and if not, retries the operation before enabling the cache. This code may be used with interrupts enabled, provided that those interrupts are properly handled and cleared. If the invalidate operation cannot complete without being interrupted due to a heavy interrupt load in the system, it is better to disable interrupts first.

```
#********* Invalidate and Enable the Instruction cache **********
__icache_cfg:
    e_li     r5, 0x2
    mtspr    1011, r5

    e_li     r7, 0x4
    e_li     r8, 0x2
    e_lis    r11, 0xFFFF
    e_or2i   r11, 0xFFFB

__icache_inv:
    mfspr    r9, 1011
    and.     r10, r7, r9
    e_beq    __icache_no_abort
    and.     r10, r11, r9
    mtspr    1011, r10
    e_b      __icache_cfg

__icache_no_abort:
    and.     r10, r8, r9
    e_bne    __icache_inv

    mfspr    r5, 1011
    e_ori     r5, r5, 0x0001
    se_isync
    mtspr     1011, r5

;#************ Invalidate and Enable the Data cache **************
__dcache_cfg:
    e_li       r5, 0x2
    mtspr      1010, r5
```

```
    e_li        r7, 0x4
    e_li        r8, 0x2
    e_lis       r11, 0xFFFF
    e_or2i      r11, 0xFFFB

__dcache_inv:
    mfspr       r9, 1010
    and.        r10, r7, r9
    e_beq       __dcache_no_abort
    and.        r10, r11, r9
    mtspr       1010, r10
    e_b         __dcache_cfg

__dcache_no_abort:
    and.       r10, r8, r9
    e_bne      __dcache_inv

    mfspr      r5, 1010
    e_ori      r5, r5, 0x0001
    se_isync
    msync
    mtspr      1010, r5
```

7.5.7 C runtime register setup

The Power architecture Enhanced Application Binary Interface (EABI) specifies certain general purpose registers have special meaning for C code execution. At this point in the initialization code the stack pointer, small data, and small data 2 base pointers are set up. EABI conformant C compilers generates the code that makes use of these pointers later on.

```
    e_lis       r1, __SP_INIT@h           ;# Initialize stack pointer r1 to
    e_or2i       r1, __SP_INIT@l          ;# value in linker command file.

    e_lis        r13, _SDA_BASE_@h        ;# Initialize r13 to sdata base
    e_or2i       r13, _SDA_BASE_@l        ;# (provided by linker).

    e_lis        r2, _SDA2_BASE_@h        ;# Initialize r2 to sdata2 base
    e_or2i       r2, _SDA2_BASE_@l        ;# (provided by linker).

    e_stwu       r0, -64(r1)              ;# Terminate stack
```

As noted in the comments above, these values are defined in the linker command file for this project.

```
__DATA_SRAM_ADDR = ADDR(.data);
__SDATA_SRAM_ADDR = ADDR(.sdata);
__DATA_SIZE = SIZEOF(.data);
__SDATA_SIZE = SIZEOF(.sdata);
__DATA_ROM_ADDR = ADDR(.ROM.data);
__SDATA_ROM_ADDR = ADDR(.ROM.sdata);
```

These values in the internal flash boot case will be used to copy initialized data from flash to SRAM, but first the SRAM must be initialized.

This runtime setup procedure may vary depending on the compiler, consult your compiler's documentation. There may also be additional setup required for initializing the C standard library.

Init .data and .bss sections

```
;# Init .data and .bss sections
    e_bl   init_data_bss
```

Jump to Main

```
e_bl        main
```

8 Start Z7 core

This typical project is generated by S32DS, power wizard use Z4 core (core0) as boot core. To turn on the other two Z7 cores, you should configure Mode Entry Module (MC_ME).

8.1 Enable core

MC_ME have three core control register, MC_ME_CCTL1\MC_ME_CCTL2\MC_ME_CCTL3. They are used to check whether core is disabled or running during run modes.

These register cannot be written after a mode change request has been made or until the mode transition has completed (i.e., while the S_MTRANS bit of the ME_GS register = '1'). A write access to this register during this time will result in the ICONF_CC flag in the ME_IS register being asserted.

When secondary cores are enabled using CCTL register in target mode for the first time without setting RMC bit, it starts booting from the BAM location which in turn causes SRAM initialization. If the secondary core is enabled for the first time, the user should always program the CADDR register and set RMC bit before making transition to target mode.

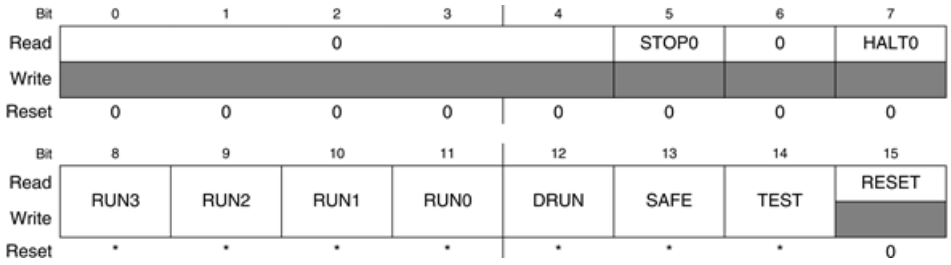

Figure 17. MC_ME CCTL register

```
{
……
    /* Enable or disable core 1 (z7a) */
    MC_ME. CCTL2 . R  = 0x00FE;        /* enable core1 */
    /* Enable or disable core 2 (z7b)*/
    MC_ME. CCTL3 . R  = 0x00FE;        /* enable core2 */
……
}
```

8.2 Setup boot address

The MC_ME_CADDR register gives the boot address for core and a bit for controlling whether core is to be reset on the next mode change or the core is configured to be running in target mode. This register can be written only as a word and cannot be written after a mode change request has been made until the mode transition has completed (i.e., while the S_MTRANS bit of the ME_GS register = '1'). A write access to this register during this time will result in the ICONF_CC flag in the ME_IS register being asserted.

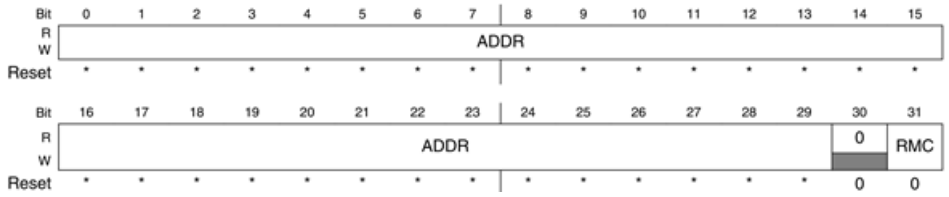

Figure 18. MC_ME_CADDR register

```
{
……
    /* Write the core address registers with the address of the first instruction */
    //MC_ME.CADDR1.R = (uint32_t)(&_start);      /* Core0 (z4) is already active */
   MC_ME. CADDR2 . R  = ( uint32_t )(&__START_ADDR_CORE_1)|0x1;    /* Core1 (z7a) active on mode change
*/
   MC_ME. CADDR3 . R  = ( uint32_t )(&__START_ADDR_CORE_2)|0x1;    /* Core2 (z7b) active on mode change
*/
……
}
```

8.3 Change core mode

Mode Control register (MC_ME_MCTL) is used to trigger software-controlled mode changes. z7 cores starts on this mode change.

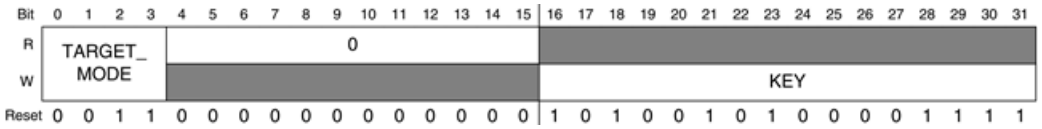

Figure 19. MC_ME_MCTL register

Target chip bits provide the target chip mode to be entered by software programming, target mode status:

```
0000 RESET (triggers a 'functional' reset event)
0001 TEST
0010 SAFE
0011 DRUN
0100 RUN0
0101 RUN1
0110 RUN2
0111 RUN3
1000 HALT0
1001 reserved
1010 STOP0
1011 reserved
1100 reserved
1101 reserved
1110 reserved
1111 RESET (triggers a 'destructive' reset event)
```

The mechanism to enter into any mode by software requires two write operations, first time with key and second time with inverted key.

```
KEY: 0101101011110000 (0x5AF0)
```

```
INVERTED KEY: 1010010100001111 (0xA50F)
```

```
{
……
    uint32_t mctl = MC_ME.MCTL.R; /* get mode status */
    MC_ME.MCTL.R = (mctl & 0xffff0000ul) | 0x5AF0ul;
    MC_ME.MCTL.R = mctl;  /* key value 2 always from MCTL */
……
}
```

The following example C function routine is used to turn on two Z7 core.

```
#define KEY_VALUE1       0x5AF0ul
#define KEY_VALUE2       0xA50Ful
voidZ7CoresInit(void)
{
#if defined(TURN_ON_CPU1) || defined(TURN_ON_CPU2)
    uint32_t mctl = MC_ME.MCTL.R;
#endif
#if defined(TURN_ON_CPU1)
    /* enable core 1 in all modes */
    MC_ME.CCTL2.R = 0x00FE;
    /* Set Start address for core 1: Will reset and start */
#if defined(START_FROM_FLASH)
    MC_ME.CADDR2.R = 0x1080000 | 0x1;
#else
    MC_ME.CADDR2.R = 0x4006a800 | 0x1;
#endif /* defined(START_FROM_FLASH) */
#endif

#if defined(TURN_ON_CPU2)
    /* enable core 2 in all modes */
    MC_ME.CCTL3.R = 0x00FE;
    /* Set Start address for core 2: Will reset and start */
#if defined(START_FROM_FLASH)
    MC_ME.CADDR3.R = 0x1100000 | 0x1;
#else
    MC_ME.CADDR3.R = 0x400d5000 | 0x1;
#endif /* defined(START_FROM_FLASH) */
#endif
#if defined( TURN_ON_CPU1) || defined(TURN_ON_CPU2)
    MC_ME.MCTL.R = (mctl & 0xffff0000ul) | KEY_VALUE1;
    MC_ME.MCTL.R = mctl;  /* key value 2 always from MCTL */
#endif
}
```

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: August, 2019
Document identifier: AN12553