

Booting Linux from a USB Flash Device on MPC5200 Systems

by David Wolfe,
Infotainment, Multimedia and Telematics Division

This document demonstrates how to boot a Linux system wholly stored on a USB Flash memory device (for example, a thumb drive, Flash stick, Flash drive, etc.) on an MPC5200 board using U-Boot as a boot monitor program. It is intended for those familiar with embedded operating systems. Familiarity with Linux would assist in the understanding of the example. The MPC5200 microcontroller is based on the e300 core which uses a PowerPC™ instruction set.

This delivery mechanism makes it easy for operating system (OS) vendors to package an operable system on a common, convenient storage device. Potential customers can plug into a minimally-configured board and run without having to install a complex system on a host computer. This procedure allows customers the opportunity to survey OSs without a significant time commitment. It also has benefit as a trivially upgradable application platform.

A Lite5200B evaluation board and the Denx Embedded Linux Development Kit (ELDK) are used in the example.

Table of Contents

1	Functional Description	2
1.1	U-Boot	2
1.2	Configuring USB	2
1.3	U-Boot Environment for Booting a RAM Disk Root File System4	
1.4	Using a Live Root File System on the USB Flash Device4	
2	Enhancements	6
2.1	File System Types	6
2.2	Running <i>fsck</i> on the Root File System	7
2.3	Loading Modules on Initial RAM Disk	8
2.4	Improved File System Layout Strategy	8

1 Functional Description

1.1 U-Boot

The boot loader for Linux on MPC5200 boards is U-Boot (“Universal Boot Loader” or “Das U-Boot”). U-Boot is able to read DOS FAT file systems on USB Flash devices.

Configuring and compiling U-Boot for USB is beyond the scope of this document. Recent MPC5200 boards shipped by Freescale contain U-Boot in boot-high Flash already configured to use USB.

A standalone program like an OS kernel can be copied from the USB Flash device into memory. The file is a binary memory image of the kernel, not an S-Record or ELF file. The kernel is then executed, and it provides the capability to get its root file system from the USB Flash device.

The Linux kernel presents a special problem because it requires a root file system before the USB driver is running and detecting devices. To work around this problem, it is possible to use U-Boot to load a RAM disk image into memory, serving as an initial root file system and then switch to the live root file system on the USB Flash device. The contents of the RAM disk is considered in the Section 1.4 describing the live USB Linux root file system. It is sufficient at this point to note the initial root file system is simple and small.

1.2 Configuring USB

Before a USB device can be accessed in U-Boot, one must run the `usb reset` and `usb scan` commands.

Occasionally the reset will fail, e.g., `USB device not responding, giving up (status=20)`. Work around the problem by unplugging and replugging in the USB Flash device. Running the reset from the U-Boot \$(pre-boot) environment script is recommended.

1.2.1 Loading Memory Images From a USB Flash Device

The U-Boot command, `fatload`, reads files from a DOS FAT file system into SDRAM. The syntax for `fatload` is as follows:

```
fatload <interface> <dev[:part]> <addr> <filename> [bytes]
- load binary file 'filename' from 'dev' on 'interface' to address 'addr' from dos filesystem
```

The command to read the file, `uImage`, from the first partition on the first USB Flash device into memory at `0x200000` is `fatload usb 0:1 0x200000 uImage`.

Putting it all together, the following commands are given to U-Boot in order. The commands typed at the U-Boot prompt are emphasized in italics below.

```

U-Boot 1.1.3 (Sep 22 2005 - 11:20:19)

CPU:   MPC5200 v2.1 at 330 MHz
       Bus 132 MHz, IPB 132 MHz, PCI 33 MHz
Board: Freescale MPC5200 (Lite5200B)
I2C:   85 kHz, ready
DRAM:  256 MB
FLASH: 32 MB
PCI:   Bus Dev VenId DevId Class Int
       00 1a 1057 5809 0680 00
In:    serial
Out:   serial
Err:   serial
Net:   FEC ETHERNET
IDE:   Bus 0: OK
Device 0: not available
Device 1: not available
=> usb reset
(Re)start USB...
USB:   scanning bus for devices... 2 USB Devices found
=> usb scan
Scan for storage device:
       scanning bus for storage devices...
Device 0: Vendor: Fujifilm Prod.: USB Drive
         Type: Removable Hard Disk
         Capacity: 244.0 MB = 0.2 GB (499712 x 512)
=> fatload usb 0:1 0x200000 uImage
reading uImage

.....

1087047 bytes read
=> fatload usb 0:1 0x400000 uRamdisk
reading uRamdisk

.....

414118 bytes read
=> setenv bootargs console=ttyS0,115200n8 root=/dev/ram init=/linuxrc rw
=> bootm 0x200000 0x400000
## Booting image at 00200000 ...
   Image Name:   Linux-2.4.25-davidw
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1086983 Bytes = 1 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
## Loading RAMDisk Image at 00400000 ...
   Image Name:   Simple Embedded Linux Framework
   Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)

```

Functional Description

```
Data Size:      414054 Bytes = 404.3 kB
Load Address:  00000000
Entry Point:   00000000
Verifying Checksum ... OK
Loading Ramdisk to 0fee9000, end 0ff4e166 ... OK
Memory BAT mapping: BAT2=256Mb, BAT3=0Mb, residual: 0Mb
Linux version 2.4.25-davidw (davidw@orthanc) (gcc version 3.2.2 20030217 (Yellow Dog
Linux 3.0 3.2.2-2a_1)) #58 Wed Sep 28 10:35:52 CDT 2005
```

1.3 U-Boot Environment for Booting a RAM Disk Root File System

Putting the above commands into a set of U-Boot environment variables makes them convenient for repeated use. The following lines can be pasted directly into a U-Boot command prompt to set the environment. Linux boots the following issue of the `run usb_linux` command.

```
setenv preboot usb reset\; echo Your message here.
setenv usbstart usb reset\; usb scan
setenv usbload_linux fatload usb 0:1 0x200000 uImage\; fatload usb 0:1 0x400000 uRamdisk
setenv bootargs console=ttyS0,115200n8 root=/dev/ram init=/linuxrc rw
setenv usb_linux run usbstart usbload_linux\; bootm 0x200000 0x400000
```

1.4 Using a Live Root File System on the USB Flash Device

Once the kernel is running with the initial RAM disk, the second part of this problem is getting Linux to use the USB Flash device as a live root file system. More sophisticated examples are discussed in [Section 2.4](#). The example was used with the ELDK 3.0 and 3.1.1 root file systems copied, and unaltered, to the second USB Flash device partition.

The file system type used by the RAM disk is not very important as long as the kernel supports it. This example uses an ext2 file system wrapped in a header used by U-Boot, using the `mkimage` program (supplied with U-Boot).

The initial RAM disk can be very small as it has very little work to do. It waits for the USB Flash device to become available then runs the `pivot_root` command to begin using the real root file system on the USB Flash device before transferring control to it.

The `bootargs` environment variable in the U-Boot configuration above is an important component for the kernel to use the initial RAM disk. Using `root=/dev/ram` tells it to use the first RAM disk while `init=/linuxrc` tells it to begin executing the `/linuxrc` program instead of the default `/sbin/init`.

The contents of the initial RAM disk are select Busybox tools (www.busybox.net), some basic `/dev` files, including RAM disk and SCSI disk nodes and the `/linuxrc` script. For reference, the RAM disk is provided with this document as both a downloadable file system image and a compressed Unix tar file.

The following `/linuxrc` script illustrates everything required of the initial ram disk. The contents are described below.

```

#!/bin/sh
PATH=/sbin:/bin
export PATH
ROOTDEV=/dev/sda2
PIVOT_ROOT=/mnt
/bin/mount -n -t proc proc /proc
/bin/mount -n -t usbfs usbfs /proc/bus/usb
#
# Wait for USB to come up
#
usb_retries=3
while test "$usb_retries" -gt 0 ; do
iif test -f /proc/bus/usb/devices ; then
usb_retries=0
else
usb_retries=`expr "$usb_retries" - 1`
echo "$usb_retries retries left for USB to come up"
sleep 1
fi
done
usb_retries=3
while test "$usb_retries" -gt 0 ; do
if grep -q 'Product=USB Drive' /proc/bus/usb/devices; then
usb_retries=0
else
usb_retries=`expr "$usb_retries" - 1`
echo "$usb_retries retries left for USB flash device to show up"
sleep 1
fi
done
while ! /bin/mount -oro $ROOTDEV /mnt; do
echo
echo "Error encountered mounting the USB flash device on $ROOTDEV."
echo 'Plugout and plugin again the USB card then hit return.'
echo
read dummy
done
cd $PIVOT_ROOT
/sbin/pivot_root . ./PIVOT_ROOT
mount -n $PIVOT_ROOT/proc/bus/usb
mount -n $PIVOT_ROOT/proc
# Can't unmount /mnt from this script with being more clever.
#umount -n /mnt
#
# Remount read/write if ELDK startup thinks / is NFS.
#
if test ! -f /etc/fstab || grep -q '^/dev/nfs' /etc/fstab; then
mount -n -orw,remount $ROOTDEV /
fi
exec /sbin/init

```

The script first unmounts the `/proc` and `/proc/bus/usb` and `/proc` pseudo file systems before it turns over control to the OS on the USB root file system by executing the `/sbin/init` program. It is

critical that `init` executes with process ID 1, the ID given by the kernel to the currently executing `/linuxrc` script; therefore, `linuxrc` starts `init` using the `exec` command replacing the run `linuxrc` process with `init` and `init` inherits process ID 1.

These are not real file systems as one might expect to find on a disk, but instead provides information from the kernel `/linuxrc` used to get state information from the USB driver. The file system waits for the USB driver to start and detect the USB Flash device. The system then attempts to mount the root file system presented by the `usb_storage` driver as a SCSI disk device. In this example, the memory images are stored in the first partition of the USB Flash device in a FAT file system, and the real root file system is stored in the second partition on an `ext2` file system. (FAT alone cannot be used because it does not support device files; however, there are ways around this limitation discussed in the [Section 2.1](#). If an error detecting the USB Flash device occurs, as with U-Boot above, unplugging the replugging in the device may clear the problem. The script waits for the return key to be struck before trying to mount the file system again. The script changes to the `/mnt` directory where it mounted the file system on the USB Flash device, executing the `pivot_root` command. This command swaps the current root file system on the RAM disk for the new one on the USB Flash device. At this point, the USB Flash device is being used for the root file system. The script first unmounts the `/proc/bus/usb` and `/proc` pseudo file systems before it turns over control to the OS on the USB root file system by executing the `/sbin/init` program. `init` brings up the rest of the operating system. Executing `init` from `/linuxrc` allows the initial RAM disk to be somewhat independent of whatever Linux distribution is used on the root file system on the USB Flash device.

NOTE

Because the `/linuxrc` is still using resources on the initial RAM disk, the initial RAM disk cannot easily be unmounted by that script. It would be possible to execute a program on the USB root file system that unmounted the RAM disk before calling `init`, or some other boot script called by `init` could unmount it. Ignoring the issue gives the example a measure of independence from the OS on the USB Flash device; however, since the contents are no longer used, unmounting the RAM disk frees up the resources (such as memory). This problem should be addressed in a production system.

2 Enhancements

2.1 File System Types

Certain strategies can be employed to place the root file system on the same USB Flash device partition as the Linux kernel and initial RAM disk memory images. This provides the flexibility of requiring only a single partition with a FAT file system on it, the way most USB Flash devices are configured by default. Many host operating systems have the capability of manipulating files on a FAT file system.

2.1.1 UMSDOS

The UMSDOS file system adds features required to make a root file system (namely device files) on top of a regular DOS/Windows FAT file system. UMSDOS relies on a user mode program, `umssync`, to enable the extended attributes in the kernel. UMSDOS is poorly supported in the Linux 2.6 kernel.

2.1.2 devfs

The `devfs` file system is a pseudo file system like `/proc`. It creates device files dynamically based on the drivers loaded in the kernel at a given time. It is mounted on `/dev` and would not replace the root file system, but provide the special device files unavailable on a simple file system like FAT. `Devfs` is deprecated in favor of `udev`, a similar dynamic tree of device files, relying on a fully-featured file system and user-mode programs.

2.1.3 RAM Disk on /dev

Any fully-featured file system like `ext2` in a RAM disk can be mounted on `/dev`. The device files can be created by the initial RAM disk `/linuxrc` script before they are needed, or with `udev`. `/dev` does not require a large RAM disk.

2.2 Running fsck on the Root File System

The Linux `fsck`, file system check, program is typically run on disk-based root file system at startup. It is unnecessary on network and RAM disk file systems. `fsck` cannot be run on mounted file systems; however, the program must be located somewhere—typically `/sbin`. This could be solved by running `fsck` from the initial root file system (IRFS), the one in the `uRamdisk` file included with this document, but that would not prevent any given Linux distribution from doing it again, and it would require additional space in the RAM disk; therefore, it must be run from the real root file system (RRFS), the one on the USB Flash device, presenting a cross-dependency.

Actually, `fsck` can be run on mounted file systems, but only if they are mounted read-only. Typically, the `ro` option is passed to the kernel thereby configuring it to mount its root file system read-only. In this case, that mounts the IRFS read-only and does not address the RRFS. It is the scripts on the RRFS which will run `fsck` on the RRFS device; therefore, the IRFS must handle the read status for the RRFS. Typically, it will mount the RRFS read-only, leaving it so when control is passed to the RRFS `init` it may run `fsck` before anything could possibly be written to the disk, causing further damage if the RRFS is in an inconsistent state.

NOTE

The IRFS in this example works read-only or read/write.

2.2.1 The Special Case for ELDK

Unmodified installations of the ELDK 3.0 and 3.1.1 root file systems assume they are served over network file system (NFS), thereby, they do not need to `remount /` with read/write access as one might do for a

local file system. In this case, they must be started with their root file system read/write. This configuration will not run `fsck` on the RRFS.

It is possible to correct the default behavior of ELDK so it does run `fsck` when the RRFS is not NFS (the case presented here). There are a few steps to make this possible:

1. The ELDK determines whether to run `fsck` based on the contents of `/etc/fstab`. If the file is missing, or empty on boot, it is created with the contents, `/dev/nfs / nfs defaults 0 0`. Changing that line to `/dev/sda2 / ext2 defaults 0 1` properly represents the RRFS presented in this document.
2. The ELDK startup scripts do not remount `/` read-write. It is required to modify `/etc/rc.d/rc.sysinit` to uncomment the executable script code under text, `# Remount the root filesystem read-write`.
3. It is also necessary to create appropriate device nodes in the `/dev` directory, for example:

```
mknod /dev/sda b 8 0
mknod /dev/sda1 b 8 1
mknod /dev/sda2 b 8 2
```

The IRFS examines `/etc/fstab` for the overly-simple regular expression, `^/dev/nfs`. If found or `/etc/fstab` is missing, the RRFS is remounted read-write before control is given to it. This check may need to be modified for some non-ELDK systems.

2.3 Loading Modules on Initial RAM Disk

Since this example goes through the effort of starting Linux with an initial RAM disk, it is possible to take advantage of this intermediate step to load kernel modules including USB and SCSI modules required to use the USB Flash device. This is commonly accomplished by personal computer Linux distributions to target the maximum peripheral base with a minimal kernel memory footprint. Only required modules to boot are loaded in the initial RAM disk, for example, the USB 1.1 or 2.0 driver could be loaded based on the processor and board combination.

2.4 Improved File System Layout Strategy

To minimize Flash wear, it is desirable to mount certain file systems as read-only. The Unix® `/usr` directory tree is designed to be mounted read-only. Consequently, read-only should work well under a properly designed Linux distribution. Directory trees required to be read/write, e.g. `/` and `/var`, can be contained in RAM disks further eliminating Flash wear. It is possible to synchronize the RAM disk file systems with images on the USB Flash device. Some directory trees such as `/boot` are not even required after the OS has booted and can be unmounted. Temporary file areas such as `/tmp` or `/dev/shm` can use special RAM disks sharing space in virtual memory able to dynamically grow and shrink as needed. If a hard drive or other backing storage is used, old entries can be paged out of main memory. This requires the system to be divided into multiple file systems mounted at strategic points in the directory tree.

Table 1. Multiple File Systems

Mount Point	Access	Medium
/	read/write	RAM disk
/boot	unmounted	live Flash
/usr	read only	live Flash
/var	read/write	RAM disk
/tmp	read/write	special RAM disk sharing space with the file cache

This type of layout can also enhance security. Multiple file systems help prevent certain types of denial-of-service attacks on writable directories like `/tmp`. Mounting read only (preferably implemented in hardware), and using copies of actual file systems in volatile RAM disk file systems, are also ways to limit access. Compromised RAM disks can be reloaded from read-only storage.

How to Reach Us:

Home Page:

www.freescale.com

E-mail:

support@freescale.com

USA/Europe or Locations Not Listed:

Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:

Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:

Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:

Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:

Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. The described product is a PowerPC microprocessor core. The PowerPC name is a trademark of IBM Corp. and is used under license. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2006. All rights reserved.