White Paper
Bruce Liao
Platform Application Engineer
Intel Corporation

Customizing Boot Media for Linux* Direct Boot

October 2013

329747-001

Executive Summary

This white paper introduces the traditional Linux*based operating system boot process, and analyzes its disadvantage. To address the shortcomings of the traditional boot process, the Linux* direct boot principle is proposed and an example is given to illustrate this principle.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. www.intel.com/embedded/edc §

Contents

Figures

Introduction

This document describes a method to set up backup operating systems with the Reference Boot Loader from Intel.

The Reference Boot Loader from Intel has a simple boot flow and low complexity, and it supports direct boot. The direct boot technology can be used to set up the backup operating system.

In some scenarios, such as in-vehicle-infotainment devices, manufacturers need a backup kernel and/or a backup rootfs to ensure that the user can switch to the backup kernel and/or rootfs to recover the original system if the active kernel and/or rootfs becomes corrupted. The direct boot technology in the Reference Boot Loader from Intel can meet the manufacturers' backup requirements.

Direct Boot Principles

Let us begin with the traditional PC Linux* boot principle.

Traditional Linux* Boot Principle

Boot in BIOS

In a PC, the Linux* boot begins in the BIOS at the address 0xFFFFFFF0. The first step of the BIOS is the power-on self-test (POST). The POST performs a check of the hardware. The second step of the BIOS is local device enumeration and initialization.

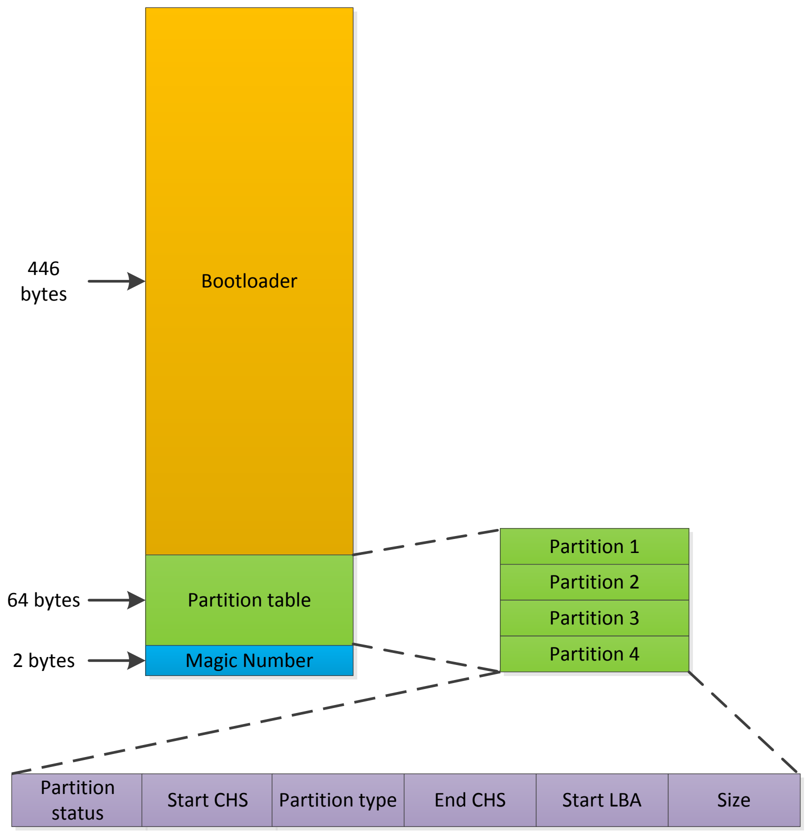
To boot an operating system, the BIOS searches for devices that are both active and bootable in the order of preference defined by the setup.

Typically, Linux* is booted from a hard disk, where the master boot record (MBR) contains the primary boot loader. The MBR is a 512-byte sector, which is located in the first sector on the disk. After the MBR is loaded into RAM, the BIOS yields control to it.

Stage 1 Boot Loader

The stage 1 boot loader that resides in the MBR is a 512-byte image containing program code and a small partition table. (See Figure 1.) The first 446 bytes are the primary boot loader, which contains executable code and error message text. The next 64 bytes are the partition table, which contains a record for each of the four partitions (16 bytes each). The MBR ends with 2 bytes that are defined as the magic number (0xAA55). The magic number serves as a validation check for the MBR.

Figure 1. MBR structure



The primary boot loader finds and loads the second boot loader (stage 2) by looking through the partition table for an active partition. When the primary boot loader finds an active partition, it scans the remaining partitions in the table to ensure that they are all inactive. When this scan is verified, the active partition's boot record is read from the device into RAM and executed.

Stage 2 Boot Loader

The stage 2 boot loader can be more aptly called the kernel loader. At this stage, the Linux kernel and optional initial RAM disk are loaded.

When the first and second stage boot loader are combined, it is called the Linux* Loader (LILO) or Grand Unified Bootloader (GRUB) in the Intel® Architecture. In this example, we are using GRUB.

The advantage of GRUB is that it recognizes Linux* file systems. GRUB can load a Linux kernel from an ext2 or ext3 file system. It does this process by turning the two-stage boot loader into a three-stage boot loader. Stage 1 (MBR) boots a stage 1.5 boot loader that understands the particular file system containing the Linux kernel image. After the stage 1.5 boot loader is loaded and running, the stage 2 boot loader can be loaded.

With stage 2 loaded, GRUB can, upon request, display a list of available kernels. You can select a kernel and even amend it with additional kernel parameters.

With the second stage boot loader in memory, the file system is consulted, and the default kernel image and initrd image are loaded into memory. With the image ready, the stage 2 boot loader invokes the kernel image.

Kernel Stage

With the kernel image in memory and control given from the stage 2 boot loader, the kernel stage begins. The kernel image is not an executable kernel but is a compressed kernel image. Typically, this kernel image is a bzImage (big compressed image, greater than 512 KB) that has been previously compressed with zlib. At the head of this kernel image is a routine that does some minimal amount of hardware setup. The routine then decompresses the kernel contained within the kernel image and places it into memory. If an initial RAM disk image is present, this routine moves it into memory and notes the image for later use. The routine then calls the kernel and the kernel boot begins.

Linux* Direct Boot Principle

From the previous section, we know the whole PC Linux* boot flow includes BIOS, stage 1 boot loader, stage 2 boot loader, and so on. The boot flow is designed to accommodate many different user scenarios, so it has lots of features and boots slowly; typically, it needs more than 5 seconds to finish loading the Linux kernel. The boot flow does not take recovery into account. If the kernel is broken, you have to re-install the system completely.

To meet the fast boot and reliability requirement of embedded customers, Intel designed the Reference Boot Loader. This loader takes care of system initialization, integrates the job of the stage 1 boot loader and the stage 2 boot loader, and provides flexibility for the users to customize the boot process.

Refer to the following Application Note, *Reference Boot Loader from Intel* (328739-001), for detailed information.

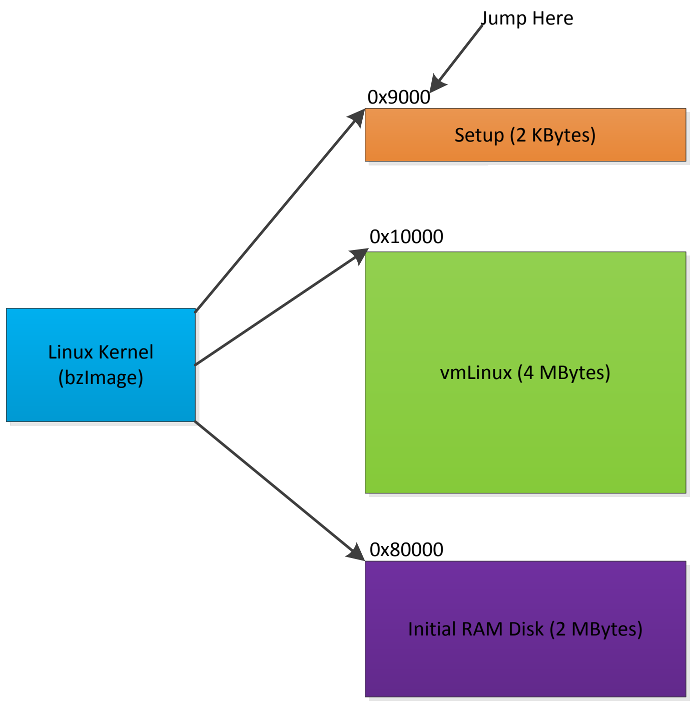
http://t.cn/zTYmc2i

Direct Boot

Because the Reference Boot Loader from Intel direct boot removes the need for the stage 1 boot loader and the stage 2 boot loader, it must implement the Linux* boot protocol to load the kernel set up image, the kernel vmLinux image, and the command line to the proper memory location. (See Figure 2.) Additionally, the Linux* setup header should be completed with correct information, such as the memory location of the kernel command line.

Refer to the boot.txt of the vanilla kernel in the \documentation\x86 directory for more information on the Linux boot protocol.

Figure 2. Kernel Load Memory Locations



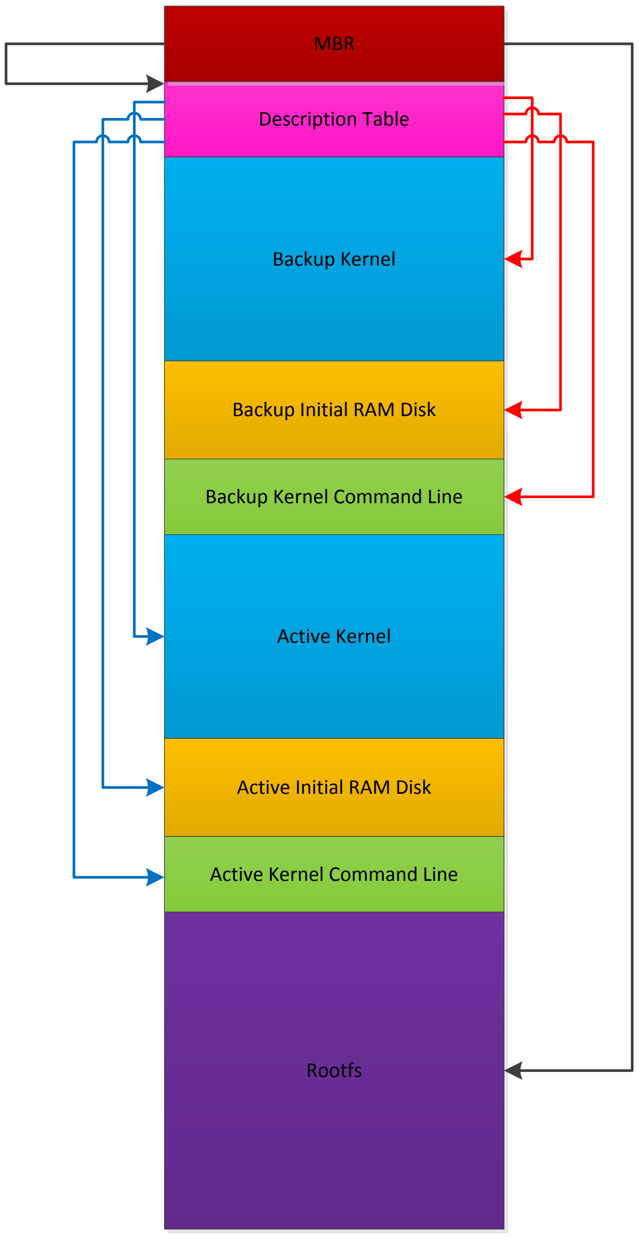
A Sample Direct Boot Methodology

The Reference Boot Loader from Intel has already provided an implementation of the Linux* direct boot. A ruby script is available in the \tools\ruby directory of the source code. You can use the script to create a direct boot disk. Refer to the next section for detailed instructions.

Figure 3 presents the layout of the boot media.

The MBR has two partitions defined in it. The first partition is a raw area that has no file system. In this partition, the active and backup kernels, initial RAM disk, and kernel command line reside. To locate these components during boot process, you need a description table to describe which block the component begins with and the length of the component. The second partition is based on the Linux* file system, and it can be ext2 or ext3, and so on.

Figure 3. Boot Media Layout



Making a Backup OS

This example is uses the MeeGo* operating system and an Intel customer reference board (CRB) on which the Intel® Atom™ Processor E6xx and the Intel® Platform Controller Hub EG20T are located. MeeGo* is a Linux*-based free mobile operating system project, which is currently hosted by the Linux* Foundation. The Linux* Foundation canceled MeeGo in September 2011 in favor of the Tizen* system.

Note: The Tizen* system is currently not easily available for downloading on the Internet. This document will be revised when the Tizen* system is available for downloading.

Although this CRB and MeeGo are no longer supported, the method shown here is applicable for all the CRBs and Linux*-based operating systems supported by the Reference Boot Loader from Intel.

To complete the OS backup, you need a working Linux* distribution host machine, a CRB, a USB drive, and a SATA drive.

1. Download the MeeGo* image to the host machine from the link below:

   http://t.cn/zT98xRA

2. Insert the USB drive into the host machine, and make sure the USB drive is unmounted before proceeding. Some Linux* distributions auto mount the USB drive when it is inserted, which can cause corruption when writing.

3. Use “dd” from the command line to byte-copy the image to the USB drive.

   ```
   eg, dd if=<image file> of=/dev/sdb bs=4096 (assuming /dev/sdb
   is the USB drive)
   ```

4. Program the CRB AMI BIOS onto flash, insert the USB drive to the CRB, and reboot, making sure that the boot order tries the USB drive first. Intel recommends using Dediprog SF100 for BIOS update.

5. During the MeeGo* installation, create a custom layout of the SATA drive. Listed below is an example:

   ```
   Partition 1, /boot, ext 3, 400 MB

   Partition 2, /      , btrfs, 1500MB

   Partition 3, swap         , 200MB
   ```

Ensure that the first partition is the boot partition and that it is greater than 200 MB. The first partition of the direct boot disk will consume 200 MB to contain the active and backup kernels, command line, and so on. If the boot partition is less than 200 MB, the rootfs will be ruined by the ruby script.

6. Insert the SATA drive into the host machine, and mount the "/boot" partition. Dump the first sector of the SATA drive with the name mbr.bin and copy the kernel file with the name bzImage.ac and bzImage.bk, as shown below:

```
dd if=/dev/sdb of=mbr.bin bs=512 count=1 (assuming /dev/sdb is
the SATA drive)
```

```
mount /dev/sdb1 /mnt
```

```
cp /mnt/boot/vmlinuz-2.6.37.6-18.1-adapation-intel-automotive
bzImage.ac
```

```
cp /mnt/boot/vmlinuz-2.6.37.6-18.1-adapation-intel-automotive
bzImage.bk
```

7. Mount the "/" partition, and open /etc/fstab, then comment the line dealing with "/boot" partition.

8. Run the Journal.rb script to format the SATA drive to the direct boot hard disk. Ensure that the mbr.bin, bzImage.ac, and bzImage.bk are in the same directory, as shown below:

```
ruby journal.rb device=/dev/sdb (assuming /dev/sdb is the SATA
drive)
```

9. Program the Reference Boot Loader from Intel onto flash, and attach the SATA drive to the CRB and boot it.

Conclusion

With the Reference Boot Loader from Intel, you can customize a direct boot media. Currently, the code can conditionally select alternate boot media to support device boot conditions, such as Battery Charging, System Update, and Boot Fail or Recovery.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://intel.com/embedded/edc.

Authors

Bruce Liao is a Platform Application Engineer with the Intel® Intelligent Systems Group.

Acronyms

CRB	Customer Reference Board
GRUB	Grand Unified Bootloader
LILO	Linux* Loader
MBR	Master Boot Record
POST	Power-On Self-Test