PRENTICE HALL

PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Essential Linux Device Drivers

"Probably the most wide ranging and complete Linux device driver book I've read."

—Alan Cox, Linux guru and key kernel developer

Sreekrishnan Venkateswaran

Foreword by Arnold Robbins

Library of Congress Cataloging-in-Publication Data:

Venkateswaran, Sreekrishnan, 1972-
Essential Linux device drivers / Sreekrishnan Venkateswaran.-- 1st ed.
p. cm.
ISBN 0-13-239655-6 (hardback : alk. paper) 1. Linux device drivers (Computer programs)
I. Title.
QA76.76.D49V35 2008
005.4'32--dc22

2008000249

Editor-in-Chief
Mark Taub

Executive Editor
Debra Williams Cauley

Managing Editor
Gina Kanouse

Project Editor
Anne Goebel

Copy Editor
Keith Cline

Indexer
Erika Millen

Proofreader
San Dee Phillips

Technical Editors
Vamsi Krishna
Jim Lieb

Publishing Coordinator
Heather Fox

Interior Designer
Laura Robbins

Cover Designer
Alan Clements

Compositor
Molly Sharp

ISBN-13: 978-0-132-39655-4
ISBN-10: 0-132-39655-6
Text printed in the United States on recycled paper at RR Donnelly in Crawfordsville, IN.
First printing March 2008

Foreword

If you're holding this book, you may be asking yourself: Why "yet another" Linux device driver book? Aren't there already a bunch of them?

The answer is: This book is a quantum leap ahead of the others.

First, it is up-to-date, covering recent 2.6 kernels. Second, and more important, this book is *thorough*. Most device driver books just cover the topics described in standard Unix internals books or operating system books, such as serial lines, disk drives, and filesystems, and, if you're lucky, the networking stack.

This book goes much further; it doesn't shy away from the hard stuff that you have to deal with on modern PC and embedded hardware, such as PCMCIA, USB, I^2C, video, audio, flash memory, wireless communications, and so on. You name it, if the Linux kernel talks to it, then this book tells you about it.

No stone is left unturned; no dark corner is left unilluminated.

Furthermore, the author has earned his stripes: It's a thrill ride just to read his description of putting Linux on a wristwatch in the late 1990s!

I'm pleased and excited to have this book as part of the Prentice Hall Open Source Software Development Series. It is a shining example of the exciting things happening in the Open Source world. I hope that you will find here what you need for your work on the kernel, and that you will enjoy the process, too!

Arnold Robbins
Series Editor

Preface

It was the late 1990s, and at IBM we were putting the Linux kernel on a wristwatch. The target device was tiny, but the task was turning out to be tough. The Memory Technology Devices subsystem didn't exist in the kernel, which meant that before a filesystem could start life on the watch's flash memory, we had to develop the necessary storage driver from scratch. Interfacing the watch's touch screen with user applications was complicated because the kernel's *input* event driver interface hadn't been conceived yet. Getting X Windows to run on the watch's LCD wasn't easy because it didn't work well with frame buffer drivers. Of what use is a waterproof Linux wristwatch if you can't stream stock quotes from your bathtub? Bluetooth integration with Linux was several years away, and months were spent porting a proprietary Bluetooth stack to Internet-enable the watch. Power management support was good enough only to squeeze a few hours of juice from the watch's battery; hence we had work cut out on that front, too. Linux-Infrared was still unstable, so we had to coax the stack before we could use an Infrared keyboard for data entry. And we had to compile the compiler and cross-compile a compact application-set because there were no accepted distributions in the consumer electronics space.

Fast forward to the present: The baby penguin has grown into a healthy teenager. What took thousands of lines of code and a year in development back then can be accomplished in a few days with the current kernels. But to become a versatile kernel engineer who can magically weave solutions, you need to understand the myriad features and facilities that Linux offers today.

About the Book

Among the various subsystems residing in the kernel source tree, the *drivers/* directory constitutes the single largest chunk and is several times bigger than the others. With new and diverse technologies arriving in popular form factors, the development of new device drivers in the kernel is accelerating steadily. The latest kernels support more than 70 device driver families.

This book is about writing Linux device drivers. It covers the design and development of major device classes supported by the kernel, including those I missed during my Linux-on-Watch days. The discussion of each driver family starts by looking at the corresponding technology, moves on to develop a practical example, and ends by looking at relevant kernel source files. Before foraying into the world of device drivers, however, this book introduces you to the kernel and discusses the important features of 2.6 Linux, emphasizing those portions that are of special interest to device driver writers.

Audience

This book is intended for the intermediate-level programmer eager to tweak the kernel to enable new devices. You should have a working knowledge of operating system concepts. For example, you should know what a system call is and why concurrency issues have to be factored in while writing kernel code. The book assumes that you have downloaded Linux on your system, poked through the kernel sources, and at least skimmed through some related documentation. And you should be pretty good in C.

Summary of Chapters

The first 4 chapters prepare you to digest the rest of the book. The next 16 chapters discuss drivers for different device families. A chapter that describes device driver debugging techniques comes next. The penultimate chapter provides perspective on maintenance and delivery. We shut down by walking through a checklist that summarizes how to set forth on your way to Linux-enablement when you get hold of a new device.

Chapter 1, "Introduction," starts our tryst with Linux. It hurries you through downloading the kernel sources, making trivial code changes, and building a bootable kernel image.

Chapter 2, "A Peek Inside the Kernel," takes a brisk look into the innards of the Linux kernel and teaches you some must-know kernel concepts. It first takes you through the boot process and then describes kernel services particularly relevant to driver development, such as kernel timers, concurrency management, and memory allocation.

Chapter 3, "Kernel Facilities," examines several kernel services that are useful components in the toolbox of driver developers. The chapter starts by looking at kernel threads, which is a way to implement background tasks inside the kernel. It then moves on to helper interfaces such as linked lists, work queues, completion functions, and notifier chains. These helper facilities simplify your code, weed out redundancies from the kernel, and help long-term maintenance.

Chapter 4, "Laying the Groundwork," builds the foundation for mastering the art of writing Linux device drivers. It introduces devices and drivers by giving you a bird's-eye view of the architecture of a typical PC-compatible system and an embedded device. It then looks at basic driver concepts such as interrupt handling and the kernel's device model.

Chapter 5, "Character Drivers," looks at the architecture of character device drivers. Several concepts introduced in this chapter, such as polling, asynchronous notification, and I/O control, are relevant to subsequent chapters, too, because many device classes discussed in the rest of the book are "super" character devices.

Chapter 6, "Serial Drivers," explains the kernel layer that handles serial devices.

Chapter 7, "Input Drivers," discusses the kernel's input subsystem that is responsible for servicing devices such as keyboards, mice, and touch-screen controllers.

Chapter 8, "The Inter-Integrated Circuit Protocol," dissects drivers for devices such as EEPROMs that are connected to a system's I^2C bus or SMBus. This chapter also looks at other serial interfaces such as SPI bus and 1-wire bus.

Chapter 9, "PCMCIA and Compact Flash," delves into the PCMCIA subsystem. It teaches you to write drivers for devices having a PCMCIA or Compact Flash form factor.

Chapter 10, "Peripheral Component Interconnect," looks at kernel support for PCI and its derivatives.

Chapter 11, "Universal Serial Bus," explores USB architecture and explains how you can use the services of the Linux-USB subsystem to write drivers for USB devices.

Chapter 12, "Video Drivers," examines the Linux-Video subsystem. It finds out the advantages offered by the frame buffer abstraction and teaches you to write frame buffer drivers.

Chapter 13, "Audio Drivers," describes the Linux-Audio framework and explains how to implement audio drivers.

Chapter 14, "Block Drivers," focuses on drivers for storage devices such as hard disks. In this chapter, you also learn about the different I/O schedulers supported by the Linux-Block subsystem.

Chapter 15, "Network Interface Cards," is devoted to network device drivers. You learn about kernel networking data structures and how to interface network drivers with protocol layers.

Chapter 16, "Linux Without Wires," looks at driving different wireless technologies such as Bluetooth, Infrared, WiFi, and cellular communication.

Chapter 17, "Memory Technology Devices," discusses flash memory enablement on embedded devices. The chapter ends by examining drivers for the Firmware Hub found on PC systems.

Chapter 18, "Embedding Linux," steps into the world of embedded Linux. It takes you through the main firmware components of an embedded solution such as bootloader, kernel, and device drivers. Given the soaring popularity of Linux in the embedded space, it's more likely that you will use the device driver skills that you acquire from this book to enable embedded systems.

Chapter 19, "Drivers in User Space," looks at driving different types of devices from user space. Some device drivers, especially ones that are heavy on policy and light on performance requirements, are better off residing in user land. This chapter also explains how the Linux process scheduler affects the response times of user mode drivers.

Chapter 20, "More Devices and Drivers," takes a tour of a potpourri of driver families not covered thus far, such as *Error Detection And Correction* (EDAC), FireWire, and ACPI.

Chapter 21, "Debugging Device Drivers," teaches about different types of debuggers that you can use to debug kernel code. In this chapter, you also learn to use trace tools, kernel probes, crash-dump, and profilers. When you develop a driver, be armed with the driver debugging skills that you learn in this chapter.

Chapter 22, "Maintenance and Delivery," provides perspective on the software development life cycle.

Chapter 23, "Shutting Down," takes you through a checklist of work items when you embark on Linux-enabling a new device. The book ends by pondering *What next?*

Device drivers sometimes need to implement code snippets in assembly, so Appendix A, "Linux Assembly," takes a look at the different facets of assembly programming on Linux. Some device drivers on x86-based systems depend directly or indirectly on the BIOS, so Appendix B, "Linux and the BIOS," teaches you how Linux interacts with the BIOS. Appendix C, "Seq Files," describes seq files, a kernel helper interface introduced in the 2.6 kernel that device drivers can use to monitor and trend data points.

The book is generally organized according to device and bus complexity, coupled with practical reasons of dependencies between chapters. So, we start off with basic device classes such as character, serial, and input. Next, we look at simple serial buses such as I^2C and SMBus. External I/O buses such as PCMCIA, PCI, and USB follow. Video, audio, block, and network devices usually interface with the processor via these I/O buses, so we look at them soon after. The next portions of the book are oriented toward embedded Linux and cover technologies such as wireless networking and flash memory. User-space drivers are discussed toward the end of the book.

Kernel Version

This book is generally up to date as of the 2.6.23/2.6.24 kernel versions. Most code listings in this book have been tested on a 2.6.23 kernel. If you are using a later version, look at Linux websites such as lwn.net to learn about the kernel changes since 2.6.23/24.

Book Website

I've set up a website at elinuxdd.com to provide updates, errata, and other information related to this book.

Conventions Used

Source code, function names, and shell commands are written `like this`. The shell prompt used is **`bash>`**. Filename are written in italics, *like this*. Italics are also used to introduce new terms.

Some chapters modify original kernel source files while implementing code examples. To clearly point out the changes, newly inserted code lines are prefixed with **`+`**, and any deleted code lines with `-`.

Sometimes, for simplicity, the book uses generic references. So if the text points you to the *arch/your-arch/* directory, it should be translated, for example, to *arch/x86/* if you are compiling the kernel for the x86 architecture. Similarly, any mention of the *include/asm-your-arch/* directory should be read as *include/asm-arm/* if you are, for instance, building the kernel for the ARM architecture. The * symbol and *X* are occasionally used as wildcard characters in filenames. So, if a chapter asks you to look at *include/linux/time*.h*, look at the header files, *time.h, timer.h, times.h,* and *timex.h* residing in the *include/linux/* directory. If a section talks about */dev/input/eventX* or */sys/devices/platform/i8042/serioX/, X* is the interface number that the kernel assigns to your device in the context of your system configuration.

The → symbol is sometimes inserted between command or kernel output to attach explanations.

Simple regular expressions are occasionally used to compactly list function prototypes. For example, the section "Direct Memory Access" in Chapter 10, "Peripheral Component Interconnect," refers to `pci_[map|unmap|dma_sync]_single()` instead of explicitly citing `pci_map_single()`, `pci_umap_single()`, and `pci_dma_sync_single()`.

Several chapters refer you to user-space configuration files. For example, the section that describes the boot process opens */etc/rc.sysinit*, and the chapter that discusses Bluetooth refers to */etc/bluetooth/pin*. The exact names and locations of such files might, however, vary according to the Linux distribution you use.

Chapter 17

Memory Technology Devices

In This Chapter

When you push the power switch on your handheld, it's more than likely that it boots from flash memory. When you click some buttons to save data on your cell phone, in all probability, your data starts life in flash memory. Today, Linux has penetrated the embedded space and is no longer confined to desktops and servers. Linux avatars manifest in PDAs, music players, set-top boxes, and even medical-grade devices. The *Memory Technology Devices* (MTD) subsystem of the kernel is responsible for interfacing your system with various flavors of flash memory found in these devices. In this chapter, let's use the example of a Linux handheld to learn about MTD.

What's Flash Memory?

Flash memory is rewritable storage that does not need power supply to hold information. Flash memory banks are usually organized into *sectors.* Unlike conventional storage, writes to flash addresses have to be preceded by an erase of the corresponding locations. Moreover, erases of portions of flash can be performed only at the granularity of individual sectors. Because of these constraints, flash memory is best used with device drivers and filesystems that are tailored to suit them. On Linux, such specially designed drivers and filesystems are provided by the MTD subsystem.

Flash memory chips generally come in two flavors: NOR and NAND. *NOR* is the variety used to store firmware images on embedded devices, whereas *NAND* is used for large, dense, cheap, but imperfect[1] storage as required by solid-state mass storage media such as USB pen drives and *Disk-On-Modules* (DOMs). NOR flash chips are connected to the processor via address and data lines like normal RAM, but NAND flash chips are interfaced using I/O and control lines. So, code resident on NOR flash can be executed in place, but that stored on NAND flash has to be copied to RAM before execution.

[1] It's normal to have bad blocks scattered across NAND flash regions as you will learn in the section, "NAND Chip Drivers."

Linux-MTD Subsystem

The kernel's MTD subsystem shown in Figure 17.1 provides support for flash and similar nonvolatile solid-state storage. It consists of the following:

- The *MTD core*, which is an infrastructure consisting of library routines and data structures used by the rest of the MTD subsystem
- *Map drivers* that decide what the processor ought to do when it receives requests for accessing the flash
- *NOR Chip drivers* that know about commands required to talk to NOR flash chips
- *NAND Chip drivers* that implement low-level support for NAND flash controllers
- *User Modules*, the layer that interacts with user-space programs
- Individual device drivers for some special flash chips

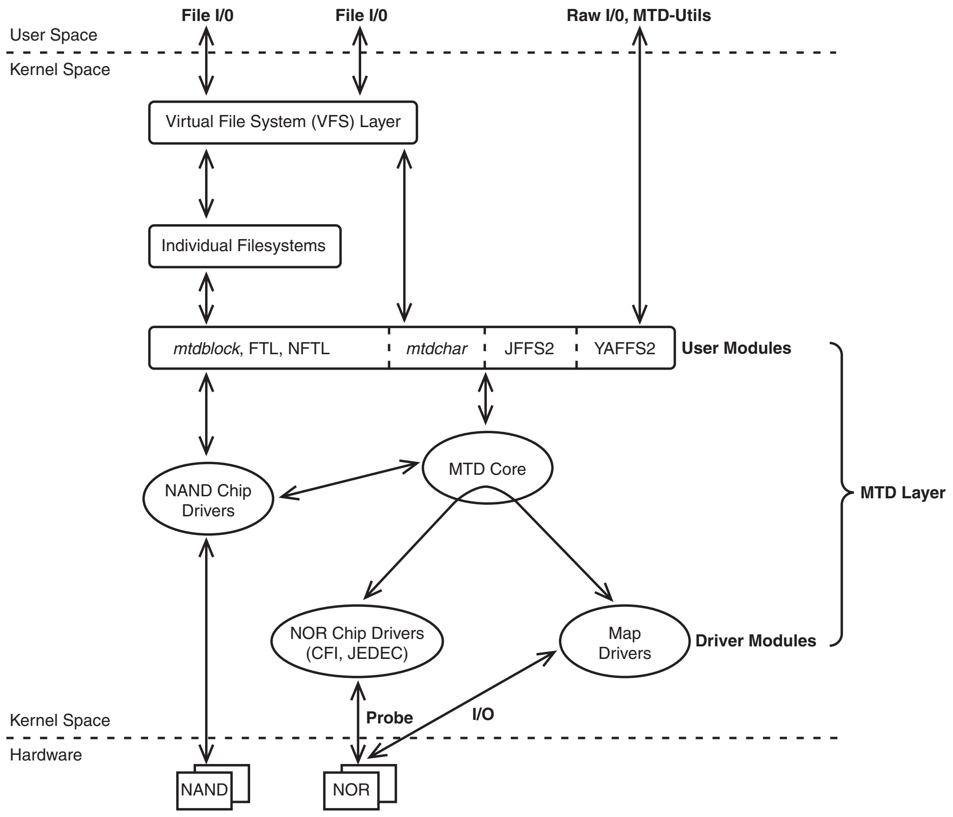


FIGURE 17.1 The Linux-MTD subsystem.

Map Drivers

To MTD-enable your device, your first task is to tell MTD how to access the flash device. For this, you have to map your flash memory range for CPU access and provide methods to operate on the flash. The next task is to inform MTD about the different storage partitions residing on your flash. Unlike hard disks on PC-compatible systems, flash-based storage does not contain a standard partition table on the media. Because of this, disk-partitioning tools such as *fdisk* and *cfdisk*[2] cannot be used to partition flash devices. Instead, partitioning information has to be implemented as part of kernel code.[3] These tasks are accomplished with the help of an MTD *map driver.*

To better understand the function of map drivers, let's look at an example.

Device Example: Handheld

Consider the Linux handheld shown in Figure 17.2. The flash has a size of 32MB and is mapped to `0xC0000000` in the processor's address space. It contains three partitions, one each for the bootloader, the kernel, and the root filesystem. The bootloader partition starts from the top of the flash, the kernel partition begins at offset `MY_KERNEL_START`, and the root filesystem starts at offset `MY_FS_START`.[4] The bootloader and the kernel reside on *read-only* partitions to avoid unexpected damage, while the filesystem partition is flagged *read-write.*

Let's first create the flash map and then proceed with the driver initialization. The map driver has to translate the flash layout shown in the figure to an `mtd_partition` structure. Listing 17.1 contains the `mtd_partition` definition corresponding to Figure 17.2. Note that the `mask_flags` field holds the permissions to be masked, so `MTD_WRITEABLE` implies a read-only partition.

LISTING 17.1 Creating an MTD Partition Map

```
#define FLASH_START          0x00000000
#define MY_KERNEL_START      0x00080000 /* 512K for bootloader */
#define MY_FS_START          0x00280000 /* 2MB for kernel */
#define FLASH_END            0x02000000 /* 32MB */
```

[2] *Fdisk* and *cfdisk* are used to manipulate the partition table residing in the first hard disk sector on PC systems.

[3] You may also pass partitioning information to MTD via the kernel command line argument `mtdpart=`, if you enable `CONFIG_MTD_CMDLINE_PARTS` during kernel configuration. Look at *drivers/mtd/cmdlinepart.c* for the usage syntax.

[4] Some devices have additional partitions for bootloader parameters, extra filesystems, and recovery kernels.

```
static struct mtd_partition pda_partitions[] = {
  {
    .name       = "pda_btldr",        /* This string is used by
                                         /proc/mtd to identify
                                         the bootloader partition */
    .size:      = (MY_KERNEL_START-FLASH_START),
    .offset     = FLASH_START,        /* Start from top of flash */
    .mask_flags = MTD_WRITEABLE       /* Read-only partition */
  },
  {
    .name       = "pda_krnl",         /* Kernel partition */
    .size:      = (MY_FS_START-MY_KERNEL_START),
    .offset     = MTDPART_OFS_APPEND, /* Start immediately after
                                         the bootloader partition */
    .mask_flags = MTD_WRITEABLE       /* Read-only partition */
  },
  {
    .name:      = "pda_fs",           /* Filesystem partition */
    .size:      = MTDPART_SIZ_FULL,   /* Use up the rest of the
                                         flash */
    .offset     = MTDPART_OFS_NEXTBLK,/* Align this partition with
                                         the erase size */
  }
};
```

Listing 17.1 uses `MTDPART_OFS_APPEND` to start a partition adjacent to the previous one. The start addresses of writeable partitions, however, need to be aligned with the erase/sector size of the flash chip. To achieve this, the filesystem partition uses `MTD_OFS_NEXTBLK` rather than `MTD_OFS_APPEND`.

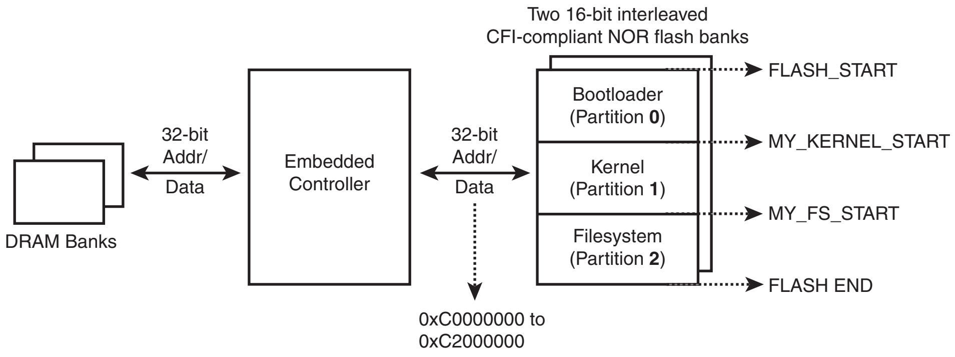


FIGURE 17.2 Flash Memory on a sample Linux handheld.

Now that you have populated the `mtd_partition` structure, let's proceed and complete a basic map driver for the example handheld. Listing 17.2 registers the map driver with the MTD core. It's implemented as a platform driver, assuming that your architecture-specific code registers an associated platform device having the same name. Rewind to the section "Device Example: Cell Phone" in Chapter 6, "Serial Drivers," for a discussion on platform devices and platform drivers. The `platform_device` is defined by the associated architecture-specific code as follows:

```
struct resource pda_flash_resource = { /* Used by Listing 17.3 */
  .start = 0xC0000000,                 /* Physical start of the
                                          flash in Figure 17.2 */
  .end   = 0xC0000000+0x02000000-1,    /* Physical end of flash */
  .flags = IORESOURCE_MEM,             /* Memory resource */
};
struct platform_device pda_platform_device = {
  .name = "pda",                   /* Platform device name */
  .id   = 0,                       /* Instance number */
  /* ... */
  .resource = &pda_flash_resource, /* See above */
};
platform_device_register(&pda_platform_device);
```

LISTING 17.2 Registering the Map Driver

```
static struct platform_driver pda_map_driver = {
  .driver = {
    .name     =  "pda",          /* ID */
   },
  .probe      =  pda_mtd_probe, /* Probe */
  .remove     =  NULL,          /* Release */
  .suspend    =  NULL,          /* Power management */
  .resume     =  NULL,          /* Power management */
};

/* Driver/module Initialization */
static int __init pda_mtd_init(void)
{
  return platform_driver_register(&pda_map_driver);
}

/* Module Exit */
```

```
static int __init pda_mtd_exit(void)
{
  return platform_driver_uregister(&pda_map_driver);
}
```

Because the kernel finds that the name of the platform driver registered in Listing 17.2 matches with that of an already-registered platform device, it invokes the probe method, `pda_mtd_probe()`, shown in Listing 17.3. This routine

- Reserves the flash memory address range using `request_mem_region()`, and obtains CPU access to that memory using `ioremap_nocache()`. You learned how to do this in Chapter 10, "Peripheral Component Interconnect."
- Populates a `map_info` structure (discussed next) with information such as the start address and size of flash memory. The information in this structure is used while performing the probing in the next step.
- Probes the flash via a suitable MTD chip driver (discussed in the next section). Only the chip driver knows how to query the chip and elicit the command-set required to access it. The chip layer tries different permutations of bus widths and interleaves while querying. In Figure 17.2, two 16-bit flash banks are connected in parallel to fill the 32-bit processor bus width, so you have a two-way interleave.
- Registers the `mtd_partition` structure that you populated earlier, with the MTD core.

Before looking at Listing 17.3, let's meet the `map_info` structure. It contains the address, size, and width of the flash memory and routines to access it:

```
struct map_info {
  char * name;                /* Name */
  unsigned long size;         /* Flash size */
  int bankwidth;              /* In bytes */
  /* ... */
  /* You need to implement custom routines for the following methods
     only if you have special needs. Else populate them with built-
     in methods using simple_map_init() as done in Listing 17.3 */
  map_word (*read)(struct map_info *, unsigned long);
  void     (*write)(struct map_info *, const map_word,
                    unsigned long);
  /* ... */
};
```

While we are in the topic of accessing flash chips, let's briefly revisit memory barriers that we discussed in Chapter 4, "Laying the Groundwork." An instruction reordering that appears semantically unchanged to the compiler (or the processor) may not be so in reality, so the ordering of data operations on flash memory is best left alone. You don't want to, for example, end up erasing a flash sector after writing to it, instead of doing the reverse. Also, the same flash chips, and hence their device drivers, are used on diverse embedded processors having different instruction reordering algorithms. For these reasons, MTD drivers are notable users of hardware memory barriers. `simple_map_write()`, a generic routine available to map drivers for use as the `write()` method in the `map_info` structure previously listed, inserts a call to `mb()` before returning. This ensures that the processor does not reorder flash reads or writes across the barrier.

LISTING 17.3 Map Driver Probe Method

```
#include <linux/mtd/mtd.h>
#include <linux/mtd/map.h>
#include <linux/ioport.h>

static int
pda_mtd_probe(struct platform_device *pdev)
{
  struct map_info *pda_map;
  struct mtd_info *pda_mtd;
  struct resource *res = pdev->resource;

  /* Populate pda_map with information obtained
     from the associated platform device */
  pda_map->virt = ioremap_nocache(res->start,
                                (res->end - res->start + 1));
  pda_map->name = pdev->dev.bus_id;
  pda_map->phys = res->start;
  pda_map->size = res->end - res->start + 1;
  pda_map->bankwidth = 2;     /* Two 16-bit banks sitting
                                 on a 32-bit bus */
  simple_map_init(&pda_map);  /* Fill in default access methods */

  /* Probe via the CFI chip driver */
  pda_mtd = do_map_probe("cfi_probe", &pda_map);
  /* Register the mtd_partition structure */
  add_mtd_partitions(pda_mtd, pda_partitions, 3); /* Three Partitions */

  /* ... */
}
```

Don't worry if the CFI probing done in Listing 17.3 seems esoteric. It's discussed in the next section when we look at NOR chip drivers.

MTD now knows how your flash device is organized and how to access it. When you boot the kernel with your map driver compiled in, user-space applications can respectively see your bootloader, kernel, and filesystem partitions as */dev/mtd/0*, */dev/mtd/1*, and */dev/mtd/2*. So, to test drive a new kernel image on the handheld, you can do this:

```
bash> dd if=zImage.new of=/dev/mtd/1
```

Flash Partitioning from Bootloaders

The Redboot bootloader maintains a partition table that holds flash layout, so if you are using Redboot on your embedded device, you can configure your flash partitions in the bootloader instead of writing an MTD map driver. To ask MTD to parse flash mapping information from Redboot's partition table, turn on `CONFIG_MTD_REDBOOT_PARTS` during kernel configuration.

NOR Chip Drivers

As you might have noticed, the NOR flash chip used by the handheld in Figure 17.2 is labeled *CFI-compliant.* CFI stands for *Common Flash Interface,* a specification designed to do away with the need for developing separate drivers to support chips from different vendors. Software can query CFI-compliant flash chips and automatically detect block sizes, timing parameters, and the command-set to be used for communication. Drivers that implement specifications such as CFI and JEDEC are called *chip drivers.*

According to the CFI specification, software must write `0x98` to location `0x55` within flash memory to initiate a query. Look at Listing 17.4 to see how MTD implements CFI query.

LISTING 17.4 Querying CFI-compliant Flash

```
/* Snippet from cfi_probe_chip() (2.6.23.1 kernel) defined in
   drivers/mtd/chips/cfi_probe.c, with comments added */

/* cfi is a pointer to struct cfi_private defined in
   include/linux/mtd/cfi.h */

/* ... */

/* Ask the device to enter query mode by sending
   0x98 to offset 0x55 */
cfi_send_gen_cmd(0x98, 0x55, base, map, cfi,
                 cfi->device_type, NULL);

/* If the device did not return the ASCII characters
   'Q', 'R' and 'Y', the chip is not CFI-compliant */
if (!qry_present(map, base, cfi)) {
  xip_enable(base, map, cfi);
  return 0;
}

/* Elicit chip parameters and the command-set, and populate
   the cfi structure */
if (!cfi->numchips) {
  return cfi_chip_setup(map, cfi);
}
/* ... */
```

The CFI specification defines various command-sets that compliant chips can implement. Some of the common ones are as follows:

- Command-set `0001`, supported by Intel and Sharp flash chips
- Command-set `0002`, implemented on AMD and Fujitsu flash chips
- Command-set `0020`, used on ST flash chips

MTD supports these command-sets as kernel modules. You can enable the one supported by your flash chip via the kernel configuration menu.

NAND Chip Drivers

NAND technology users such as USB pen drives, DOMs, Compact Flash memory, and SD/MMC cards emulate standard storage interfaces such as SCSI or IDE over NAND flash, so you don't need to develop NAND drivers to communicate with them.[5] On-board NAND flash chips need special drivers, however, and are the topic of this section.

As you learned previously in this chapter, NAND flash chips, unlike their NOR counterparts, are not connected to the CPU via data and address lines. They interface to the CPU through special electronics called a *NAND flash controller* that is part of many embedded processors. To read data from NAND flash, the CPU issues an appropriate *read* command to the NAND controller. The controller transfers data from the requested flash location to an internal RAM memory, also part of the controller. The data transfer is done in units of the flash chip's *page size* (for example, 2KB). In general, the denser the flash chip, the larger is its page size. Note that the page size is different from the flash chip's *block size*, which is the minimum erasable flash memory unit (for example, 16KB). After the transfer operation completes, the CPU reads the requested NAND contents from the internal RAM. Writes to NAND flash are done similarly, except that the controller transfers data from the internal RAM to flash. The connection diagram of NAND flash memory on an embedded device is shown in Figure 17.3.

Because of this unconventional mode of addressing, you need special drivers to work with NAND storage. MTD provides such drivers to manage NAND-resident data. If you are using a supported chip, you have to enable only the appropriate low-level MTD NAND driver. If you are writing a NAND flash driver, however, you need to explore two datasheets: the NAND flash controller and the NAND flash chip.

NAND flash chips do not support automatic configuration using protocols such as CFI. You have to manually inform MTD about the properties of your NAND chip by adding an entry to the `nand_flash_ids[]` table defined in *drivers/mtd/nand/nand_ids.c*. Each entry in the table consists of an identifier name, the device ID, page size, erase block size, chip size, and options such as the bus width.

[5] Unless you are writing drivers for the storage media itself. If you are embedding Linux on a device that will export part of its NAND partition to the outside world as a USB mass storage device, you do have to contend with NAND drivers.

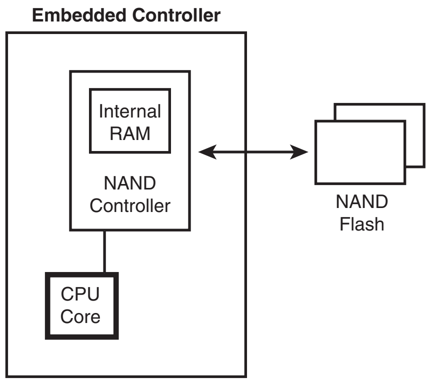


FIGURE 17.3 NAND flash connection.

There is another characteristic that goes hand in hand with NAND memory. NAND flash chips, unlike NOR chips, are not faultless. It's normal to have some problem bits and bad blocks scattered across NAND flash regions. To handle this, NAND devices associate a *spare area* with each flash page (for example, 64 bytes of spare area for each 2KB data page). The spare area contains *out-of-band* (OOB) information to help perform bad block management and error correction. The OOB area includes *error correcting codes* (ECCs) to implement error correction and detection. ECC algorithms correct single-bit errors and detect multibit errors. The `nand_ecclayout` structure defined in *include/mtd/mtd-abi.h* specifies the layout of the OOB spare area:

```
struct nand_ecclayout {
  uint 32_t eccbytes;
  uint32_t  eccpos[64];
  uint32_t  oobavail;
  struct    nand_oobfree oobfree[MTD_MAX_OOBFREE_ENTRIES];
};
```

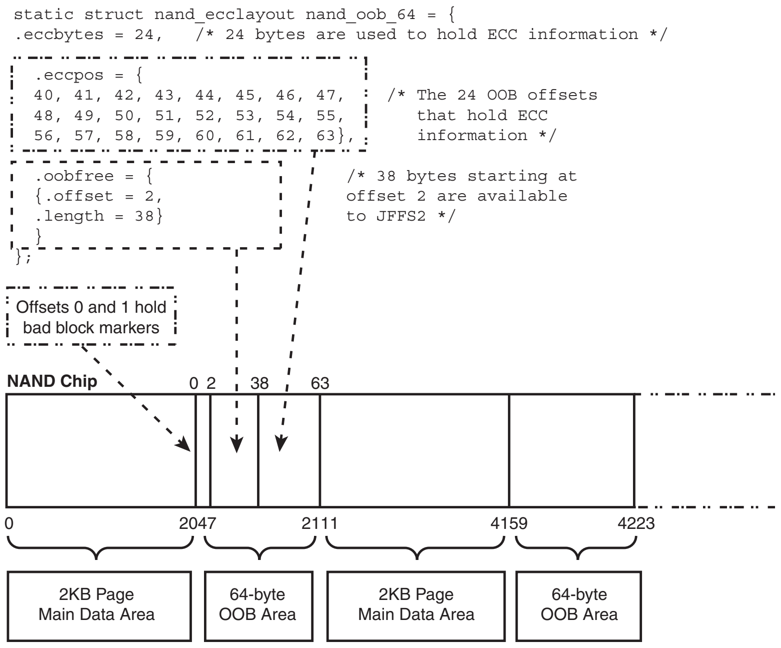
In this structure, `eccbytes` holds the number of OOB bytes that store ECC data, and `eccpos` is an array of offsets into the OOB area that contains the ECC data. `oobfree` records the unused bytes in the OOB area available to flash filesystems for storing flags such as *clean markers* that signal successful completion of erase operations.

Individual NAND drivers initialize their `nand_ecclayout` according to the chip's properties. Figure 17.4 illustrates the layout of a NAND flash chip having a page size of 2KB. The OOB semantics used by the figure is the default for 2KB page-sized chips as defined in the generic NAND driver, *drivers/mtd/nand/nand_base.c*.

Often, the NAND controller performs error correction and detection in hardware by operating on the ECC fields in the OOB area. If your NAND controller does not support error management, however, you will need to get MTD to do that for you in software. The MTD *nand_ecc* driver (*drivers/mtd/nand/nand_ecc.c*) implements software ECC.

Figure 17.4 also shows OOB memory bytes that contain bad block markers. These markers are used to flag faulty flash blocks and are usually present in the OOB region belonging to the first page of each block. The position of the marker inside the OOB area depends on the properties of the chip. Bad block markers are either set at the factory during manufacture, or by software when it detects wear in a block. MTD implements bad block management in *drivers/mtd/nand/nand_bbt.c*.

The `mtd_partition` structure used in Listing 17.1 for the NOR flash in Figure 17.2 works for NAND memory, too. After you MTD-enable your NAND flash, you can access the constituent partitions using standard device nodes such as */dev/mtd/X* and */dev/mtdblock/X*. If you have a mix of NOR and NAND memories on your hardware, *X* can be either a NOR or a NAND partition. If you have a total of more than 32 flash partitions, accordingly change the value of `MAX_MTD_DEVICES` in *include/linux/mtd/mtd.h*.



```
static struct nand_ecclayout nand_oob_64 = {
.eccbytes = 24,   /* 24 bytes are used to hold ECC information */
  .eccpos = {
  40, 41, 42, 43, 44, 45, 46, 47,    /* The 24 OOB offsets
  48, 49, 50, 51, 52, 53, 54, 55,       that hold ECC
  56, 57, 58, 59, 60, 61, 62, 63},      information */
  .oobfree = {                      /* 38 bytes starting at
  {.offset = 2,                        offset 2 are available
  .length = 38}                        to JFFS2 */
  }
};
```

FIGURE 17.4 Layout of a NAND flash chip.

To effectively make use of NAND storage, you need to use a filesystem tuned for NAND access, such as JFFS2 or YAFFS2, in tandem with the low-level NAND driver. We discuss these filesystems in the next section.

User Modules

After you have added a map driver and chosen the right chip driver, you're all set to let higher layers use the flash. User-space applications that perform file I/O need to view the flash device as if it were a disk, whereas programs that desire to accomplish raw I/O access the flash as if it were a character device. The MTD layer that achieves these and more is called *User Modules*, as shown in Figure 17.1. Let's look at the components constituting this layer.

Block Device Emulation

The MTD subsystem provides a block driver called *mtdblock* that emulates a hard disk over flash memory. You can put any filesystem, say EXT2, over the emulated flash disk. Mtdblock hides complicated flash access procedures (such as preceding a write with an erase of the corresponding sector) from the filesystem. Device nodes created by mtdblock are named */dev/mtdblock/X,* where *X* is the partition number. To create an EXT2 filesystem on the pda_fs partition of the handheld, as shown in Figure 17.2, do the following:

```
bash> mkfs.ext2 /dev/mtdblock/2     → Create an EXT2 filesystem
                                      on the second partition
bash> mount /dev/mtdblock/2 /mnt    → Mount the partition
```

As you will soon see, it's a much better idea to use JFFS2 rather than EXT2 to hold files on flash filesystem partitions.

The *File Translation Layer* (FTL) and the *NAND File Translation Layer* (NFTL) perform a transformation called *wear leveling.* Flash memory sectors can withstand only a finite number of erase operations (in the order of 100,000). Wear leveling prolongs flash life by distributing memory usage across the chip. Both FTL and NFTL provide device interfaces similar to *mtdblock* over which you can put normal filesystems. The corresponding device nodes are named */dev/nftl/X,* where *X* is the partition number. Certain algorithms used in these modules are patented, so there could be restrictions on usage.

Char Device Emulation

The mtdchar driver presents a linear view of the underlying flash device, rather than the block-oriented view required by filesystems. Device nodes created by mtdchar are named */dev/mtd/X,* where *X* is the partition number. You may update the bootloader partition of the handheld as shown in Figure 17.2, by using dd over the corresponding mtdchar interface:

```
bash> dd if=bootloader.bin of=/dev/mtd/0
```

An example use of a raw mtdchar partition is to hold POST error logs generated by the bootloader on an embedded device. Another use of a char flash partition on an embedded system is to store information similar to that present in the CMOS or the EEPROM on PC-compatible systems. This includes the boot order, power-on password, and *Vital Product Data* (VPD) such as the device serial number and model number.

JFFS2

Journaling Flash File System (JFFS) is considered the best-suited filesystem for flash memory. Currently, version 2 (JFFS2) is in use, and JFFS3 is under development. JFFS was originally written for NOR flash chips, but support for NAND devices is merged with the 2.6 kernel.

Normal Linux filesystems are designed for desktop computers that are shut down gracefully. JFFS2 is designed for embedded systems where power failure can occur abruptly, and where the storage device can tolerate only a finite number of erases. During flash erase operations, current sector contents are saved in RAM. If there is a power loss during the slow erase process, entire contents of that sector can get lost. JFFS2 circumvents this problem using a log-structured design. New data is appended to a log that lives in an erased region. Each JFFS2 node contains metadata to track disjoint file locations. Memory is periodically reclaimed using garbage collection. Because of this design, flash writes do not have to go through a save-erase-write cycle, and this improves power-down reliability. The log-structure also increases flash life span by spreading out writes.

To create a JFFS2 image of a tree living under */path/to/filesystem/* on a flash chip having an erase size of 256KB, use *mkfs.jffs2* as follows:

```
bash> mkfs.jffs2 -e 256KiB –r /path/to/filesystem/ -o jffs2.img
```

JFFS2 includes a *garbage collector* (GC) that reclaims flash regions that are no longer in use. The garbage collection algorithm depends on the erase size, so supplying an accurate value makes it more efficient. To obtain the erase size of your flash partitions, you may seek the help of */proc/mtd*. The output for the Linux handheld shown in Figure 17.2 is as follows:

```
bash> cat /proc/mtd
dev:    size    erasesize   name
mtd0: 00100000  00040000  "pda_btldr"
mtd1: 00200000  00040000  "pda_krnl"
mtd2: 01400000  00040000  "pda_fs"
```

JFFS2 supports compression. Enable appropriate options under `CONFIG_JFFS2_COMPRESSION_OPTIONS` to choose available compressors, and look at *fs/jffs2/compr*.c* for their implementations.

Note that JFFS2 filesystem images are usually created on the host machine where you do cross-development and then transferred to the desired flash partition on the target device via a suitable download mechanism such as serial port, USB, or NFS. More on this in Chapter 18, "Embedding Linux."

YAFFS2

The implementation of JFFS2 in the 2.6 kernel includes features to work with the limitations of NAND flash, but *Yet Another Flash File System* (YAFFS) is a filesystem that is designed to function under constraints specific to NAND memory. YAFFS is not part of the mainline kernel, but some embedded distributions prepatch their kernels with support for YAFFS2, the current version of YAFFS.

You can download YAFFS2 source code and documentation from www.yaffs.net.

MTD-Utils

The MTD-utils package, downloadable from ftp://ftp.infradead.org/pub/mtd-utils/, contains several useful tools that work on top of MTD-enabled flash memory. Examples of included utilities are *flash_eraseall*, *nanddump*, *nandwrite*, and *sumtool*.

To erase the second flash partition (on NOR or NAND devices), use flash_eraseall as follows:

```
bash> flash_eraseall -j /dev/mtd/2
```

Because NAND chips may contain bad blocks, use ECC-aware programs such as nandwrite and nanddump to copy raw data, instead of general-purpose utilities, such as dd. To store the JFFS2 image that you created previously, on to the second NAND partition, do this:

```
bash> nandwrite /dev/mtd/2 jffs2.img
```

You can reduce JFFS2 mount times by inserting summary information into a JFFS2 image using sumtool and turning on `CONFIG_JFFS2_SUMMARY` while configuring your kernel. To write a summarized JFFS2 image to the previous NAND flash, do this:

```
bash> sumtool -e 256KiB -i jffs2.img -o jffs2.summary.img
bash> nandwrite /dev/mtd/2 jffs2.summary.img
bash> mount -t jffs2 /dev/mtdblock/2 /mnt
```

Configuring MTD

To MTD-enable your kernel, you have to choose the appropriate configuration options. For the flash chip shown in Figure 17.2, the required options are as follows:

```
CONFIG_MTD=y                 → Enable the MTD subsystem
CONFIG_MTD_PARTITIONS=y      → Support for multiple partitions
CONFIG_MTD_GEN_PROBE=y       → Common routines for chip probing
CONFIG_MTD_CFI=y             → Enable CFI chip driver
CONFIG_MTD_PDA_MAP=y         → Option to enable the map driver
CONFIG_JFFS2_FS=y            → Enable JFFS2
```

`CONFIG_MTD_PDA_MAP` is assumed to be a new option added to enable the map driver we previously wrote. Each of these features can also be built as a kernel module unless you have an MTD-resident root filesystem. To mount the filesystem partition in Figure 17.2 as the root device during boot, ask your bootloader to append `root=/dev/mtdblock/2` to the command-line string that it passes to the kernel.

You may reduce kernel footprint by eliminating redundant probing. Because our example handheld has two parallel 16-bit banks sitting on a 32-bit physical bus (thus resulting in a two-way interleave and a 2-byte bank width), you can optimize using these additional options:

```
CONFIG_MTD_CFI_ADV_OPTIONS=y
CONFIG_MTD_CFI_GEOMETRY=y
CONFIG_MTD_MAP_BANK_WIDTH_2=y
CONFIG_MTD_CFI_I2=y
```

`CONFIG_MTD_MAP_BANK_WIDTH_2` enables a CFI bus width of 2, and `CONFIG_MTD_CFI_I2` sets an interleave of 2.

eXecute In Place

With *eXecute In Place* (XIP), you can run the kernel directly from flash. Because you do away with the extra step of copying the kernel to RAM, your kernel boots faster. The downside is that your flash memory requirement increases because the kernel has to be stored uncompressed. Before deciding to go the XIP route, also be aware that the slower instruction fetch times from flash can impact runtime performance.

The Firmware Hub

PC-compatible systems use a NOR flash chip called the *Firmware Hub* (FWH) to hold the BIOS. The FWH is not directly connected to the processor's address and data bus. Instead, it's interfaced via the *Low Pin Count* (LPC) bus, which is part of South Bridge chipsets. The connection diagram is shown in Figure 17.5.

The MTD subsystem includes drivers to interface the processor with the FWH. FWHs are usually not compliant with the CFI specification. Instead, they conform to the JEDEC (*Joint Electron Device Engineering Council*) standard. To inform MTD about a yet unsupported JEDEC chip, add an entry to the `jedec_table` array in *drivers/mtd/chips/jedec_probe.c* with information such as the chip manufacturer ID and the command-set ID. Here is an example:

```
static const struct amd_flash_info jedec_table[] = {
  /* ... */
  {
    .mfr_id  = MANUFACTURER_ID, /* E.g.: MANUFACTURER_ST */
    .dev_id  = DEVICE_ID,       /* E.g.: M50FW080 */
    .name    = "MYNAME",        /* E.g.: "M50FW080" */
    .uaddr   = {
      [0] = MTD_UADDR_UNNECESSARY,
    },
    .DevSize  = SIZE_1MiB,    /* E.g.: 1MB */
    .CmdSet   = CMDSET,       /* Command-set to communicate with the
                                 flash chip e.g., P_ID_INTEL_EXT */
    .NumEraseRegions = 1,     /* One region */
    .regions  = {
      ERASEINFO (0x10000, 16),/* Sixteen 64K sectors */
    }
  },
  /* ... */
};
```

When you have your chip details imprinted in the `jedec_table` as shown here, MTD should recognize your flash, provided you have enabled the right kernel configuration options. The following configuration makes the kernel aware of an FWH that interfaces to the processor via an Intel ICH2 or ICH4 South Bridge chipset:

```
CONFIG_MTD=y                 → Enable the MTD subsystem
CONFIG_MTD_GEN_PROBE=y       → Common routines for chip probing
CONFIG_MTD_JEDECPROBE=y      → JEDEC chip driver
CONFIG_MTD_CFI_INTELEXT=y    → The command-set for communicating
                               with the chip
CONFIG_MTD_ICHXROM=y         → The map driver
```

`CONFIG_MTD_JEDECPROBE` enables the JEDEC MTD chip driver, and `CONFIG_MTD_ICH2ROM` adds the MTD map driver that maps the FWH to the processor's address space. In addition, you need to include the appropriate command-set implementation (for example, `CONFIG_MTD_CFI_INTELEXT` for Intel Extension commands).

After these modules have been loaded, you can talk to the FWH from user-space applications via device nodes exported by MTD. You can, for example, reprogram the BIOS from user space using a simple application, as shown in Listing 17.5. Be warned that incorrectly operating this program can corrupt the BIOS and render your system unbootable!

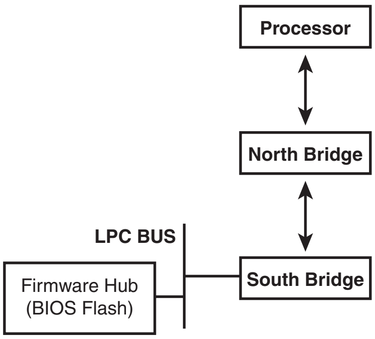


FIGURE 17.5 The Firmware Hub on a PC-compatible system.

Listing 17.5 operates on the MTD char device associated with the FWH, which it assumes to be */dev/mtd/0*. The program issues three MTD-specific *ioctl* commands:

- MEMUNLOCK to unlock the flash sectors prior to programming
- MEMERASE to erase flash sectors prior to rewriting
- MEMLOCK to relock the sectors after programming

LISTING 17.5 Updating the BIOS

```
#include <linux/mtd/mtd.h>
#include <stdio.h>
#include <fcntl.h>
#include <asm/ioctl.h>
#include <signal.h>
#include <sys/stat.h>

#define BLOCK_SIZE    4096
#define NUM_SECTORS   16
#define SECTOR_SIZE   64*1024

int
main(int argc, char *argv[])
{
  int fwh_fd, image_fd;
  int usect=0, lsect=0, ret;
  struct erase_info_user fwh_erase_info;
  char buffer[BLOCK_SIZE];
  struct stat statb;
```

```
/* Ignore SIGINTR(^C) and SIGSTOP (^Z), lest
   you end up with a corrupted flash and an
   unbootable system */
sigignore(SIGINT);
sigignore(SIGTSTP);

/* Open MTD char device */
fwh_fd = open("/dev/mtd/0", O_RDWR);
if (fwh_fd < 0) exit(1);

/* Open BIOS image */
image_fd = open("bios.img", O_RDONLY);
if (image_fd < 0) exit(2);

/* Sanity check */
fstat(image_fd, &statb);
if (statb.st_size != SECTOR_SIZE*NUM_SECTORS) {
  printf("BIOS image looks bad, exiting.\n");
  exit(3);
}

/* Unlock and erase all sectors */
while (usect < NUM_SECTORS) {
  printf("Unlocking & Erasing Sector[%d]\r", usect+1);

  fwh_erase_info.start = usect*SECTOR_SIZE;
  fwh_erase_info.length = SECTOR_SIZE;

  ret = ioctl(fwh_fd, MEMUNLOCK, &fwh_erase_info);
  if (ret != 0) goto bios_done;

  ret = ioctl(fwh_fd, MEMERASE, &fwh_erase_info);
  if (ret != 0) goto bios_done;
  usect++;
}

 /* Read blocks from the BIOS image and dump it to the
    Firmware Hub */
while ((ret = read(image_fd, buffer, BLOCK_SIZE)) != 0) {
  if (ret < 0) goto bios_done;
  ret = write(fwh_fd, buffer, ret);
  if (ret <= 0) goto bios_done;
}
```

```
  /* Verify by reading blocks from the BIOS flash and comparing
     with the image file */

  /* ... */

 bios_done:

  /* Lock back the unlocked sectors */
  while (lsect < usect) {
    printf("Relocking Sector[%d]\r", lsect+1);

    fwh_erase_info.start  = lsect*SECTOR_SIZE;
    fwh_erase_info.length = SECTOR_SIZE;

    ret = ioctl(fwh_fd, MEMLOCK, &fwh_erase_info);
    if (ret != 0) printf("Relock failed on sector %d!\n", lsect);
    lsect++;
  }

  close(image_fd);
  close(fwh_fd);

}
```

Debugging

To debug flash-related problems, enable `CONFIG_MTD_DEBUG` (*Device Drivers → Memory Technology Devices → Debugging*) during kernel configuration. You can further tune the debug verbosity level to between 0 and 3.

The Linux-MTD project page www.linux-mtd.infradead.org has FAQs, various pieces of documentation, and a *Linux-MTD JFFS HOWTO* that provides insights into JFFS2 design. The linux-mtd mailing list is the place to discuss questions related to MTD device drivers. Look at http://lists.infradead.org/pipermail/linux-mtd/ for the mailing list archives.

Looking at the Sources

In the kernel tree, the *drivers/mtd/* directory contains the sources for the MTD layer. Map, chip, and NAND drivers live in the *drivers/mtd/maps/*, *drivers/mtd/chips/*, and *drivers/mtd/nand/* subdirectories, respectively. Most MTD data structures are defined in header files present in *include/linux/mtd/.*

To access an unsupported BIOS firmware hub from Linux, implement a driver using *drivers/mtd/maps/ichxrom.c* as your starting point.

For examples of operating on NAND OOB data from user space, look at *nanddump.c* and *nandwrite.c* in the MTD-utils package.

Table 17.1contains the main data structures used in this chapter and their location in the source tree. Table 17.2 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

TABLE 17.1 Summary of Data Structures

Data Structure	Location	Description
`mtd_partition`	*include/linux/mtd/partitions.h*	Representation of a flash chip's partition layout.
`map_info`	*include/linux/mtd/map.h*	Low-level access routines implemented by the map driver are passed to the chip driver using this structure.
`mtd_info`	*include/linux/mtd/mtd.h*	General device-specific information.
`erase_info`, `erase_info_user`	*include/linux/mtd/mtd.h, include/mtd/mtd-abi.h*	Structures used for flash erase management.
`cfi_private`	*include/linux/mtd/cfi.h*	Device-specific information maintained by NOR chip drivers.
`amd_flash_info`	*drivers/mtd/chips/jedec_probe.c*	Device-specific information supplied to the JEDEC chip driver.
`nand_ecclayout`	*include/mtd/mtd-abi.h*	Layout of the OOB spare area of a NAND chip.

TABLE 17.2 Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`simple_map_init()`	*drivers/mtd/maps/map_funcs.c*	Initializes a `map_info` structure with generic flash access methods
`do_map_probe()`	*drivers/mtd/chips/chipreg.c*	Probes the NOR flash via a chip driver
`add_mtd_partitions()`	*drivers/mtd/mtdpart.c*	Registers an `mtd_partition` structure with the MTD core

Chapter 18

Embedding Linux

In This Chapter

Linux is making inroads into industry domains such as consumer electronics, telecom, networking, defense, and health care. With its popularity surging in the embedded space, it's more likely that you will use your Linux device driver skills to enable embedded devices rather than legacy systems. In this chapter, let's enter the world of embedded Linux wearing the lens of a device driver developer. Let's look at the software components of a typical embedded Linux solution and see how the device classes that you saw in the previous chapters tie in with common embedded hardware.

Challenges

Embedded systems present several significant software challenges:

- Embedded software has to be cross-compiled and then downloaded to the target device to be tested and verified.
- Embedded systems, unlike PC-compatible computers, do not have fast processors, fat caches, and wholesome storage.
- It's often difficult to get mature development and debug tools for embedded hardware for free.
- The Linux community has a lot more experience on the x86 platform, so you are less likely to get instant online help from experts if you working on embedded computers.
- The hardware evolves in stages. You may have to start software development on a proof-of-concept prototype or a reference board, and progressively move on to engineering-level debug hardware and a few passes of production-level units.

All these result in a longer development cycle.

From a device-driver perspective, embedded software developers often face interfaces not commonly found on conventional computers. Figure 18.1 (which is an expanded version of Figure 4.2 in Chapter 4, "Laying the Groundwork") shows a hypothetical embedded device that could be a handheld, smart phone, *point-of-sale* (POS) terminal, kiosk, navigation system, gaming device, telemetry gadget on an automobile dashboard, IP phone, music player, digital set-top box, or even a pacemaker programmer. The device is built around an SoC and has some combination of flash memory, SDRAM, LCD, touch screen, USB OTG, serial ports, audio codec, connectivity, SD/MMC controller, Compact Flash, I^{2}C devices, SPI devices, JTAG, biometrics, smart card interfaces, keypad, LEDs, switches, and electronics specific to the industry domain. Modifying and debugging drivers for some of these devices can be tougher than usual: NAND flash drivers have to handle problems such as bad blocks and failed bits, unlike standard IDE storage drivers. Flash-based filesystems such as JFFS2, are more complex to debug than EXT2 or EXT3 filesystems. A USB OTG driver is more involved than a USB OHCI driver. The SPI subsystem on the kernel is not as mature as, say, the serial layer. Moreover, the industry domain using the embedded device might impose specific requirements such as quick response times or fast boot.

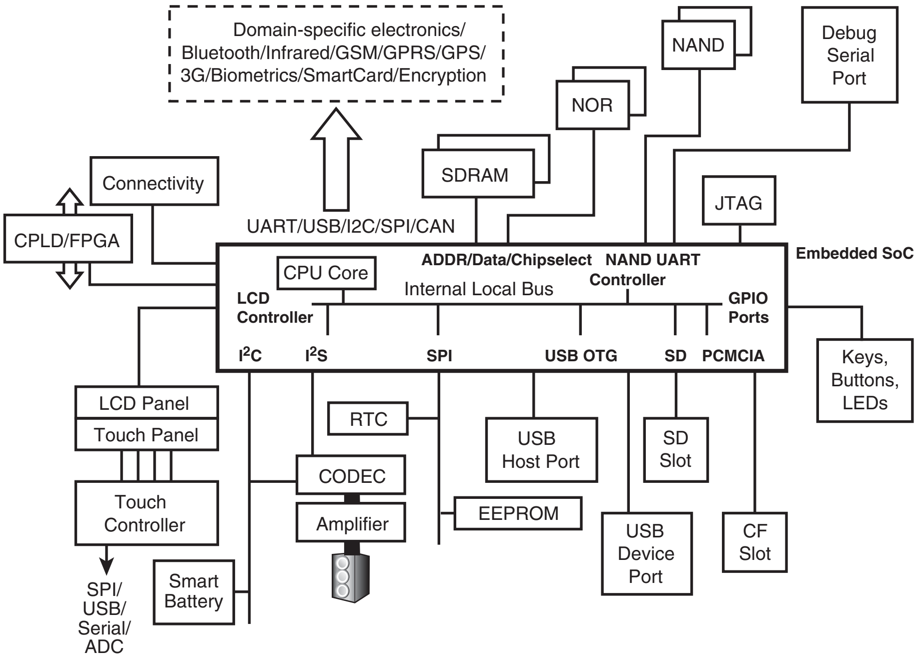


FIGURE 18.1 Block diagram of a hypothetical embedded device.

Component Selection

Evaluating and selecting components is one of the important tasks undertaken during the concept phase of a project. Look at the sidebar "Choosing a Processor and Peripherals" for some important factors that hardware designers and product managers consider while choosing components for building an embedded device. In today's world, where time to market is often the critical factor driving device design, the software engineer also has a considerable say in shaping component selection. Availability of a Linux distribution can influence processor choice, while existence of device drivers or close starting points can affect the choice of peripheral chipsets.

Although the kernel engineer needs to do due diligence and evaluate several Linux distributions (or even operating systems), he may nix a technologically superior distribution in favor of a familiar one if he believes that'll mitigate project risks. Or a preferred distribution might be the one that offers indemnification from lawsuits arising out of kernel bugs, if that is a crucial consideration in the relevant industry domain. The electrical engineer can limit evaluation to processors supported by the chosen distribution and prefer peripheral chipsets enabled by the distribution in question.

Choosing a Processor and Peripherals

Let's look at some common questions that electrical engineers and product managers ask when selecting components for an embedded device. Assume that a hypothetical processor *P* is on the shortlist because it satisfies basic product requirements such as power consumption and packaging. *P* and accompanying peripheral chipsets are under evaluation:

Performance: Is the processor frequency sufficient to drive target applications? If the embedded device intends to implement CPU-intensive tasks, does the MIPS budgeting for all software subsystems balance with the processor's MIPS rating? If the target device requires high-resolution imaging, for example, will the MHz impact of graphics manipulation drag down the performance of other subsystems, such as networking?

Cost: Will I save a buck on the component but end up spending two more on the surrounding electronics? For example, will *P* need an extra regulator? Will I need to throw in an additional accessory, for example, an RTC chip, because *P* does not have one built-in? Does *P* have more pins than other processors under evaluation leading to a denser board having a larger number of layers and vias that increase the raw board cost? Does *P* consume more power and generate more heat necessitating a bigger power supply and additional passive components? Is there errata in the data sheet that has the possibility of increasing software development costs?

Functionality: What's the maximum size of DRAM, SRAM, NOR, and NAND memory that *P* can address?

Business Planning: Does *P*'s vendor offer an upgrade path to a higher horsepower processor that is a drop-in (pin-compatible) replacement? Is the vendor company stable?

Supplier: Is this a single-source component? If so, is the supplier volatile? What are the lead times to procure the parts?

End-of-Life: Is *P* likely to go end-of-life before the expected lifespan of the embedded device?

Credibility: Is *P* an accepted component? Do peripheral chipsets under evaluation have an industry segment behind them? Perhaps a landscape LCD under consideration is being used on automobile dashboards?

Ruggedness: Need the components be MIL (military) or industrial grade?

One has to evaluate different candidates and figure out the sweet spot in terms of all these.

Tool Chains

Because the target device is unlikely to be binary-compatible with your host development platform, you have to cross-compile embedded software using *tool chains.* Setting up a full-fledged tool chain entails building the following:

1. The GNU C (cross-)Compiler. GCC supports all platforms that Linux runs on, but you have to configure and build it to generate code for your target architecture. Essentially, you have to compile the compiler and generate the appropriate cross-compiler.
2. *Glibc,* the set of C libraries that you will need when you build applications for the target device.
3. *Binutils,* which includes the cross-assembler, and tools such as *objdump.*

Getting a development tool chain in place used to be a daunting task several years ago but is usually straightforward today because Linux distributions offer precompiled binaries and easy-installation tools for a variety of architectures.

Embedded Bootloaders

Bootloader development is usually the starting point of any embedded software effort. You have to decide whether to write a bootloader from scratch or tailor an existing open source bootloader to suit your needs. Each candidate bootloader might be built based on a different philosophy: small footprint, easy portability, fast boot, or the capability to support certain specific features. After you home-in on a starting point, you can design and implement device-specific modifications.

In this section, let's use the term *bootloader* to mean the boot suite. This includes the following:

- The BIOS, if present
- Any bootstrap code needed to put the bootloader onto the boot device
- One or more stages[1] of the actual bootloader
- Any program executing on an external host machine that talks with the bootloader for the purpose of downloading firmware onto the target device

At the minimum, a bootloader is responsible for processor- and board-specific initializations, loading a kernel and an optional initial ramdisk into memory and passing control to the kernel. In addition, a bootloader might be in charge of providing BIOS services, performing POST, supporting firmware downloads to the target, and passing memory layout and configuration information to the kernel. On embedded devices that use encrypted firmware images for security reasons, bootloaders may have the task of decrypting firmware. Some bootloaders support a debug monitor to load and debug stand-alone code on to the target device. You may also decide to build a failure-recovery mechanism into your bootloader to recoup from kernel corruption on the field.

In general, bootloader architecture depends on the processor family, the chipsets present on the hardware platform, the boot device, and the operating system running on the device. To illustrate the effects of the processor family on the boot suite, consider the following:

- A bootloader for a device designed around the StrongARM processor has to know whether it's booting the system or waking it up from sleep, because the processor starts execution from the top of its address space (the bootloader) in both cases. The bootloader has to pass control to the kernel code that restores the system state if it's waking up from sleep or load the kernel from the boot device if the system is starting from reset.
- An x86 bootloader might need to switch to protected mode to load a kernel bigger than the 1MB real-mode limit.
- Embedded systems not based on x86 platforms cannot avail the services of a legacy BIOS. So, if you want your embedded device to boot, for example, from an external USB device, you have to build USB capabilities into your bootloader.

[1] In embedded bootloader parlance, the first stage of a two-stage bootloader is sometimes called the *Initial Program Loader* (IPL), and the second stage is called the *Secondary Program Loader* (SPL).

- Even when two platforms are based on similar processor cores, the bootloader architecture may differ based on the SoC. For example, consider two ARM-based devices, the Compaq iPAQ H3900 PDA and the Darwin Jukebox. The former is built around the Intel PXA250 controller chip, which has an XScale CPU based on an ARMv5 core, and the latter is designed using the Cirrus Logic EP7312 controller that uses an ARMv3 core. Whereas XScale supports JTAG (named after the *Joint Test Action Group,* which developed this hardware-assisted debugging standard) to load a bootloader onto flash, the EP7312 has a bootstrap mode to accomplish the same task.

The boot suite needs a mechanism to transfer a bootloader image from the host development system to the target's boot device. This is called *bootstrapping.* Bootstrapping is straightforward on PC-compatible systems where the BIOS flash is programmed using an external burner if it's corrupted or updated after booting into an operating system if it's healthy. Embedded devices, however, do not have a generic method for bootstrapping.

To illustrate bootstrapping on an embedded system, take the example of the Cirrus Logic EP7211 controller (which is the predecessor of the EP7312 discussed in the previous section). The EP7211 executes code from a small internal 128-byte memory when it's powered on in a bootstrap mode. This 128-byte code downloads a bootstrap image from a host via the serial port to an on-board 2KB SRAM and transfers control to it. The boot suite has to be thus architected into three stages, each loaded at a different address:

- The first stage (the 128-byte image) is part of processor firmware.
- The second stage lives in the on-chip SRAM, so it can be up to 2KB. This is the bootstrapper.
- The bootstrapper downloads the actual bootloader image from an external host to the top of flash memory. The bootloader gets control when the processor powers on in normal operation mode.

Note that the processor-resident microcode (the first stage) itself cannot function as the bootstrapper because a bootstrapper needs to have the capability to program flash memory. Because many types of flash chips can be used with a processor, the bootstrapper code needs to be board-specific.

Many controller chips do not support a bootstrap mode. Instead, the bootloader is written to flash via a JTAG interface. You can use your JTAG debugger's command interface to access the processor's debug logic and burn the bootloader to the target device's flash memory. We will have a more detailed discussion on JTAG debugging in the section "JTAG Debuggers" in Chapter 21, "Debugging Device Drivers."

There are controllers that support both bootstrap execution mode and JTAG. The Freescale i.MX21 (and its upgraded version i.MX27) based on an ARM9 core is one such controller.

After a bootloader is resident on flash, it can update itself as well as other firmware components such as the kernel and the root filesystem. The bootloader can directly talk to a host machine and download firmware components via interfaces such as UART, USB, or Ethernet.

Table 18.1 looks at a few example Linux bootloaders for ARM, PowerPC, and x86.

TABLE 18.1 Linux Bootloaders

Processor Platform	Linux Bootloaders
ARM	RedBoot (www.cygwin.com/redboot) is a bootloader popular on ARM-based hardware. Redboot is based on a hardware abstraction offered by the eCos operating system (http://ecos.sourceware.org/).
	The *BootLoader Object* or BLOB (http://sourceforge.net/projects/blob/), a bootloader originally developed for StrongARM-based boards, is commonly custom ported to other ARM-based platforms, too. BLOB is built as two images, one that performs minimal initializations, and the second that forms the bulk of the bootloader. The first image relocates the second to RAM, so the bootloader can easily upgrade itself.
PowerPC	PowerPC chips used on embedded devices include SoCs such as IBM's 405LP and the 440GP, and Motorola's MPC7xx and MPC8xx. Bootloaders such as U-Boot (http://sourceforge.net/projects/u-boot/), SLOF, and PIBS boot Linux on PowerPC-based hardware.
x86	Most x86-based systems boot from disk drives. Embedded x86 boards may boot from solid-state disks rather than mechanical drives. The first stage of a disk-resident bootloader consists of a sector-sized chunk that is loaded by the BIOS. This is called the *Master Boot Record* (MBR) and contains up to 446 bytes of code, four partition table entries consuming 16 bytes each, and a 2-byte signature (thus making up a 512-byte sector). The MBR is responsible for loading the second stage of the bootloader. Each intervening stage has its own tasks, but the final stage lets you choose the kernel image and command-line arguments, loads the kernel and any initial ramdisk to memory, and transfers control to the kernel. As an illustration, let's look at three bootloaders popularly used to boot Linux on x86-based hardware: • The *Linux Loader* or LILO (http://freshmeat.net/projects/lilo/) is packaged along with some Linux distributions. When the first stage of the bootloader is written to the boot sector, LILO precalculates the disk locations of the second stage and the kernel. If you build a new kernel image, you have to rewrite the boot sector. The second stage allows the user to interactively select the kernel image and configure command-line arguments. It then loads the kernel to memory.

Processor Platform	Linux Bootloaders
	• GRUB (www.gnu.org/software/grub) is different from LILO in that the kernel image can live in any supported filesystem, and the boot sector need not be rewritten if the kernel image changes. GRUB has an extra stage 1.5 that understands the filesystem holding the boot images. Currently supported filesystems are EXT2, DOS FAT, BSD FFS, IBM JFS, SGI XFS, Minix, and Reiserfs. GRUB complies with the Multiboot specification, which allows any complying operating system to boot via any complying bootloader. You looked at a sample GRUB configuration file in Chapter 2, "A Peek Inside the Kernel." • SYSLINUX (http://syslinux.zytor.com/) is a no-frills Linux bootloader. It understands the FAT filesystem, so you can store the kernel image and the second stage bootloader on a FAT partition.

Giving due thought to the design and architecture of the bootloader suite lays a solid foundation for embedded software development. The key is to choose the right bootloader as your starting point. The benefits range from a shorter software development cycle to a feature-rich and robust device.

Memory Layout

Figure 18.2 shows an example memory layout on an embedded device. The bootloader sits on top of the NOR flash. Following the bootloader lies the *param* block, a statically compiled binary image of kernel command-line arguments. The compressed kernel image comes next. The filesystem occupies the rest of the available flash memory. In the initial phase, when you start development with a first-shot kernel, the filesystem is usually a compressed ramdisk (*initrd* or *initramfs*), because having a flash-based filesystem entails getting the kernel MTD subsystem configured and running.

During power-on, the bootloader in Figure 18.2 uncompresses the kernel and loads it to DRAM at `0xc0200000`. It then loads the ramdisk at `0xc0280000` (unless you build an *initramfs* into the base kernel as you learned in Chapter 2). Finally, it obtains command-line arguments from the *param* block and transfers control to the kernel.

Because you may have to work with unconventional consoles and memory partitions on embedded devices, you have to pass the right command-line arguments to the kernel. For the device in Figure 18.2, this is a possible command line:

```
console=/dev/ttyS0,115200n8 root=/dev/ram initrd=0xC0280000
```

When you have the kernel MTD drivers recognizing your flash partitions, the area of flash that holds the ramdisk can instead contain a JFFS2-based filesystem. With this, you don't have to load the initrd to DRAM. Assuming that you have mapped the bootloader, param block, kernel, and filesystem to separate MTD partitions, the command line now looks like this:

```
console=/dev/ttyS0,115200n8 root=/dev/mtdblock3
```

See the sidebar "ATAGs" for another method of passing parameters from the bootloader to the kernel.

ATAGs

On ARM kernels, command-line arguments are deprecated in favor of a tagged list of parameters. This mechanism, called ATAG, is described in *Documentation/arm/Booting*. To pass a parameter to the kernel, create the corresponding tag in system memory from the bootloader, supply a kernel function to parse it, and add the latter to the list of tag parsing functions using the `__tagtable()` macro. The `tag` structure and its relatives are defined in *include/asm-arm/setup.h*, whereas *arch/arm/kernel/setup.c* contains functions that parse several predefined ATAGs.

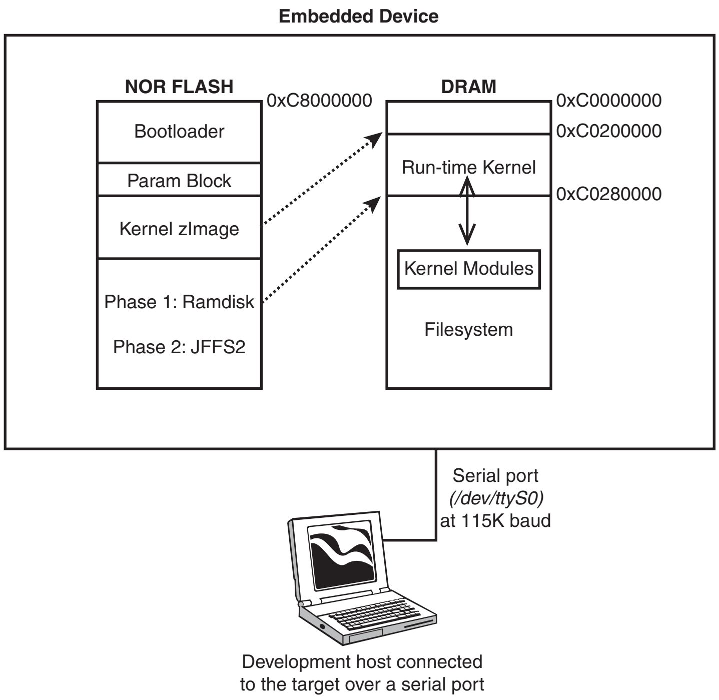


FIGURE 18.2 Example memory layout on an embedded device.

Kernel Porting

Like setting up tool chains, porting the kernel to your target device was a serious affair a few years ago. One had to evaluate the stability of the current kernel tree for the architecture of interest, apply available patches that were not yet part of the mainline, make modifications, and hope for good luck. But today, you are likely to find a close starting point, not just for your SoC, but for a hardware board that is similar to yours. For example, if you are designing an embedded device around the Freescale i.MX21 processor, you have the option of starting off with the kernel port (*arch/arm/mach-imx/*) for the i.MX21-based reference board built by the processor vendor. If you thus start development from a suitable distribution-supplied or standard kernel available for a board that resembles yours, chances are, you won't have to grapple with complex kernel bring-up issues.

But even with a close match, you are likely to face issues caused by modified memory maps, changed chip selects, board-specific GPIO assignments, dissimilar clock sources, disparate flash banks, timing requirements of a new LCD panel, or a different debug UART port. A change in clocking for example, can ripple through dozens of registers and impact the operation of several I/O peripherals. You might need an in-depth reading of the CPU reference manual to resolve it. To figure out a modified interrupt pin routing caused by a different GPIO assignment, you might have to pore over your board schematics. To program an LCD controller with `HSYNC` and `VSYNC` durations appropriate to your LCD panel, you may need to connect an oscilloscope to your board and digest the information that it gathers.

Depending on the demands on your device, you may also need to make kernel changes unrelated to bring up. It could be as simple as exporting some information via procfs or as complex as modifying the kernel for fast boot.

After you have the base kernel running, you can turn your attention to enabling device drivers for the different I/O interfaces on your hardware.

---

uClinux

uClinux is a branch of the Linux kernel intended for lower-end microprocessors that have no Memory Management Units (MMUs). uClinux ports are available for processors such as H8, Blackfin, and Dragonball. Most portions of uClinux are merged with the mainline 2.6 kernel.

The uClinux project is hosted at www.uclinux.org. The website contains patches, documentation, the code repository, list of supported architectures, and information for subscribing to the uclinux-dev mailing list.

Embedded Drivers

One of the reasons Linux is so popular in the embedded space is that its formidable application suite works regardless of the hardware platform, thanks to kernel abstraction layers that lie beneath them. So, as shown in Figure 18.3, all you need to do to get a feature-rich embedded system is to implement the low-level device drivers ensconced between the abstraction layers and the hardware. You need to do one of the following for each peripheral interface on your device:

- Qualify an existing driver. Test and verify that it works as it's supposed to.
- Find a driver that is a close match and modify it for your hardware.
- Write a driver from scratch.

Assuming a kernel engineer participates in component selection, you're likely to have existing drivers or close enough matches for most peripheral devices. To take advantage of existing drivers, go through the block diagram and schematics of your hardware, identify the different chipsets, and cobble together a working kernel configuration file that enables the right drivers. Based on your footprint or boot time requirements, modularize possible device drivers or build them into the base kernel.

To learn about device drivers for I/O interfaces commonly found on embedded hardware, let's take a clockwise tour around the embedded controller shown in Figure 18.1, starting with the NOR flash.

Flash Memory

Embedded devices such as the one in Figure 18.2, boot from flash memory and have filesystem data resident on flash-based storage. Many devices use a small NOR flash component for the former and a NAND flash part for the latter.[2] NOR memory, thus, holds the bootloader and the base kernel, whereas NAND storage contains filesystem partitions and device driver modules.

Flash drivers are supported by the kernel's MTD subsystem discussed in Chapter 17, "Memory Technology Devices." If you're using an MTD-supported chip, you need to write only an MTD map driver to suitably partition the flash to hold the bootloader, kernel, and filesystem. Listings 17.1, 17.2, and 17.3 in Chapter 17 implement a map driver for the Linux handheld, as shown in Figure 17.2 of the same chapter.

[2] In today's embedded market where the Bill Of Material (BOM) cost is often all-important, it's not uncommon for devices to contain only NAND storage. Such devices boot from NAND flash and have their filesystems also reside in NAND memory. NAND boot needs support from both the processor and the bootloader.

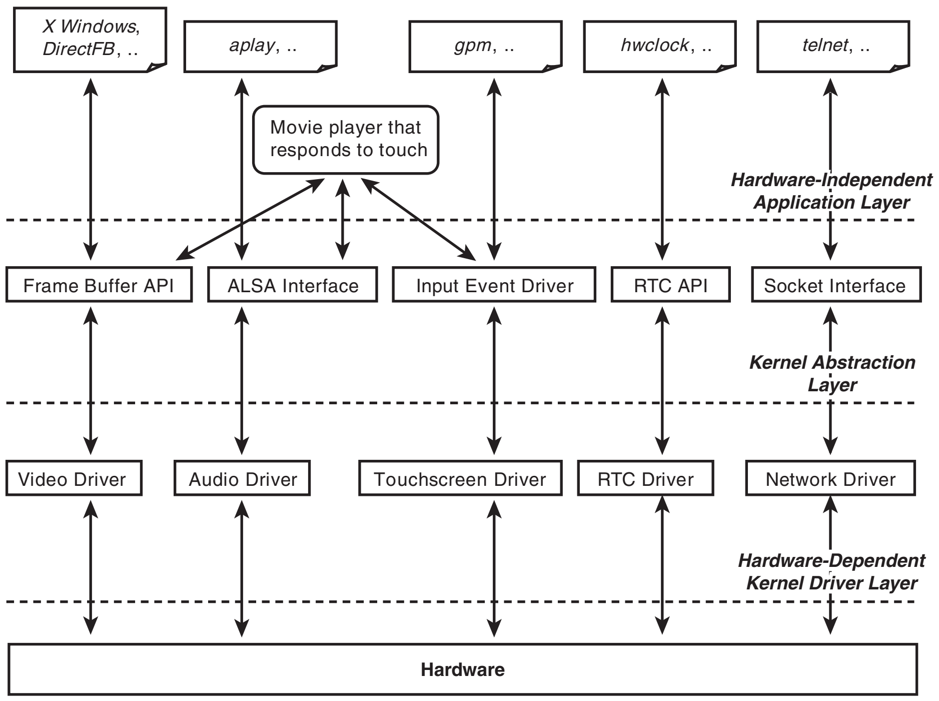


FIGURE 18.3 Hardware-independent applications and hardware-dependent drivers.

UART

The UART is responsible for serial communication and is an interface you are likely to find on all microcontrollers. UARTs are considered basic hardware, so the kernel contains UART drivers for all microcontrollers on which it runs. On embedded devices, UARTs are used to interface the processor with debug serial ports, modems, touch controllers, GPRS chipsets, Bluetooth chipsets, GPS devices, telemetry electronics, and so on.

Look at Chapter 6, "Serial Drivers," for a detailed discussion on the Linux serial subsystem.

Buttons and Wheels

Your device may have several miscellaneous peripherals such as keypads (micro keyboards organized in the common QWERTY layout, data-entry devices having overloaded keys as found in cell phones, keypads having ABC-type layout, and so on), LEDs, roller wheels, and buttons. These I/O devices interface with the CPU via GPIO lines or a CPLD (see the following "CPLD/FPGA" section). Drivers for such peripherals are usually straightforward char or misc drivers. Some of the drivers export device-access via procfs or sysfs rather than through */dev* nodes.

PCMCIA/CF

A PCMCIA or CF slot is a common add-on to embedded devices. The advantage of, say, WiFi enabling an embedded device using a CF card is that you won't have to respin the board if the WiFi controller goes end of life. Also, because diverse technologies are available in the PCMCIA/CF form factor, you have the freedom to change the connectivity mode from WiFi to another technology such as Bluetooth later. The disadvantage of such a scheme is that even with mechanical retaining, sockets are inherently unreliable. There is the possibility of the card coming loose due to shock and vibe, and resulting intermittent connections.

PCMCIA and CF device drivers are discussed in Chapter 9, "PCMCIA and Compact Flash."

SD/MMC

Many embedded processors include controllers that communicate with SD/MMC media. SD/MMC storage is built using NAND flash memory. Like CF cards, SD/MMC cards add several gigabytes of memory to your device. They also offer an easy memory upgrade path, because the available density of SD/MMC cards is constantly increasing.

Chapter 14, "Block Drivers," points you to the SD/MMC subsystem in the kernel.

USB

Legacy computers support the USB host mode, by which you can communicate with most classes of USB devices. Embedded systems frequently also require support for the USB device mode, wherein the system itself functions as a USB device and plugs into other host computers.

As you saw in Chapter 11, "Universal Serial Bus," many embedded controllers support USB OTG that lets your device work either as a USB host or as a USB device. It allows you, for example, to connect a USB pen drive to your embedded device. It also allows your embedded device to function as a USB pen drive by exporting part of its local storage for external access. The Linux USB subsystem offers drivers for USB OTG. For hardware that is not compatible with OTG, the USB Gadget project, now part of the mainline kernel, brings USB device capability.

RTC

Many embedded SoCs include RTC support to keep track of wall time, but some rely on an external RTC chip. Unlike x86-based computers where the RTC is part of the South Bridge chipset, embedded controllers commonly interface with external RTCs via slow serial buses such as I^2C or SPI. You can drive such RTCs by writing client drivers that use the services of the I^2C or SPI core as discussed in Chapter 8, "The Inter-Integrated Circuit Protocol." Chapter 2 and Chapter 5, "Character Drivers," discussed RTC support on x86-based systems.

Audio

As you saw in Chapter 13, "Audio Drivers," an audio codec converts digital audio data to analog sound signals for playback via speakers and performs the reverse operation for recording through a microphone. The codec's connection with the CPU depends on the digital audio interface supported by the embedded controller. The usual way to communicate with a codec is via buses, such as AC'97 or I^2S.

Touch Screen

Touch is the primary input mechanism on several embedded devices. Many PDAs offer soft keyboards for data entry. In Chapter 6, we developed a driver for a serial touch controller, and in Chapter 7, "Input Drivers," we looked at a touch controller that interfaced with the CPU via the SPI bus.

If your driver conforms to the *input* API, it should be straightforward to tie it with a graphical user interface. You might, however, need to add custom support to calibrate and linearize the touch panel.

Video

Some embedded systems are headless, but many have associated displays. A suitably oriented (landscape or portrait) LCD panel is connected to the video controller that is part of the embedded SoC. Many LCD panels come with integrated touch screens.

As you learned in Chapter 12, "Video Drivers," frame buffers insulate applications from display hardware, so porting a compliant GUI to your device is easy, as long as your display driver conforms to the frame buffer interface.

CPLD/FPGA

Complex Programmable Logic Devices (CPLDs) or their heavy-duty counterparts, *Field Programmable Gate Arrays* (FPGAs), can add a thick layer of fast OS-independent logic. You can program CPLDs (and FPGAs) in a language such as *Very high speed integrated circuit Hardware Description Language* (VHDL). Electrical signals between the processor and peripherals propagate through the CPLD, so by appropriately programming the CPLD, the OS obtains elegant register interfaces for performing complex I/O. The VHDL code in the CPLD internally latches these register contents onto the data bus after performing necessary control logic.

Consider, for example, an external serial LCD controller that has to be driven by shifting in each pixel bit. The Linux driver for this device will have a tough time toggling the clock and wiggling I/O pins several times for sending each pixel or command byte to the serial LCD controller. If this LCD controller is routed to the processor via a CPLD, however, the VHDL code can perform the necessary serial shifting by clocking each bit in and present a parallel register interface to the OS for command and data. With these virtual LCD command and data registers, the LCD driver implementation is rendered simple. Essentially, the CPLD converts the cumbersome serial LCD controller to a convenient, parallel one.

If the CPLD engineer and the Linux driver developer collaborate, they can arrive at an optimum partitioning between the VHDL code and the Linux driver that'll save time and cost.

Connectivity

Connectivity injects intelligence, so there are few embedded devices that have no communication capability. Popular networking technologies found on embedded devices include WiFi, Bluetooth, cellular modems, Ethernet, and radio communication.

Chapter 15, "Network Interface Cards," explored device drivers for wired networking, and Chapter 16, "Linux Without Wires," looked at drivers for wireless communication technologies.

Domain-Specific Electronics

Your device is likely to contain electronics specific to the usage industry domain. It could be a telemetry interface for a hospital-grade device, a sensor for automotive hardware, biometrics for a security gadget, GPRS for a cellular phone, or GPS for a navigation system. These peripherals usually communicate with the embedded controller over standard I/O interfaces such as UART, USB, I^2C, SPI, or *controller area network* (CAN). For devices interfacing via a UART, you often have little work to do at the device driver level because the UART driver takes care of the communication. For devices such as a fingerprint sensor that interface via USB, you may have to write a USB client driver. You might also face proprietary interfaces, such as a switching fabric for a network processor, in which case, you may need to write a full-fledged device driver.

Consider the digital media space. Cable or *Direct-to-home* (DTH) interface systems are usually built around *set-top box* (STB) chipsets. These chips have capabilities such as personal video recording (recording multiple channels to a hard disk, recording a channel while viewing another and so forth) and conditional access (allowing the service provider to control what the end user sees depending on subscription). To achieve this, STB chips have a processor core coupled with a powerful graphics engine. The latter implements MPEG codecs in hardware. Such audio-video codecs can decode compressed digital media standards such as MPEG2 and MPEG4. (MPEG is an acronym for *Moving Picture Experts Group,* the body responsible for developing motion picture standards.) If you are embedding Linux onto an STB, you will need to drive such audio-video codecs.

More Drivers

If your device serves a life-critical industry domain such as health care, the system memory might have ECC capabilities. Chapter 20, "More Devices and Drivers," discusses ECC reporting.

If your embedded device is battery powered, you may want to use a suitable CPU frequency governor to dynamically scale processor frequency and save power. Chapter 20 also discusses CPU frequency drivers and power management.

Most embedded processors have a built-in hardware watchdog that recovers the system from freezes. You looked at watchdog drivers in Chapter 5. Use a suitable driver from *drivers/char/watchdog/* as the starting point to implement a driver for your system's watchdog.

If your embedded device contains circuitry to detect *brownout,*[3] you might need to add capability to the kernel to sense that condition and take appropriate action.

Several embedded SoCs contain built-in *pulse-width modulator* (PWM) units. PWMs let you digitally control analog devices such as buzzers. The voltage level

[3] *Brownout* is the scenario when input voltage drops below tolerable levels. (*Blackout*, on the other hand, refers to total loss of power.) Brownout detection is especially relevant if your device is powered by a technology such as Power over Ethernet (PoE) rather than a conventional wall socket.

supplied to the target device is varied by programming the PWM's duty cycle (the On time of the PWM's output waveform relative to its period). LCD brightness is another example of a feature controllable using PWMs. Depending on the target device and the usage scenario, you can implement char or misc driver interfaces to PWMs.

The Root Filesystem

Before the advent of Linux distributions, it used to be a project by itself to put together a compact application-set tailored to suit the size limitations of available storage. One had to cobble together the sources of a minimal set of utilities, libraries, tools, and daemons; ensure that their versions liked each other; and cross-compile them. Today's distributions supply a ready-made application-set built for supported processors and offer tools that let you pick and choose components at the granularity of packages. Of course, you may still want to implement custom utilities and tools to supplement the distribution-supplied applications.

On embedded devices, flash memory (discussed in Chapter 17) is the commonly used vehicle to hold the application-set and is mounted as the root device at the end of the boot process. Hard disks are uncommon because they are power-intensive, bulky, and have moving parts that are not tolerant to shock and vibe. Common places that hold the root filesystem on embedded devices include the following:

- An initial ramdisk (*initramfs* or *initrd*) is usually the starting point before you get drivers for other potential root devices working and is used for development purposes.
- NFS-mounting the root filesystem is a development strategy much more powerful than using a ramdisk. We discuss this in detail in the next section.
- Storage media such as flash chips, SD/MMC cards, CF cards, DOCs, and DOMs.

Note that it may not be a good idea to let all the data stay in the root partition. It's common to spread files across different storage partitions and tag desired *read-write* or *read-only* protection flags, especially if there is the possibility that the device will be shut down abruptly.

NFS-Mounted Root

NFS-mounting the root filesystem can serve as a catalyst to hasten the embedded development cycle. In this case, the root filesystem physically resides on your development host and not on the target, so its size is virtually unlimited and not restricted by the amount of storage locally available on the target. Downloading device driver modules or applications to the target, as well as uploading logs, is as simple (and fast) as copying them to */path/to/target/rootfilesystem/* on your development host. Such ease of testing and debugging is a good reason why you should insist on having Ethernet on engineering-level hardware, even if production units won't have Ethernet support. Having Ethernet on your board also lets your bootloader use the Trivial File Transfer Protocol (TFTP) to download the kernel image to the target over a network.

Table 18.2[4] shows the typical steps needed to get TFTP and NFS working with your embedded device. It assumes that your development host also doubles up as TFTP, NFS, and DHCP servers, and that the bootloader (BLOB in this example) supports the Ethernet chipset used on the embedded device.

TABLE 18.2 Saving Development Time with TFTP and NFS

	Target Embedded Device	Host Development Platform

Kernel Boot over TFTP	Configure the IP address of the target and the server (host) from the bootloader prompt:
	`/* Target IP */`
	`blob> ip 4.1.1.2`
	`/* Host IP */`
	`blob> server 4.1.1.1`
	`/* Kernel image */`
	`blob> TftpFile /tftpdir/zImage`
	`/* Pull the Kernel over the net */`
	`blob> tftp`
	`TFTPing /tftpboot/zImage............Ok`
	`blob>`
		Configure the host IP address:
		`bash> ifconfig eth0 4.1.1.1`
		Install and configure the TFTP server (the exact steps depend on your distribution):
		`bash> cat /etc/xinetd.conf/tftp`
		`service tftp`
		`{`
		`  socket_type   = dgram`
		`  protocol      = udp`
		`  wait          = yes`
		`  user          = root`
		`  server        = /usr/sbin/in.tftpd`
		`  server_args   = /tftpdir`
		`  disable       = no`
		`  per_source    = 11`
		`  cps           = 100 2`
		`  flags         = IPv4`
		`}`
		Make sure that the TFTP server is present in */usr/sbin/in.tftpd* and that *xinetd* is alive.
		Compile the target kernel with NFS enabled and copy it to */tftpdir/zImage.*

Continues

[4] The filenames and directory path names used in Table 18.2 are distribution-dependent.

TABLE 18.2 Continued

	Target Embedded Device	Host Development Platform
Root file-system over NFS	**`blob> boot console=/dev/ttyS0,115200n8 root=/dev/nfs ip=dhcp`**	Export */path/to/target/root/* for NFS access:
	`/*Kernel boot messages*/`	**`bash> cat /etc/exports`**
	`/* ... */`	`/path/to/target/root/ *(rw,sync,no_root_squash,no_all_squash)`
	`VFS: Mounted root (nfs filesystem)`	Start NFS:
	`/* ... */`	**`bash> service nfs start`**
	`login:`	Configure the DHCP server. The kernel on the embedded device relies on this server to assign it the 4.1.1.2 IP address during boot and to supply */path/to/target/root/:*
		`bash> cat /etc/dhcpd.conf`
		`...`
		`subnet 4.1.1.0 netmask 255.255.255.0 {`
		`range 4.1.1.2 4.1.1.10`
		`max-lease-time 43200`
		`option routers 4.1.1.1`
		`option ip-forwarding off`
		`option broadcast-address 4.1.1.255`
		`option subnet-mask 255.255.255.0`
		`group {`
		`  next-server 4.1.1.1`
		`  host target-device {`
		`   /* MAC of the embedded device */`
		`   hardware Ethernet AA:BB:CC:DD:EE:FF;`
		`   fixed-address 4.1.1.2;`
		`   option root-path "/path/to/target/root/";`
		`  }`
		`}`
		`...`
		`bash> service dhcpd start`
		`bash>`

Compact Middleware

Embedded devices that are tight on memory prefer middleware implementations that have small footprint and low runtime memory requirements. The trade-offs usually lie in features, standards compatibility, and speed. Let's take a look at some popular compact middleware solutions that may be potential candidates for populating your root filesystem.

BusyBox is a tool commonly used to provide a multi-utility environment on embedded systems having limited memory. It scratches out some features but provides an optimized replacement for several shell utilities.

uClibc is a compact version of the GNU C library that was originally developed to work with uClinux. uClibc works on normal Linux systems, too, and is licensed under LGPL. If your embedded device is short on space, try uClibc rather than glibc.

Embedded systems that need to run an X Windows server commonly rely on *TinyX*, a low-footprint X server shipped along with the XFree86 4.0 code. TinyX runs over frame buffer drivers and can be used on devices, such as the one showed in Figure 12.6 of Chapter 12.

Thttpd is a lightweight HTTP server that makes low demands on CPU and memory resources.

Even if you are creating a non-Linux solution using a tiny 8-bit MMU-less microcontroller, you will likely want it to interoperate with Linux. Assume, for example, that you are writing deeply embedded firmware for an Infrared storage keychain. The keychain can hold a gigabyte of personal data that can be accessed via a web browser from your Linux laptop over Infrared. If you are running a compact TCP/IP stack, such as *uIP* over a minimal IrDA stack such as *Pico-IrDA* on the Infrared keychain, you have the task of ensuring their interoperability with the corresponding Linux protocol stacks.

Table 18.3 lists the home pages of the compact middleware projects referred to in this section.

TABLE 18.3 Examples of Compact Middleware

Name	Description	Download Location
BusyBox	Small footprint shell environment	www.busybox.net
uClibc	Small-sized version of glibc	www.uclibc.org
TinyX	X server for devices that are tight on memory	Part of the X Windows source tree downloadable from ftp://ftp.xfree86.org/pub/XFree86/4.0/
Thttpd	Tiny HTTP server	www.acme.com/software/thttpd
uIP	Compact TCP/IP stack for microcontrollers	www.sics.se/~adam/uip
Pico-IrDA	Minimal IrDA stack for microcontrollers	http://blaulogic.com/pico_irda.shtml

Test Infrastructure

Most industry domains that use embedded devices are governed by regulatory bodies. Having an extensible and robust test infrastructure is likely to be as important as implementing modifications to the kernel and device drivers. Broadly, the test framework is responsible for the following:

1. Demonstrating compliance to obtain regulatory approvals. If your system is a medical-grade device for the U.S. market, for example, you should orient your test suite for getting approvals from the *Food and Drug Administration* (FDA).
2. Most electronic devices intended for the U.S. market have to comply with emission standards such as *electromagnetic interference* (EMI) and *electromagnetic compatibility* (EMC) as laid down by the *Federal Communications Commission* (FCC). To demonstrate compliance, you may need to run a battery of tests inside a chamber that models different operating environments. You might also have to verify that the system runs normally when an electrostatic gun is pointed at different parts of the board.
3. Build verification tests. Whenever you build a software deliverable, subject it to *quality assurance* (QA) using these tests.
4. Manufacturing tests. Each time a device is assembled, you have to verify its functionality using a set of tests. These tests assume significance when manufacturing moves into volume production.

To have a common test base for all these, it's a good idea to implement your test harness over Linux, rather than develop it as a stand-alone suite. Stand-alone code is not easily scalable or extendable. Adding a simple test to ping the next-hop router is a five-line script on a Linux-based test system but can entail writing a network driver and a protocol stack if you are using a stand-alone test monitor.

A test engineer need not be a kernel guru but will need to imbibe implementation information from the development team and think critically.

Debugging

Before closing this chapter, let's visit a few topics related to debugging embedded software.

Board Rework

Navigating board schematics and datasheets is an important debugging skill you need while bringing up the bootloader or kernel on embedded hardware. Understanding your board's *placement plot*, which is a file that shows the position of chips on your board, is a big help when you are debugging a potential hardware problem using an oscilloscope, or when you need to perform minor board rework. *Reference designators* (such as U10 and U11 in Figure 18.4) associate each chip in the schematic with the placement plot. *Printed circuit boards* (PCBs) are usually clothed with *silk screens* that print the reference designator near each chip.

Consider this fictitious scenario where USB enumeration doesn't occur on your board under test. The USB hub driver detects device insertions but is not able to assign endpoint addresses. A close look at the schematics reveals that the connections originating from the `SPEED` and `MODE` pins of the USB transceiver have been interchanged by mistake. An examination of the placement plot identifies the location of the transceiver on the PCB. Matching the transceiver's reference designator on the placement plot with the silk screen on the PCB pinpoints the places where you have to solder "yellow wires" to repair the faulty connections.

A multimeter and an oscilloscope are worthy additions to your embedded debugging toolkit. As an illustration, let's consider an example situation involving the I^2C RTC, as shown in Figure 8.3 of Chapter 8. That figure is reproduced there with a multimeter/scope attached to probe points of interest. Consider this scenario: You have written an I^2C client driver for this RTC chip as described in the section "Device Example: Real Time Clock" in Chapter 8. However, when you run your driver on the board, it renders the system unbootable. Neither does the bootloader come up when you reset the board, nor does your JTAG debugger connect to the target. To understand possible causes of this seemingly fatal error, let's take a closer look at the connection diagram. Because both the RTC and the CPU need an external clock, the board supplies it using a single 32KHz crystal. This 32KHz clock needs to be buffered, however. The RTC buffers the clock for its use and makes it available on an output pin for free. This pin, `CLK_OUT`, feeds the clock to the processor. Connect an oscilloscope (or a multimeter that can measure frequency) between `CLK_OUT` and ground to verify the processor clock frequency. As you can see in Figure 18.4, the scope reads 1KHz rather than the expected 32KHz! What could be wrong here?

The RTC control register contains bits that choose the frequency of `CLK_OUT`. While probing the chip (on the lines of `myrtc_attach()` in Chapter 8), the driver has erroneously initialized these bits to generate 1KHz on `CLK_OUT`. RTC registers are nonvolatile because of the battery backup, so the control register holds this bad value across reboots. The resulting skewed clock is sufficient to render the system unbootable. Disconnect the RTC's backup battery, drain the registers, reconnect the battery, verify using the scope that the 32KHz clock is restored on `CLK_OUT`, fix your driver code, and start afresh!

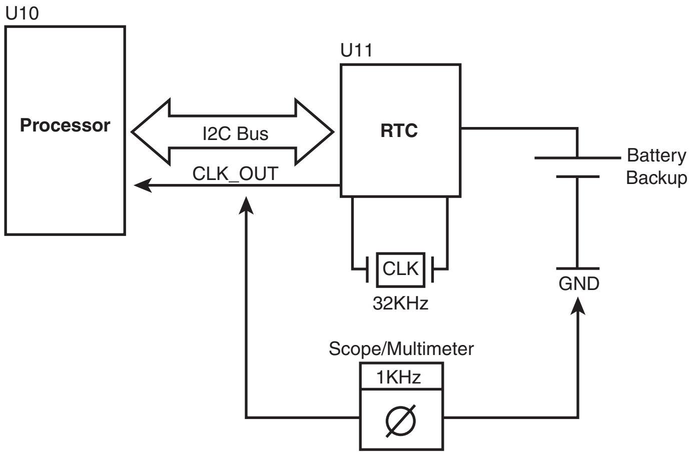


FIGURE 18.4 Debugging an I^{2}C RTC on an embedded system.

Debuggers

You can use most of the debugging techniques that you will learn in Chapter 21 while embedding Linux. Kernel debuggers are available for several processor platforms. JTAG debuggers, also explored in Chapter 21, are more powerful than kernel debuggers and are popularly used in the embedded space to debug the bootloader, base kernel, and device-driver modules.

Index