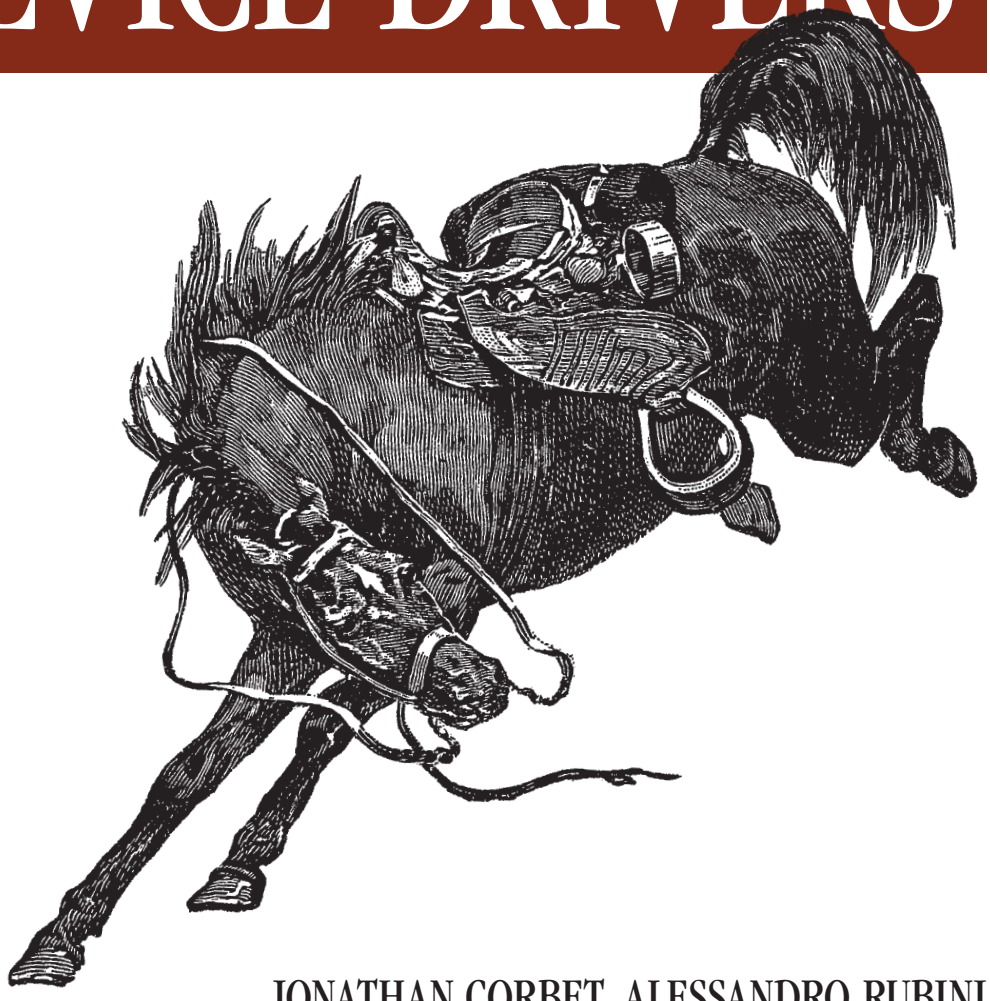
WHERE THE KERNEL MEETS THE HARDWARE

3rd Edition

LINUX DEVICE DRIVERS



JONATHAN CORBET, ALESSANDRO RUBINI
& GREG KROAH-HARTMAN

O'REILLY®

CHAPTER 1
An Introduction to Device Drivers

One of the many advantages of free operating systems, as typified by Linux, is that their internals are open for all to view. The operating system, once a dark and mysterious area whose code was restricted to a small number of programmers, can now be readily examined, understood, and modified by anybody with the requisite skills. Linux has helped to democratize operating systems. The Linux kernel remains a large and complex body of code, however, and would-be kernel hackers need an entry point where they can approach the code without being overwhelmed by complexity. Often, device drivers provide that gateway.

Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

There are a number of reasons to be interested in the writing of Linux device drivers. The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future. Individuals may need to know about drivers in order to gain access to a particular device that is of interest to them. Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets. And the open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users.

This book teaches you how to write your own drivers and how to hack around in related parts of the kernel. We have taken a device-independent approach; the programming techniques and interfaces are presented, whenever possible, without being tied to any specific device. Each driver is different; as a driver writer, you need to

understand your specific device well. But most of the principles and basic techniques are the same for all drivers. This book cannot teach you about your device, but it gives you a handle on the background you need to make your device work.

As you learn to write drivers, you find out a lot about the Linux kernel in general; this may help you understand how your machine works and why things aren't always as fast as you expect or don't do quite what you want. We introduce new ideas gradually, starting off with very simple drivers and building on them; every new concept is accompanied by sample code that doesn't need special hardware to be tested.

This chapter doesn't actually get into writing code. However, we introduce some background concepts about the Linux kernel that you'll be glad you know later, when we do launch into programming.

The Role of the Device Driver

As a programmer, you are able to make your own choices about your driver, and choose an acceptable trade-off between the programming time required and the flexibility of the result. Though it may appear strange to say that a driver is "flexible," we like this word because it emphasizes that the role of a device driver is providing *mechanism*, not *policy*.

The distinction between mechanism and policy is one of the best ideas behind the Unix design. Most programming problems can indeed be split into two parts: "what capabilities are to be provided" (the mechanism) and "how those capabilities can be used" (the policy). If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs.

For example, Unix management of the graphic display is split between the X server, which knows the hardware and offers a unified interface to user programs, and the window and session managers, which implement a particular policy without knowing anything about the hardware. People can use the same window manager on different hardware, and different users can run different configurations on the same workstation. Even completely different desktop environments, such as KDE and GNOME, can coexist on the same system. Another example is the layered structure of TCP/IP networking: the operating system offers the socket abstraction, which implements no policy regarding the data to be transferred, while different servers are in charge of the services (and their associated policies). Moreover, a server like *ftpd* provides the file transfer mechanism, while users can use whatever client they prefer; both command-line and graphic clients exist, and anyone can write a new user interface to transfer files.

Where drivers are concerned, the same separation of mechanism and policy applies. The floppy driver is policy free—its role is only to show the diskette as a continuous

array of data blocks. Higher levels of the system provide policies, such as who may access the floppy drive, whether the drive is accessed directly or via a filesystem, and whether users may mount filesystems on the drive. Since different environments usually need to use hardware in different ways, it's important to be as policy free as possible.

When writing drivers, a programmer should pay particular attention to this fundamental concept: write kernel code to access the hardware, but don't force particular policies on the user, since different users have different needs. The driver should deal with making the hardware available, leaving all the issues about *how* to use the hardware to the applications. A driver, then, is flexible if it offers access to the hardware capabilities without adding constraints. Sometimes, however, some policy decisions must be made. For example, a digital I/O driver may only offer byte-wide access to the hardware in order to avoid the extra code needed to handle individual bits.

You can also look at your driver from a different perspective: it is a software layer that lies between the applications and the actual device. This privileged role of the driver allows the driver programmer to choose exactly how the device should appear: different drivers can offer different capabilities, even for the same device. The actual driver design should be a balance between many different considerations. For instance, a single device may be used concurrently by different programs, and the driver programmer has complete freedom to determine how to handle concurrency. You could implement memory mapping on the device independently of its hardware capabilities, or you could provide a user library to help application programmers implement new policies on top of the available primitives, and so forth. One major consideration is the trade-off between the desire to present the user with as many options as possible and the time you have to write the driver, as well as the need to keep things simple so that errors don't creep in.

Policy-free drivers have a number of typical characteristics. These include support for both synchronous and asynchronous operation, the ability to be opened multiple times, the ability to exploit the full capabilities of the hardware, and the lack of software layers to "simplify things" or provide policy-related operations. Drivers of this sort not only work better for their end users, but also turn out to be easier to write and maintain as well. Being policy-free is actually a common target for software designers.

Many device drivers, indeed, are released together with user programs to help with configuration and access to the target device. Those programs can range from simple utilities to complete graphical applications. Examples include the *tunelp* program, which adjusts how the parallel port printer driver operates, and the graphical *cardctl* utility that is part of the PCMCIA driver package. Often a client library is provided as well, which provides capabilities that do not need to be implemented as part of the driver itself.

The scope of this book is the kernel, so we try not to deal with policy issues or with application programs or support libraries. Sometimes we talk about different policies and how to support them, but we won't go into much detail about programs using the device or the policies they enforce. You should understand, however, that user programs are an integral part of a software package and that even policy-free packages are distributed with configuration files that apply a default behavior to the underlying mechanisms.

Splitting the Kernel

In a Unix system, several concurrent *processes* attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource. The *kernel* is the big chunk of executable code in charge of handling all such requests. Although the distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split (as shown in Figure 1-1) into the following parts:

Process management
: The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.

Memory management
: The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple *malloc*/*free* pair to much more complex functionalities.

Filesystems
: Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the Linux-standard ext3 filesystem, the commonly used FAT filesystem or several others.

Device control

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a *device driver*. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive. This aspect of the kernel's functions is our primary interest in this book.

Networking

Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

Loadable Modules

One of the good features of Linux is the ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel (and remove functionality as well) while the system is up and running.

Each piece of code that can be added to the kernel at runtime is called a *module*. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the *insmod* program and can be unlinked by the *rmmod* program.

Figure 1-1 identifies different classes of modules in charge of specific tasks—a module is said to belong to a specific class according to the functionality it offers. The placement of modules in Figure 1-1 covers the most important classes, but is far from complete because more and more functionality in Linux is being modularized.

Classes of Devices and Modules

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a *char module*, a *block module*, or a *network module*. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability.

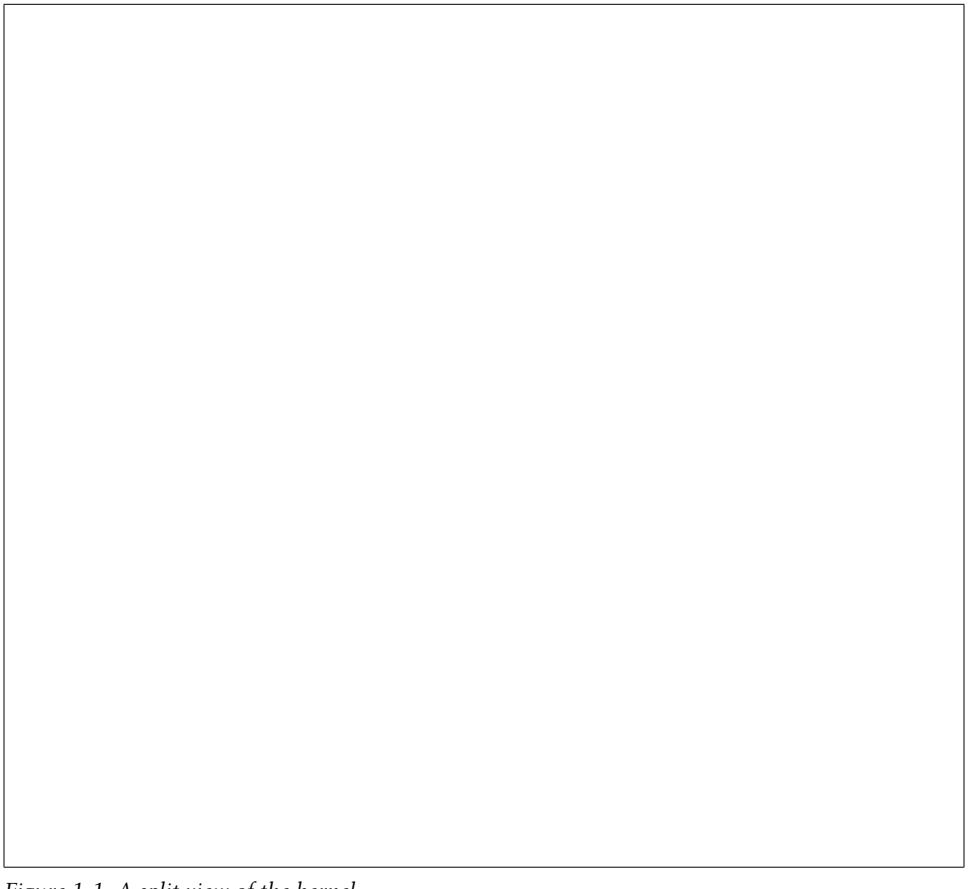


Figure 1-1. A split view of the kernel

The three classes are:

Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the *open*, *close*, *read*, and *write* system calls. The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as */dev/tty1* and */dev/lp0*. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using *mmap* or *lseek*.

Block devices

Like char devices, block devices are accessed by filesystem nodes in the */dev* directory. A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an *interface* is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.

Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as */dev/tty1* is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as `eth0`), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of *read* and *write*, the kernel calls functions related to packet transmission.

There are other ways of classifying driver modules that are orthogonal to the above device types. In general, some types of drivers work with additional layers of kernel support functions for a given type of device. For example, one can talk of universal serial bus (USB) modules, serial modules, SCSI modules, and so on. Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as a char device (a USB serial port, say), a block device (a USB memory card reader), or a network device (a USB Ethernet interface).

Other classes of device drivers have been added to the kernel in recent times, including FireWire drivers and I2O drivers. In the same way that they handled USB and SCSI drivers, kernel developers collected class-wide features and exported them to driver implementers to avoid duplicating work and bugs, thus simplifying and strengthening the process of writing such drivers.

In addition to device drivers, other functionalities, both hardware and software, are modularized in the kernel. One common example is filesystems. A filesystem type determines how information is organized on a block device in order to represent a tree of directories and files. Such an entity is not a device driver, in that there's no explicit device associated with the way the information is laid down; the filesystem type is instead a software driver, because it maps the low-level data structures to high-level data structures. It is the filesystem that determines how long a filename can be and what information about each file is stored in a directory entry. The filesystem module must implement the lowest level of the system calls that access directories and files, by mapping filenames and paths (as well as other information, such as access modes) to data structures stored in data blocks. Such an interface is completely independent of the actual data transfer to and from the disk (or other medium), which is accomplished by a block device driver.

If you think of how strongly a Unix system depends on the underlying filesystem, you'll realize that such a software concept is vital to system operation. The ability to decode filesystem information stays at the lowest level of the kernel hierarchy and is of utmost importance; even if you write a block driver for your new CD-ROM, it is useless if you are not able to run *ls* or *cp* on the data it hosts. Linux supports the concept of a filesystem module, whose software interface declares the different operations that can be performed on a filesystem inode, directory, file, and superblock. It's quite unusual for a programmer to actually need to write a filesystem module, because the official kernel already includes code for the most important filesystem types.

Security Issues

Security is an increasingly important concern in modern times. We will discuss security-related issues as they come up throughout the book. There are a few general concepts, however, that are worth mentioning now.

Any security check in the system is enforced by kernel code. If the kernel has security holes, then the system as a whole has holes. In the official kernel distribution, only an authorized user can load modules; the system call *init_module* checks if the invoking process is authorized to load a module into the kernel. Thus, when running an official kernel, only the superuser,* or an intruder who has succeeded in becoming privileged, can exploit the power of privileged code.

When possible, driver writers should avoid encoding security policy in their code. Security is a policy issue that is often best handled at higher levels within the kernel, under the control of the system administrator. There are always exceptions, however.

* Technically, only somebody with the `CAP_SYS_MODULE` capability can perform this operation. We discuss capabilities in Chapter 6.

As a device driver writer, you should be aware of situations in which some types of device access could adversely affect the system as a whole and should provide adequate controls. For example, device operations that affect global resources (such as setting an interrupt line), which could damage the hardware (loading firmware, for example), or that could affect other users (such as setting a default block size on a tape drive), are usually only available to sufficiently privileged users, and this check must be made in the driver itself.

Driver writers must also be careful, of course, to avoid introducing security bugs. The C programming language makes it easy to make several types of errors. Many current security problems are created, for example, by *buffer overrun* errors, in which the programmer forgets to check how much data is written to a buffer, and data ends up written beyond the end of the buffer, thus overwriting unrelated data. Such errors can compromise the entire system and must be avoided. Fortunately, avoiding these errors is usually relatively easy in the device driver context, in which the interface to the user is narrowly defined and highly controlled.

Some other general security ideas are worth keeping in mind. Any input received from user processes should be treated with great suspicion; never trust it unless you can verify it. Be careful with uninitialized memory; any memory obtained from the kernel should be zeroed or otherwise initialized before being made available to a user process or device. Otherwise, information leakage (disclosure of data, passwords, etc.) could result. If your device interprets data sent to it, be sure the user cannot send anything that could compromise the system. Finally, think about the possible effect of device operations; if there are specific operations (e.g., reloading the firmware on an adapter board or formatting a disk) that could affect the system, those operations should almost certainly be restricted to privileged users.

Be careful, also, when receiving software from third parties, especially when the kernel is concerned: because everybody has access to the source code, everybody can break and recompile things. Although you can usually trust precompiled kernels found in your distribution, you should avoid running kernels compiled by an untrusted friend—if you wouldn't run a precompiled binary as root, then you'd better not run a precompiled kernel. For example, a maliciously modified kernel could allow anyone to load a module, thus opening an unexpected back door via *init_module*.

Note that the Linux kernel can be compiled to have no module support whatsoever, thus closing any module-related security holes. In this case, of course, all needed drivers must be built directly into the kernel itself. It is also possible, with 2.2 and later kernels, to disable the loading of kernel modules after system boot via the capability mechanism.

Version Numbering

Before digging into programming, we should comment on the version numbering scheme used in Linux and which versions are covered by this book.

First of all, note that *every* software package used in a Linux system has its own release number, and there are often interdependencies across them: you need a particular version of one package to run a particular version of another package. The creators of Linux distributions usually handle the messy problem of matching packages, and the user who installs from a prepackaged distribution doesn't need to deal with version numbers. Those who replace and upgrade system software, on the other hand, are on their own in this regard. Fortunately, almost all modern distributions support the upgrade of single packages by checking interpackage dependencies; the distribution's package manager generally does not allow an upgrade until the dependencies are satisfied.

To run the examples we introduce during the discussion, you won't need particular versions of any tool beyond what the 2.6 kernel requires; any recent Linux distribution can be used to run our examples. We won't detail specific requirements, because the file *Documentation/Changes* in your kernel sources is the best source of such information if you experience any problems.

As far as the kernel is concerned, the even-numbered kernel versions (i.e., 2.6.*x*) are the stable ones that are intended for general distribution. The odd versions (such as 2.7.*x*), on the contrary, are development snapshots and are quite ephemeral; the latest of them represents the current status of development, but becomes obsolete in a few days or so.

This book covers Version 2.6 of the kernel. Our focus has been to show all the features available to device driver writers in 2.6.10, the current version at the time we are writing. This edition of the book does not cover prior versions of the kernel. For those of you who are interested, the second edition covered Versions 2.0 through 2.4 in detail. That edition is still available online at *http://lwn.net/Kernel/LDD2/*.

Kernel programmers should be aware that the development process changed with 2.6. The 2.6 series is now accepting changes that previously would have been considered too large for a "stable" kernel. Among other things, that means that internal kernel programming interfaces can change, thus potentially obsoleting parts of this book; for this reason, the sample code accompanying the text is known to work with 2.6.10, but some modules don't compile under earlier versions. Programmers wanting to keep up with kernel programming changes are encouraged to join the mailing lists and to make use of the web sites listed in the bibliography. There is also a web page maintained at *http://lwn.net/Articles/2.6-kernel-api/*, which contains information about API changes that have happened since this book was published.

This text doesn't talk specifically about odd-numbered kernel versions. General users never have a reason to run development kernels. Developers experimenting with new features, however, want to be running the latest development release. They usually keep upgrading to the most recent version to pick up bug fixes and new implementations of features. Note, however, that there's no guarantee on experimental kernels,* and nobody helps you if you have problems due to a bug in a noncurrent odd-numbered kernel. Those who run odd-numbered versions of the kernel are usually skilled enough to dig in the code without the need for a textbook, which is another reason why we don't talk about development kernels here.

Another feature of Linux is that it is a platform-independent operating system, not just "a Unix clone for PC clones" anymore: it currently supports some 20 architectures. This book is platform independent as far as possible, and all the code samples have been tested on at least the x86 and x86-64 platforms. Because the code has been tested on both 32-bit and 64-bit processors, it should compile and run on all other platforms. As you might expect, the code samples that rely on particular hardware don't work on all the supported platforms, but this is always stated in the source code.

License Terms

Linux is licensed under Version 2 of the GNU General Public License (GPL), a document devised for the GNU project by the Free Software Foundation. The GPL allows anybody to redistribute, and even sell, a product covered by the GPL, as long as the recipient has access to the source and is able to exercise the same rights. Additionally, any software product derived from a product covered by the GPL must, if it is redistributed at all, be released under the GPL.

The main goal of such a license is to allow the growth of knowledge by permitting everybody to modify programs at will; at the same time, people selling software to the public can still do their job. Despite this simple objective, there's a never-ending discussion about the GPL and its use. If you want to read the license, you can find it in several places in your system, including the top directory of your kernel source tree in the *COPYING* file.

Vendors often ask whether they can distribute kernel modules in binary form only. The answer to that question has been deliberately left ambiguous. Distribution of binary modules—as long as they adhere to the published kernel interface—has been tolerated so far. But the copyrights on the kernel are held by many developers, and not all of them agree that kernel modules are not derived products. If you or your employer wish to distribute kernel modules under a nonfree license, you really need

* Note that there's no guarantee on even-numbered kernels as well, unless you rely on a commercial provider that grants its own warranty.

to discuss the situation with your legal counsel. Please note also that the kernel developers have no qualms against breaking binary modules between kernel releases, even in the middle of a stable kernel series. If it is at all possible, both you and your users are better off if you release your module as free software.

If you want your code to go into the mainline kernel, or if your code requires patches to the kernel, you *must* use a GPL-compatible license as soon as you release the code. Although personal use of your changes doesn't force the GPL on you, if you distribute your code, you must include the source code in the distribution—people acquiring your package must be allowed to rebuild the binary at will.

As far as this book is concerned, most of the code is freely redistributable, either in source or binary form, and neither we nor O'Reilly retain any right on any derived works. All the programs are available at *ftp://ftp.ora.com/pub/examples/linux/drivers/*, and the exact license terms are stated in the *LICENSE* file in the same directory.

Joining the Kernel Development Community

As you begin writing modules for the Linux kernel, you become part of a larger community of developers. Within that community, you can find not only people engaged in similar work, but also a group of highly committed engineers working toward making Linux a better system. These people can be a source of help, ideas, and critical review as well—they will be the first people you will likely turn to when you are looking for testers for a new driver.

The central gathering point for Linux kernel developers is the *linux-kernel* mailing list. All major kernel developers, from Linus Torvalds on down, subscribe to this list. Please note that the list is not for the faint of heart: traffic as of this writing can run up to 200 messages per day or more. Nonetheless, following this list is essential for those who are interested in kernel development; it also can be a top-quality resource for those in need of kernel development help.

To join the linux-kernel list, follow the instructions found in the linux-kernel mailing list FAQ: *http://www.tux.org/lkml*. Read the rest of the FAQ while you are at it; there is a great deal of useful information there. Linux kernel developers are busy people, and they are much more inclined to help people who have clearly done their homework first.

Overview of the Book

From here on, we enter the world of kernel programming. Chapter 2 introduces modularization, explaining the secrets of the art and showing the code for running modules. Chapter 3 talks about char drivers and shows the complete code for a

memory-based device driver that can be read and written for fun. Using memory as the hardware base for the device allows anyone to run the sample code without the need to acquire special hardware.

Debugging techniques are vital tools for the programmer and are introduced in Chapter 4. Equally important for those who would hack on contemporary kernels is the management of concurrency and race conditions. Chapter 5 concerns itself with the problems posed by concurrent access to resources and introduces the Linux mechanisms for controlling concurrency.

With debugging and concurrency management skills in place, we move to advanced features of char drivers, such as blocking operations, the use of *select*, and the important *ioctl* call; these topics are the subject of Chapter 6.

Before dealing with hardware management, we dissect a few more of the kernel's software interfaces: Chapter 7 shows how time is managed in the kernel, and Chapter 8 explains memory allocation.

Next we focus on hardware. Chapter 9 describes the management of I/O ports and memory buffers that live on the device; after that comes interrupt handling, in Chapter 10. Unfortunately, not everyone is able to run the sample code for these chapters, because some hardware support *is* actually needed to test the software interface interrupts. We've tried our best to keep required hardware support to a minimum, but you still need some simple hardware, such as a standard parallel port, to work with the sample code for these chapters.

Chapter 11 covers the use of data types in the kernel and the writing of portable code.

The second half of the book is dedicated to more advanced topics. We start by getting deeper into the hardware and, in particular, the functioning of specific peripheral buses. Chapter 12 covers the details of writing drivers for PCI devices, and Chapter 13 examines the API for working with USB devices.

With an understanding of peripheral buses in place, we can take a detailed look at the Linux device model, which is the abstraction layer used by the kernel to describe the hardware and software resources it is managing. Chapter 14 is a bottom-up look at the device model infrastructure, starting with the kobject type and working up from there. It covers the integration of the device model with real hardware; it then uses that knowledge to cover topics like hot-pluggable devices and power management.

In Chapter 15, we take a diversion into Linux memory management. This chapter shows how to map kernel memory into user space (the *mmap* system call), map user memory into kernel space (with *get_user_pages*), and how to map either kind of memory into device space (to perform direct memory access [DMA] operations).

Our understanding of memory will be useful for the following two chapters, which cover the other major driver classes. Chapter 16 introduces block drivers and shows how they are different from the char drivers we have worked with so far. Then Chapter 17 gets into the writing of network drivers. We finish up with a discussion of serial drivers (Chapter 18) and a bibliography.