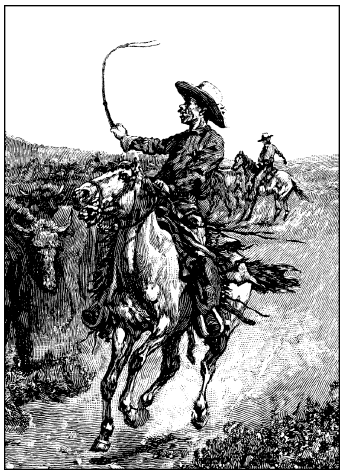
CHAPTER ONE

AN INTRODUCTION TO DEVICE DRIVERS



As the popularity of the Linux system continues to grow, the interest in writing Linux device drivers steadily increases. Most of Linux is independent of the hardware it runs on, and most users can be (happily) unaware of hardware issues. But, for each piece of hardware supported by Linux, somebody somewhere has written a driver to make it work with the system. Without device drivers, there is no functioning system.

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 will teach 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 will 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 will give you a handle on the background you need to make your device work.

As you learn to write drivers, you will 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'll introduce new ideas gradually, starting off with very simple drivers and building upon them; every new concept will be 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 will be able to make your own choices about your driver, choosing 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 in which you have to do the writing 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'll try not to deal with policy issues, or with application programs or support libraries. Sometimes we'll 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. Though 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 exotic 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

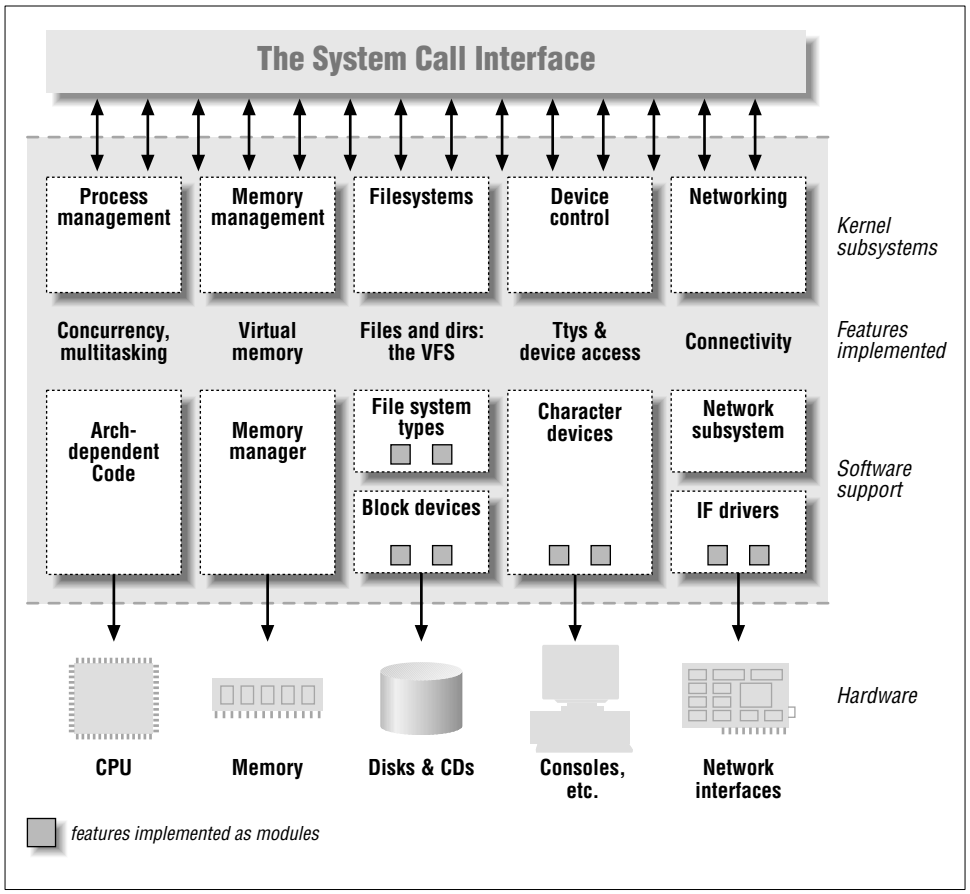


Figure 1-1. A split view of the kernel

throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, diskettes may be formatted with either the Linux-standard ext2 filesystem or with the commonly used FAT filesystem.

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 streamer. 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.

Toward the end of this book, in Chapter 16, you'll find a road map to the Linux kernel, but these few paragraphs should suffice for now.

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 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 Unix way of looking at devices distinguishes between three 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.

The three classes are the following:

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 something that can host a filesystem, such as a disk. In most Unix systems, a block device can be accessed only as multiples of a block, where a block is usually one kilobyte of data or another power of 2. Linux 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. A block driver offers the kernel the same interface as a char driver, as well as an additional block-oriented interface that is invisible to the user or applications opening the */dev* entry points. That block interface, though, is essential to be able to *mount* a filesystem.

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. Though both Telnet and FTP connections are stream oriented, they transmit using the same device; the device doesn't see the individual streams, but only the data 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.

Other classes of driver modules exist in Linux. The modules in each class exploit public services the kernel offers to deal with specific types of devices. Therefore,

one can talk of universal serial bus (USB) modules, serial modules, and so on. The most common nonstandard class of devices is that of SCSI* drivers. Although every peripheral connected to the SCSI bus appears in */dev* as either a char device or a block device, the internal organization of the software is different.

Just as network interface cards provide the network subsystem with hardware-related functionality, so a SCSI controller provides the SCSI subsystem with access to the actual interface cable. SCSI is a communication protocol between the computer and peripheral devices, and every SCSI device responds to the same protocol, independently of what controller board is plugged into the computer. The Linux kernel therefore embeds a SCSI *implementation* (i.e., the mapping of file operations to the SCSI communication protocol). The driver writer has to implement the mapping between the SCSI abstraction and the physical cable. This mapping depends on the SCSI controller and is independent of the devices attached to the SCSI cable.

Other classes of device drivers have been added to the kernel in recent times, including USB drivers, FireWire drivers, and I2O drivers. In the same way that they handled 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. Beyond device drivers, filesystems are perhaps the most important class of modules in the Linux system. 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 higher-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,

* SCSI is an acronym for Small Computer Systems Interface; it is an established standard in the workstation and high-end server market.

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.

Security has two faces, which can be called *deliberate* and *incidental*. One security problem is the damage a user can cause through the misuse of existing programs, or by incidentally exploiting bugs; a different issue is what kind of (mis)functionality a programmer can deliberately implement. The programmer has, obviously, much more power than a plain user. In other words, it's as dangerous to run a program you got from somebody else from the root account as it is to give him or her a root shell now and then. Although having access to a compiler is not a security hole per se, the hole can appear when compiled code is actually executed; everyone should be careful with modules, because a kernel module can do anything. A module is just as powerful as a superuser shell.

Any security check in the system is enforced by kernel code. If the kernel has security holes, then the system has holes. In the official kernel distribution, only an authorized user can load modules; the system call *create_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. 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) 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

* Version 2.0 of the kernel allows only the superuser to run privileged code, while version 2.2 has more sophisticated capability checks. We discuss this in "Capabilities and Restricted Operations" in Chapter 5.

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 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, formatting a disk) that could affect the system, those operations should probably 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 *create_module*.

Note that the Linux kernel can be compiled to have no module support whatsoever, thus closing any 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'd like to 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. Fortunately, almost all modern distributions support the upgrade of single packages by checking interpackage dependencies; the distribution's package manager generally will 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 but the kernel; 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 problem.

As far as the kernel is concerned, the even-numbered kernel versions (i.e., 2.2.*x* and 2.4.*x*) are the stable ones that are intended for general distribution. The odd versions (such as 2.3.*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 versions 2.0 through 2.4 of the kernel. Our focus has been to show all the features available to device driver writers in 2.4, the current version at the time we are writing. We also try to cover 2.2 thoroughly, in those areas where the features differ between 2.2 and 2.4. We also note features that are not available in 2.0, and offer workarounds where space permits. In general, the code we show is designed to compile and run on a wide range of kernel versions; in particular, it has all been tested with version 2.4.4, and, where applicable, with 2.2.18 and 2.0.38 as well.

This text doesn't talk specifically about odd-numbered kernel versions. General users will never have a reason to run development kernels. Developers experimenting with new features, however, will want to be running the latest development release. They will 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 will help 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 is successfully being used with Alpha and SPARC processors, 68000 and PowerPC platforms, as well as a few more. This book is platform independent as far as possible, and all the code samples have been tested on several platforms, such as the PC brands, Alpha, ARM, IA-64, M68k, PowerPC, SPARC, SPARC64, and VR41xx (MIPS). 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.

---

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

License Terms

Linux is licensed with 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 is allowed to rebuild an exact copy of the binary files from source. 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 directory */usr/src/linux*, as a file called *COPYING*.

Third-party and custom modules are not part of the Linux kernel, and thus you're not forced to license them under the GPL. A module *uses* the kernel through a well-defined interface, but is not part of it, similar to the way user programs use the kernel through system calls. Note that the exemption to GPL licensing applies only to modules that use only the published module interface. Modules that dig deeper into the kernel must adhere to the "derived work" terms of the GPL.

In brief, if your code goes in the kernel, you must use the GPL 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. If you write a module, on the other hand, you are allowed to distribute it in binary form. However, this is not always practical, as modules should in general be recompiled for each kernel version that they will be linked with (as explained in Chapter 2, in the section "Version Dependency," and Chapter 11, in the section "Version Control in Modules"). New kernel releases—even minor stable releases—often break compiled modules, requiring a recompile. Linus Torvalds has stated publicly that he has no problem with this behavior, and that binary modules should be expected to work only with the kernel under which they were compiled. As a module writer, you will generally serve your users better by making source available.

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 & Associates retain any right on any derived works. All the programs are available through FTP from *ftp://ftp.ora.com/pub/examples/linux/drivers/*, and the exact license terms are stated in the file *LICENSE* in the same directory.

When sample programs include parts of the kernel code, the GPL applies: the comments accompanying source code are very clear about that. This only happens for a pair of source files that are very minor to the topic of this book.

Joining the Kernel Development Community

As you get into 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, of ideas, and of 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*. Please 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. Then, with our new debugging skills, 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 5.

Before dealing with hardware management, we dissect a few more of the kernel's software interfaces: Chapter 6 shows how time is managed in the kernel, and Chapter 7 explains memory allocation.

Next we focus on hardware. Chapter 8 describes the management of I/O ports and memory buffers that live on the device; after that comes interrupt handling, in Chapter 9. Unfortunately, not everyone will be able to run the sample code for these chapters, because some hardware support *is* actually needed to test the software interface to interrupts. We've tried our best to keep required hardware support to a minimum, but you still need to put your hands on the soldering iron to build your hardware "device." The device is a single jumper wire that plugs into the parallel port, so we hope this is not a problem.

Chapter 10 offers some additional suggestions about writing kernel software and about portability issues.

In the second part of this book, we get more ambitious; thus, Chapter 11 starts over with modularization issues, going deeper into the topic.

Chapter 12 then describes how block drivers are implemented, outlining the aspects that differentiate them from char drivers. Following that, Chapter 13 explains what we left out from the previous treatment of memory management: *mmap* and direct memory access (DMA). At this point, everything about char and block drivers has been introduced.

The third main class of drivers is introduced next. Chapter 14 talks in some detail about network interfaces and dissects the code of the sample network driver.

A few features of device drivers depend directly on the interface bus where the peripheral fits, so Chapter 15 provides an overview of the main features of the bus implementations most frequently found nowadays, with a special focus on PCI and USB support offered in the kernel.

Finally, Chapter 16 is a tour of the kernel source: it is meant to be a starting point for people who want to understand the overall design, but who may be scared by the huge amount of source code that makes up Linux.