CHAPTER SIXTEEN

PHYSICAL LAYOUT OF THE KERNEL SOURCE

So far, we've talked about the Linux kernel from the perspective of writing device drivers. Once you begin playing with the kernel, however, you may find that you want to "understand it all." In fact, you may find yourself passing whole days navigating through the source code and grepping your way through the source tree to uncover the relationships among the different parts of the kernel.

This kind of "heavy grepping" is one of the tasks your authors perform quite often, and it is an efficient way to retrieve information from the source code. Nowadays you can even exploit Internet resources to understand the kernel source tree; some of them are listed in the Preface. But despite Internet resources, wise use of *grep*,* *less*, and possibly *ctags* or *etags* can still be the best way to extract information from the kernel sources.

In our opinion, acquiring a bit of a knowledge base before sitting down in front of your preferred shell prompt can be helpful. Therefore, this chapter presents a quick overview of the Linux kernel source files based on version 2.4.2. If you're interested in other versions, some of the descriptions may not apply literally. Whole sections may be missing (like the *drivers/media* directory that was introduced in 2.4.0-test6 by moving various preexisting drivers to this new directory). We hope the following information is useful, even if not authoritative, for browsing other versions of the kernel.

Every pathname is given relative to the source root (usually */usr/src/linux*), while filenames with no directory component are assumed to reside in the "current" directory—the one being discussed. Header files (when named with < and > angle brackets) are given relative to the *include* directory of the source tree. We won't dissect the *Documentation* directory, as its role is self-explanatory.

* Usually, *find* and *xargs* are needed to build a command line for *grep*. Although not trivial, proficient use of Unix tools is outside of the scope of this book.

Booting the Kernel

The usual way to look at a program is to start where execution begins. As far as Linux is concerned, it's hard to tell *where* execution begins—it depends on how you define "begins."

The architecture-independent starting point is *start_kernel* in *init/main.c*. This function is invoked from architecture-specific code, to which it never returns. It is in charge of spinning the wheel and can thus be considered the "mother of all functions," the first breath in the computer's life. Before *start_kernel*, there was chaos.

By the time *start_kernel* is invoked, the processor has been initialized, protected mode* has been entered, the processor is executing at the highest privilege level (sometimes called *supervisor mode*), and interrupts are disabled. The *start_kernel* function is in charge of initializing all the kernel data structures. It does this by calling external functions to perform subtasks, since each setup function is defined in the appropriate kernel subsystem.

The first function called by *start_kernel*, after acquiring the kernel lock and printing the Linux banner string, is *setup_arch*. This allows platform-specific C-language code to run; *setup_arch* receives a pointer to the local `command_line` pointer in *start_kernel*, so it can make it point to the real (platform-dependent) location where the command line is stored. As the next step, *start_kernel* passes the command line to *parse_options* (defined in the same *init/main.c* file) so that the boot options can be honored.

Command-line parsing is performed by calling handler functions associated with each kernel argument (for example, `video=` is associated with *video_setup*). Each function usually ends up setting variables that are used later, when the associated facility is initialized. The internal organization of command-line parsing is similar to the init calls mechanism, described later.

After parsing, *start_kernel* activates the various basic functionalities of the system. This includes setting up interrupt tables, activating the timer interrupt, and initializing the console and memory management. All of this is performed by functions declared elsewhere in platform-specific code. The function continues by initializing less basic kernel subsystems, including buffer management, signal handling, and file and inode management.

Finally, *start_kernel* forks the *init* kernel thread (which gets 1 as a process ID) and executes the *idle* function (again, defined in architecture-specific code).

The initial boot sequence can thus be summarized as follows:

---

* This concept only makes sense on the x86 architecture. More mature architectures don't find themselves in a limited backward-compatible mode when they power up.

1. System firmware or a boot loader arranges for the kernel to be placed at the proper address in memory. This code is usually external to Linux source code.
2. Architecture-specific assembly code performs very low-level tasks, like initializing memory and setting up CPU registers so that C code can run flawlessly. This includes selecting a stack area and setting the stack pointer accordingly. The amount of such code varies from platform to platform; it can range from a few dozen lines up to a few thousand lines.
3. *start_kernel* is called. It acquires the kernel lock, prints the banner, and calls *setup_arch*.
4. Architecture-specific C-language code completes low-level initialization and retrieves a command line for *start_kernel* to use.
5. *start_kernel* parses the command line and calls the handlers associated with the keyword it identifies.
6. *start_kernel* initializes basic facilities and forks the *init* thread.

It is the task of the *init* thread to perform all other initialization. The thread is part of the same *init/main.c* file, and the bulk of the initialization (init) calls are performed by *do_basic_setup*. The function initializes all bus subsystems that it finds (PCI, SBus, and so on). It then invokes *do_initcalls*; device driver initialization is performed as part of the *initcall* processing.

The idea of init calls was added in version 2.3.13 and is not available in older kernels; it is designed to avoid hairy `#ifdef` conditionals all over the initialization code. Every optional kernel feature (device driver or whatever) must be initialized only if configured in the system, so the call to initialization functions used to be surrounded by `#ifdef CONFIG_`*`FEATURE`* and `#endif`. With init calls, each optional feature declares its own initialization function; the compilation process then places a reference to the function in a special ELF section. At boot time, *do_initcalls* scans the ELF section to invoke all the relevant initialization functions.

The same idea is applied to command-line arguments. Each driver that can receive a command-line argument at boot time defines a data structure that associates the argument with a function. A pointer to the data structure is placed into a separate ELF section, so *parse_option* can scan this section for each command-line option and invoke the associated driver function, if a match is found. The remaining arguments end up in either the environment or the command line of the *init* process. All the magic for init calls and ELF sections is part of `<linux/init.h>`.

Unfortunately, this init call idea works only when no ordering is required across the various initialization functions, so a few `#ifdef`s are still present in *init/main.c*.

It's interesting to see how the idea of init calls and its application to the list of command-line arguments helped reduce the amount of conditional compilation in the code:

```
morgana% grep -c ifdef linux-2.[024]/init/main.c
linux-2.0/init/main.c:120
linux-2.2/init/main.c:246
linux-2.4/init/main.c:35
```

Despite the huge addition of new features over time, the amount of conditional compilation dropped significantly in 2.4 with the adoption of init calls. Another advantage of this technique is that device driver maintainers don't need to patch *main.c* every time they add support for a new command-line argument. The addition of new features to the kernel has been greatly facilitated by this technique and there are no more hairy cross references all over the boot code. But as a side effect, 2.4 can't be compiled into older file formats that are less flexible than ELF. For this reason, *uClinux** developers switched from COFF to ELF while porting their system from 2.0 to 2.4.

Another side effect of extensive use of ELF sections is that the final pass in compiling the kernel is not a conventional link pass as it used to be. Every platform now defines exactly how to link the kernel image (the *vmlinux* file) by means of an *ldscript* file; the file is called *vmlinux.lds* in the source tree of each platform. Use of *ld* scripts is described in the standard documentation for the *binutils* package.

There is yet another advantage to putting the initialization code into a special section. Once initialization is complete, that code is no longer needed. Since this code has been isolated, the kernel is able to dump it and reclaim the memory it occupies.

Before Booting

In the previous section, we treated *start_kernel* as the first kernel function. However, you might be interested in what happens *before* that point, so we'll step back to take a quick look at that topic. The uninterested reader can jump directly to the next section.

As suggested, the code that runs before *start_kernel* is, for the most part, assembly code, but several platforms call library C functions from there (most commonly, *inflate*, the core of *gunzip*).

On most common platforms, the code that runs before *start_kernel* is mainly devoted to moving the kernel around after the computer's firmware (possibly with

* *uClinux* is a version of the Linux kernel that can run on processors without an MMU. This is typical in the embedded world, and several M68k and ARM processors have no hardware memory management. *uClinux* stands for microcontroller Linux, since it's meant to run on microcontrollers rather than full-fledged computers.

the help of a boot loader) has loaded it into RAM from some other storage, such as a local disk or a remote workstation over the network.

It's not uncommon, though, to find some rudimentary boot loader code inside the *boot* directory of an architecture-specific tree. For example, *arch/i386/boot* includes code that can load the rest of the kernel off a floppy disk and activate it. The file *bootsect.S* that you will find there, however, can run only off a floppy disk and is by no means a complete boot loader (for example, it is unable to pass a command line to the kernel it loads). Nonetheless, copying a new kernel to a floppy is still a handy way to quickly boot it on the PC.

A known limitation of the x86 platform is that the CPU can see only 640 KB of system memory when it is powered on, no matter how large your installed memory is. Dealing with the limitation requires the kernel to be compressed, and support for decompression is available in *arch/i386/boot* together with other code such as VGA mode setting. On the PC, because of this limit, you can't do anything with a *vmlinux* kernel image, and the file you actually boot is called *zImage* or *bzImage*; the boot sector described earlier is actually prepended to this file rather than to *vmlinux*. We won't spend more time on the booting process on the x86 platform, since you can choose from several boot loaders, and the topic is generally well discussed elsewhere.

Some platforms differ greatly in the layout of their boot code from the PC. Sometimes the code must deal with several variations of the same architecture. This is the case, for example, with ARM, MIPS, and M68k. These platforms cover a wide variety of CPU and system types, ranging from powerful servers and workstations down to PDAs or embedded appliances. Different environments require different boot code and sometimes even different *ld* scripts to compile the kernel image. Some of this support is not included in the official kernel tree published by Linus and is available only from third-party Concurrent Versions System (CVS) trees that closely track the official tree but have not yet been merged. Current examples include the SGI CVS tree for MIPS workstations and the LinuxCE CVS tree for MIPS-based palm computers. Nonetheless, we'd like to spend a few words on this topic because we feel it's an interesting one. Everything from *start_kernel* onward is based on this extra complexity but doesn't notice it.

Specific *ld* scripts and makefile rules are needed especially for embedded systems, and particularly for variants without a memory management unit, which are supported by *uClinux*. When you have no hardware MMU that maps virtual addresses to physical ones, you must link the kernel to be executed from the physical address where it will be loaded in the target platform. It's not uncommon in small systems to link the kernel so that it is loaded into read-only memory (usually flash memory), where it is directly activated at power-on time without the help of any boot loader.

When the kernel is executed directly from flash memory, the makefiles, *ld* scripts, and boot code work in tight cooperation. The *ld* rules place the code and read-only segments (such as the init calls information) into flash memory, while placing the data segments (data and block started by symbol (BSS)) in system RAM. The result is that the two sets are not consecutive. The makefile, then, offers special rules to coalesce all these sections into consecutive addresses and convert them to a format suitable for upload to the target system. Coalescing is mandatory because the data segment contains initialized data structures that must get written to read-only memory or otherwise be lost. Finally, assembly code that runs before *start_kernel* must copy over the data segment from flash memory to RAM (to the address where the linker placed it) and zero out the address range associated with the BSS segment. Only after this remapping has taken place can C-language code run.

When you upload a new kernel to the target system, the firmware there retrieves the data file from the network or from a serial channel and writes it to flash memory. The intermediate format used to upload the kernel to a target computer varies from system to system, because it depends on how the actual upload takes place. But in each case, this format is a generic container of binary data used to transfer the compiled image using standardized tools. For example, the BIN format is meant to be transferred over a network, while the S3 format is a hexadecimal ASCII file sent to the target system through a serial cable.* Most of the time, when powering on the system, the user can select whether to boot Linux or to type firmware commands.

The init Process

When *start_kernel* forks out the *init* thread (implemented by the *init* function in *init/main.c*), it is still running in kernel mode, and so is the *init* thread. When all initializations described earlier are complete, the thread drops the kernel lock and prepares to execute the user-space *init* process. The file being executed resides in */sbin/init*, */etc/init*, or */bin/init*. If none of those are found, */bin/sh* is run as a recovery measure in case the real *init* got lost or corrupted. As an alternative, the user can specify on the kernel command line which file the *init* thread should execute.

The procedure to enter user space is simple. The code opens */dev/console* as standard input by calling the *open* system call and connects the console to *stdout* and *stderr* by calling *dup*; it finally calls *execve* to execute the user-space program.

The thread is able to invoke system calls while running in kernel mode because *init/main.c* has declared `__KERNEL_SYSCALLS__` before including `<asm/unistd.h>`. The header defines special code that allows kernel code to

* We are not describing the formats or the tools in detail, because the information is readily available to people researching embedded Linux.

invoke a limited number of system calls just as if it were running in user space. More information about kernel system calls can be found in *http://www.linux.it/kerneldocs/ksys*.

The final call to *execve* finalizes the transition to user space. There is no magic involved in this transition. As with any *execve* call in Unix, this one replaces the memory maps of the current process with new memory maps defined by the binary file being executed (you should remember how executing a file means mapping it to the virtual address space of the current process). It doesn't matter that, in this case, the calling process is running in kernel space. That's transparent to the implementation of *execve*, which just finds that there are no previous memory maps to release before activating the new ones.

Whatever the system setup or command line, the *init* process is now executing in user space and any further kernel operation takes place in response to system calls coming from *init* itself or from the processes it forks out.

More information about how the *init* process brings up the whole system can be found in *http://www.linux.it/kerneldocs/init*. We'll now proceed on our tour by looking at the system calls implemented in each source directory, and then at how device drivers are laid out and organized in the source tree.

The kernel Directory

Some kernel facilities—those associated with filesystems, memory management, and networking—live in their own source trees. The *kernel* directory of the source tree includes all other basic facilities.

The most important such facility is scheduling. Thus, *sched.c*, together with `<linux/sched.h>`, can be considered the most important source file in the Linux kernel. In addition to the scheduler proper, implemented by *schedule*, the file defines the system calls that control process priorities and all the mechanisms for sleeping and waking.

The *fork* and *exit* system calls are implemented by two files that are named after them. They are comprehensive and well-structured files that deal with everything related to process creation and destruction.

The delivery of kernel messages is implemented in *printk.c*, which is also concerned with console management. Console code is not trivial, since the concept of "console" is pretty abstract nowadays and includes the text screen (either native or based on the frame buffer), the serial port, and even the printer port.

Other facilities that are implemented in this directory are time handling (*time.c*), kernel timers (*timer.c*), signal delivery and handling (*signal.c*), module management and related system calls (*module.c*), the *kmod* thread (*kmod.c*), systemwide power management (*pm.c*), tasklets (*softirq.c*), and the panic function (*panic.c*).

The fs Directory

File handling is at the core of any Unix system, and the *fs* directory in Linux is the fattest of all directories. It includes all the filesystems supported by the current Linux version, each in its own subdirectory, as well as the most important system calls after *fork* and *exit*.

The *execve* system call lives in *exec.c* and relies on the various available binary formats to actually interpret the binary data found in the executable files. The most important binary format nowadays is ELF, implemented by *binfmt_elf.c*. *binfmt_script.c* supports the execution of interpreted files. After detecting the need for an interpreter (usually on the `#!` or "shebang" line), the file relies on the other binary formats to load the interpreter.

Miscellaneous binary formats (such as the Java executable format) can be defined by the user with a */proc* interface defined in *binfmt_misc.c*. The *misc* binary format is able to identify an interpreted binary format based on the contents of the executable file, and fire the appropriate interpreter with appropriate arguments. The tool is configured via */proc/sys/fs/binfmt_misc*.

The fundamental system calls for file access are defined in *open.c* and *read_write.c*. The former also defines *close* and several other file-access system calls (*chown*, for instance). *select.c* implements *select* and *poll*. *pipe.c* and *fifo.c* implement pipes and named pipes. *readdir.c* implements the *getdents* system call, which is how user-space programs read directories (the name stands for "get directory entries"). Other programming interfaces to access directory data (such as the *readdir* interface) are all implemented in user space as library functions, based on the *getdents* system call.

Most system calls related to moving files around, such as *mkdir*, *rmdir*, *rename*, *link*, *symlink*, and *mknod*, are implemented in *namei.c*, which in turn lays its foundations on the directory entry cache that lives in *dcache.c*.

Mounting and unmounting filesystems, as well as support for the use of a temporary root for *initrd*, are implemented in *super.c*.

Of particular interest to device driver writers is *devices.c*, which implements the char and block driver registries and acts as dispatcher for all devices. It does so by implementing the generic *open* method that is used before the device-specific `file_operations` structure is fetched and used. *read* and *write* for block devices are implemented in *block_dev.c*, which in turn delegates to *buffer.c* everything related to buffer management.

There are several other files in this directory, but they are less interesting. The most important ones are *inode.c* and *file.c*, which manage the internal organization of file and inode data structures; *ioctl.c*, which implements *ioctl*; and *dquot.c*, which implements quotas.

As we suggested, most of the subdirectories of *fs* host individual filesystem implementations. However, *fs/partitions* is not a filesystem type but rather a container for partition management code. Some files in there are always compiled, regardless of kernel configuration, while other files that implement support for specific partitioning schemes can be individually enabled or disabled.

The mm Directory

The last major directory of kernel source files is devoted to memory management. The files in this directory implement all the data structures that are used throughout the system to manage memory-related issues. While memory management is founded on registers and features specific to a given CPU, we've already seen in Chapter 13 how most of the code has been made platform independent. Interested users can check how *asm/arch-*`arch`*/mm* implements the lowest level for a specific computer platform.

The *kmalloc*/*kfree* memory allocation engine is defined in *slab.c*. This file is a completely new implementation that replaces what used to live in *kmalloc.c*. The latter file doesn't exist anymore after version 2.0.

While most programmers are familiar with how an operating system manages memory in blocks and pages, Linux (taking an idea from Sun Microsystem's Solaris) uses an additional, more flexible concept called a *slab*. Each slab is a cache that contains multiple memory objects of the same size. Some slabs are specialized and contain structs of a certain type used by a certain part of the kernel; others are more general and contain memory regions of 32 bytes, 64 bytes, and so on. The advantage of using slabs is that structs or other regions of memory can be cached and reused with very little overhead; the more ponderous technique of allocating and freeing pages is invoked less often.

The other important allocation tool, *vmalloc*, and the function that lies behind them all, *get_free_pages*, are defined in *vmalloc.c* and *page_alloc.c* respectively. Both are pretty straightforward and make interesting reading.

In addition to allocation services, a memory management system must offer memory mappings. After all, *mmap* is the foundation of many system activities, including the execution of a file. The actual *sys_mmap* function doesn't live here, though. It is buried in architecture-specific code, because system calls with more than five arguments need special handling in relation to CPU registers. The function that implements *mmap* for all platforms is *do_mmap_pgoff*, defined in *mmap.c*. The same file implements *sys_sendfile* and *sys_brk*. The latter may look unrelated, because *brk* is used to raise the maximum virtual address usable by a process. Actually, Linux (and most current Unices) creates new virtual address space for a process by mapping pages from */dev/zero*.

The mechanisms for mapping a regular file into memory have been placed in *filemap.c*; the file acts on pretty low-level data structures within the memory management system. *mprotect* and *remap* are implemented in two files of the same names; memory locking appears in *mlock.c*.

When a process has several memory maps active, you need an efficient way to look for free areas in its memory address space. To this end, all memory maps of a process are laid out in an Adelson-Velski-Landis (AVL) tree. The software structure is implemented in *mmap_avl.c*.

Swap file initialization and removal (i.e., the *swapon* and *swapoff* system calls) are in *swapfile.c*. The scope of *swap_state.c* is the swap cache, and page aging is in *swap.c*. What is known as *swapping* is not defined here. Instead, it is part of managing memory pages, implemented by the *kswapd* thread.

The lowest level of page-table management is implemented by the *memory.c* file, which still carries the original notes by Linus when he implemented the first real memory management features in December 1991. Everything that happens at lower levels is part of architecture-specific code (often hidden as macros in the header files).

Code specific to high-memory management (the memory beyond that which can be addressed directly by the kernel, especially used in the x86 world to accommodate more than 4 GB of RAM without abandoning the 32-bit architecture) is in *highmem.c*, as you may imagine.

vmscan.c implements the *kswapd* kernel thread. This is the procedure that looks for unused and old pages in order to free them or send them to swap space, as already suggested. It's a well-commented source file because fine-tuning these algorithms is the key factor to overall system performance. Every design choice in this nontrivial and critical section needs to be well motivated, which explains the good amount of comments.

The rest of the source files found in the *mm* directory deal with minor but sometimes important details, like the *oom_killer*, a procedure that elects which process to kill when the system runs out of memory.

Interestingly, the *uClinux* port of the Linux kernel to MMU-less processors introduces a separate *mmnommu* directory. It closely replicates the official *mm* while leaving out any MMU-related code. The developers chose this path to avoid adding a mess of conditional code in the *mm* source tree. Since *uClinux* is not (yet) integrated with the mainstream kernel, you'll need to download a *uClinux* CVS tree or tar ball if you want to compare the two directories (both included in the *uClinux* tree).

The net directory

The *net* directory in the Linux file hierarchy is the repository for the socket abstraction and the network protocols; these features account for a lot of code, since Linux supports several different network protocols. Each protocol (IP, IPX, and so on) lives in its own subdirectory; the directory for IP is called *ipv4* because it represents version 4 of the protocol. The new standard (not yet in wide use as we write this) is called *ipv6* and is implemented in Linux as well. Unix-domain sockets are treated as just another network protocol; their implementation can be found in the *unix* subdirectory.

The network implementation in Linux is based on the same file operations that act on device files. This is natural, because network connections (sockets) are described by normal file descriptors. The file *socket.c* is the locus of the socket file operations. It dispatches the system calls to one of the network protocols via a `struct proto_ops` structure. This structure is defined by each network protocol to map system calls to its specific, low-level data handling operations.

Not every subdirectory of *net* is used to define a protocol family. There are a few notable exceptions: *core*, *bridge*, *ethernet*, *sunrpc*, and *khttpd*.

Files in *core* implement generic network features such as device handling, firewalls, multicasting, and aliases; this includes the handling of socket buffers (*core/skbuff.c*) and socket operations that remain independent of the underlying protocol (*core/sock.c*). The device-independent data management that sits near device-specific code is defined in *core/dev.c*.

The *ethernet* and *bridge* directories are used to implement specific low-level functionalities, specifically, the Ethernet-related helper functions described in Chapter 14, and bridging functionality.

sunrpc and *khttpd* are peculiar because they include kernel-level implementations of tasks that are usually carried out in user space.

In *sunrpc* you can find support functions for the kernel-level NFS server (which is an RPC-based service), while *khttpd* implements a kernel-space web server. Those services have been brought to kernel space to avoid the overhead of system calls and context switches during time-critical tasks. Both have demonstrated good performance in this mode. The *khttpd* subsystem, however, has already been rendered obsolete by *TUX*, which, as of this writing, holds the record for the world's fastest web server. *TUX* will likely be integrated into the 2.5 kernel series.

The two remaining source files within *net* are *sysctl_net.c* and *netsyms.c*. The former is the back end of the *sysctl* mechanism,* and the latter is just a list of

* *sysctl* has not been described in this book; interested readers can have a look at Alessandro's description of this mechanism at *http://www.linux.it/kerneldocs/sysctl*.

`EXPORT_SYMBOL` declarations. There are several such files all over the kernel, usually one in each major directory.

ipc and lib

The smallest directories (in size) in the Linux source tree are *ipc* and *lib*. The former is an implementation of the System V interprocess communication primitives, namely semaphores, message queues, and shared memory; they often get forgotten, but many applications use them (especially shared memory). The latter directory includes generic support functions, similar to the ones available in the standard C library.

The generic library functions are a very small subset of those available in user space, but cover the indispensable things you generally need to write code: string functions (including *simple_atol* to convert a string to a `long` integer with error checking) and `<ctype.h>` functions. The most important file in this directory is *vsprintf.c*; it implements the function by the same name, which sits at the core of *sprintf* and *printk*. Another important file is *inflate.c*, which includes the decompressing code of *gzip*.

include and arch

In a quick overview of the kernel source code, there's little to say about headers and architecture-specific code. Header files have been introduced all over the book, so their role (and the separation between *include/linux* and *include/asm*) should already be clear.

Architecture-specific code, on the other hand, has never been introduced in detail, but it doesn't easily lend itself to discussion. Inside each architecture's directory you usually find a file hierarchy similar to the top-level one (i.e., there are *mm* and *kernel* subdirectories), but also boot-related code and assembly source files. The most important assembly file within each supported architecture is called *kernel/entry.S*; it's the back end of the system call mechanism (i.e., the place where user processes enter kernel mode). Besides that, however, there's little in common across the various architectures, and describing them all would make no sense.

Drivers

Current Linux kernels support a huge number of devices. Device drivers account for half of the size of the source tree (actually two-thirds if you exclude architecture-specific code that you are not using). They account for almost 1500 C-language files and more than 800 headers.

The *drivers* directory itself doesn't host any source file, only subdirectories (and, obviously, a makefile).

Structuring the huge amount of source code is not easy, and the developers haven't followed any strict rules. The original division between *drivers/char* and *drivers/block* is inefficient nowadays, and more directories have been created according to several different requirements. Still, the most generic char and block drivers are found in *drivers/char* and *drivers/block*, so we'll start by visiting those two.

drivers/char

The *drivers/char* directory is perhaps the most important in the *drivers* hierarchy, because it hosts a lot of driver-independent code.

The generic tty layer (as well as line disciplines, tty software drivers, and similar features) is implemented in this directory. *console.c* defines the `linux` terminal type (by implementing its specific escape sequences and keyboard encoding). *vt.c* defines the virtual consoles, including code for switching from one virtual console to another. Selection support (the cut-and-paste capability of the Linux text console) is implemented by *selection.c*; the default line discipline is implemented by *n_tty.c*.

There are other files that, despite what you might expect, are device independent. *lp.c* implements a generic parallel port printer driver that includes a console-on-line-printer capability. It remains device independent by using the *parport* device driver to map operations to actual hardware (as seen in Figure 2-2). Similarly, *keyboard.c* implements the higher levels of keyboard handling; it exports the *handle_scancode* function so that platform-specific keyboard drivers (like *pc_keyb.c*, in the same directory) can benefit from generalized management. *mem.c* implements */dev/mem*, */dev/null*, and */dev/zero*, basic resources you can't do without.

Actually, since *mem.c* is never left out of the compilation process, it has been elected as the home of *chr_dev_init*, which in turn initializes several other device drivers if they have been selected for compilation.

There are other device-independent and platform-independent source files in *drivers/char*. If you are interested in looking at the role of each source file, the best place to start is the makefile for this directory, an interesting and pretty much self-explanatory file.

drivers/block

Like the preceding *drivers/char* directory, *drivers/block* has been present in Linux development for a long time. It used to host all block device drivers, and for this reason it included some device-independent code that is still present.

The most important file is *ll_rw_blk.c* (low-level read-write block). It implements all the request management functions that we described in Chapter 12.

A relatively new entry in this directory is *blkpg.c* (added as of 2.3.3). The file implements generic code for partition and geometry handling in block devices. Its code, together with the *fs/partitions* directory described earlier, replaces what was earlier part of "generic hard disk" support. The file called *genhd.c* still exists, but now includes only the generic initialization function for block drivers (similar to the one for char drivers that is part of *mem.c*). One of the public functions exported by *blkpg.c* is *blk_ioctl*, covered by "The ioctl Method" in Chapter 12.

The last device-independent file found in *drivers/block* is *elevator.o*. This file implements the mechanism to change the elevator function associated with a block device driver. The functionality can be exploited by means of *ioctl* commands briefly introduced in "The ioctl Method."

In addition to the hardware-dependent device drivers you would expect to find in *drivers/block*, the directory also includes software device drivers that are inherently cross-platform, just like the *sbull* and *spull* drivers that we introduced in this book. They are the RAM disk *rd.c*, the "network block device" *nbd.c*, and the loopback block device *loop.c*. The loopback device is used to mount files as if they were block devices. (See the manpage for *mount*, where it describes the *-o loop* option.) The network block device can be used to access remote resources as block devices (thus allowing, for example, a remote swap device).

Other files in the directory implement drivers for specific hardware, such as the various different floppy drives, the old-fashioned x86 XT disk controller, and a few more. Most of the important families of block drivers have been moved to a separate directory.

drivers/ide

The IDE family of device drivers used to live in *drivers/block* but has expanded to the point where they were moved into a separate directory. As a matter of fact, the IDE interface has been enhanced and extended over time in order to support more than just conventional hard disks. For example, IDE tapes are now supported as well.

The *drivers/ide* directory is a whole world of its own, with some generalized code and its own programming interface. You'll note in the directory some files that are just a few kilobytes long; they include only the IDE controller detection code, and rely on the generalized IDE driver for everything else. They are interesting reading if you are curious about IDE drivers.

drivers/md

This directory is concerned with implementing RAID functionality and the Logical Volume Manager abstraction. The code registers its own char and block major

numbers, so it can be considered a driver just like those traditional drivers; nonetheless, the code has been kept separate because it has nothing to do with direct hardware management.

drivers/cdrom

This directory hosts the generic CD-ROM interface. Both the IDE and SCSI *cdrom* drivers rely on *drivers/cdrom/cdrom.c* for some of their functionality. The main entry points to the file are *register_cdrom* and *unregister_cdrom*; the caller passes them a pointer to `struct cdrom_device_info` as the main object involved in CD-ROM management.

Other files in this directory are concerned with specific hardware drives that are neither IDE nor SCSI. Those devices are pretty rare nowadays, as they have been made obsolete by modern IDE controllers.

drivers/scsi

Everything related to the SCSI bus has always been placed in this directory. This includes both controller-independent support for specific devices (such as hard drives and tapes) and drivers for specific SCSI controller boards.

Management of the SCSI bus interface is scattered in several files: *scsi.c*, *hosts.c*, *scsi_ioctl.c*, and a dozen more. If you are interested in the whole list, you'd better browse the makefile, where `scsi_mod-objs` is defined. All public entry points to this group of files have been collected in *scsi_syms.c*.

Code that supports a specific type of hardware drive plugs into the SCSI core system by calling *scsi_register_module* with an argument of `MODULE_SCSI_DEV`. This is how disk support is added to the core system by *sd.c*, CD-ROM support by *sr.c* (which, internally, refers to the *cdrom_* class of functions), tape support by *st.c*, and generic devices by *sg.c*.

The "generic" driver is used to provide user-space programs with direct access to SCSI devices. The underlying device can be virtually anything; currently both CD burners and scanner programs rely on the SCSI generic device to access the hardware they drive. By opening the */dev/sg* devices, a user-space driver can do anything it needs without specific support in the kernel.

Host adapters (i.e., SCSI controller hardware) can be plugged into the core system by calling *scsi_register_module* with an argument of `MODULE_SCSI_HA`. Most drivers currently do that by using the *scsi_module.c* facility to register themselves: the driver's source file defines its (static) data structures and then includes *scsi_module.c*. This file defines standard initialization and cleanup functions, based on `<linux/init.h>` and the init calls mechanisms. This technique allows drivers to serve as either modules or compiled-in functions without any `#ifdef` lines.

Interestingly, one of the host adapters supported in *drivers/scsi* is the IDE SCSI emulation code, a software host adapter that maps to IDE devices. It is used, as an example, for CD mastering: the system sees all of the drives as SCSI devices, and the user-space program need only be SCSI aware.

Please note that several SCSI drivers have been contributed to Linux by the manufacturers rather than by your preferred hacker community; therefore not all of them are fun reading.

drivers/net

As you might expect, this directory is the home for most interface adapters. Unlike *drivers/scsi*, this directory doesn't include the actual communication protocols, which live in the top-level *net* directory tree. Nonetheless, there's still some bit of software abstraction implemented in *drivers/net*, namely, the implementation of the various line disciplines used by serial-based network communication.

The line discipline is the software layer responsible for the data that traverses the communication line. Every tty device has a line discipline attached. Each line discipline is identified by a number, and the number, as usual, is specified using a symbolic name. The default Linux line discipline is `N_TTY`, that is, the normal tty management routines, defined in *drivers/char/n_tty.c*.

When PPP, SLIP, or other communication protocols are concerned, however, the default line discipline must be replaced. User-space programs switch the discipline to `N_PPP` or `N_SLIP`, and the default will be restored when the device is finally closed. The reason that *pppd* and *slattach* don't exit, after setting up the communication link is just this: as soon as they exit, the device is closed and the default line discipline gets restored.

The job of initializing network drivers hasn't yet been transferred to the init calls mechanism, because some subtle technical details prevent the switch. Initialization is therefore still performed the old way: the *Space.c* file performs the initialization by scanning a list of known hardware and probing for it. The list is controlled by `#ifdef` directives that select which devices are actually included at compile time.

drivers/sound

Like *drivers/scsi* and *drivers/net*, this directory includes all the drivers for sound cards. The contents of the directory are somewhat similar to the SCSI directory: a few files make up the core sound system, and individual device drivers stack on top of it. The core sound system is in charge of requesting the major number `SOUND_MAJOR` and dispatching any use of it to the underlying device drivers. A hardware driver plugs into the core by calling *sound_install_audiodrv*, declared in *dev_table.c*.

The list of device-independent files in this directory is pretty long, since it includes generic support for mixers, generic support for sequencers, and so on. To those who want to probe further, we suggest using the makefile as a reference to what is what.

drivers/video

Here you find all the frame buffer video devices. The directory is concerned with video output, not video input. Like */drivers/sound*, the whole directory implements a single char device driver; a core frame buffer system dispatches actual access to the various frame buffers available on the computer.

The entry point to */dev/fb* devices is in *fbmem.c*. The file registers the major number and maintains an internal list of which frame buffer device is in charge of each minor number. A hardware driver registers itself by calling *register_framebuffer*, passing a pointer to `struct fb_info`. The data structure includes everything that's needed for specific device management. It includes the *open* and *release* methods, but no *read*, *write*, or *mmap*; these methods are implemented in a generalized way in *fbmem.c* itself.

In addition to frame buffer memory, this directory is in charge of frame buffer consoles. Because the layout of pixels in frame buffer memory is standardized to some extent, kernel developers have been able to implement generic console support for the various layouts of display memory. Once a hardware driver registers its own `struct fb_info`, it automatically gets a text console attached to it, according to its declared layout of video memory.

Unfortunately, there is no real standardization in this area, so the kernel currently supports 17 different screen layouts; they range from the fairly standard 16-bit and 32-bit color displays to the hairy VGA and Mac pixel placements. The files concerned with placing text on frame buffers are called *fbcon-`name`.c*.

When the first frame buffer device is registered, the function *register_framebuffer* calls *take_over_console* (exported by *drivers/char/console.c*) in order to actually set up the current frame buffer as the system console. At boot time, before frame buffer initialization, the console is either the native text screen or, if none is there, the first serial port. The command line starting the kernel, of course, can override the default by selecting a specific console device. Kernel developers created *take_over_console* to add support for frame buffer consoles without complicating the boot code. (Usually frame buffer drivers depend on PCI or equivalent support, so they can't be active too early during the boot process.) The *take_over_console* feature, however, is not limited to frame buffers; it's available to any code involving any hardware. If you want to transmit kernel messages using a Morse beeper or UDP network packets, you can do that by calling *take_over_console* from your kernel module.

drivers/input

Input management is another facility meant to simplify and standardize activities that are common to several drivers, and to offer a unified interface to user space. The core file here is called *input.c*. It registers itself as a char driver using `INPUT_MAJOR` as its major number. Its role is collecting events from low-level device drivers and dispatching them to higher layers.

The input interface is defined in `<linux/input.h>`. Each low-level driver registers itself by calling *input_register_device*. After registration, users are able to feed new events to the system by calling *input_event*.

Higher-level modules can register with *input.c* by calling *input_register_handler* and specifying what kind of events they are interested in. This is, for example, how *keybdev.c* expresses its interest in keyboard events (which it ultimately feeds to *driver/char/keyboard.c*).

A high-level module can also register its own minor numbers so it can use its own file operations and become the owner of an input-related special file in */dev*. Currently, however, third-party modules can't easily register minor numbers, and the feature can be used reliably only by the files in *drivers/input*. Minor numbers can currently be used to support mice, joysticks, and generic even channels in user space.

drivers/media

This directory, introduced as of version 2.4.0-test7, collects other communication media, currently radio and video input devices. Both the *media/radio* and *media/video* drivers currently stack on *video/videodev.c*, which implements the "Video For Linux" API.

video/videodev.c is a generic container. It requests a major number and makes it available to hardware drivers. Individual low-level drivers register by calling *video_register_device*. They pass a pointer to their own `struct video_device` and an integer that specifies the type of device. Supported devices are frame grabbers (`VFL_TYPE_GRABBER`), radios (`VFL_TYPE_RADIO`), teletext devices (`VFL_TYPE_VTX`), and undecoded vertical-blank information (`VFL_TYPE_VBI`).

Bus-Specific Directories

Some of the subdirectories of *drivers* are specific to devices that plug into a particular bus architecture. They have been separated from the generic *char* and *block* directories because quite a good deal of code is generic to the bus architecture (as opposed to specific to the hardware device).

The least populated of these directories is *drivers/pci*. It contains only code that talks with PCI controllers (or to system BIOS), whereas PCI hardware drivers are scattered all over the place. The PCI interface is so widespread that it makes no sense to relegate PCI cards to a specific place.

If you are wondering whether ISA has a specific directory, the answer is no. There are no specific ISA support files because the bus offers no resource management or standardization to build a software layer over it. ISA hardware drivers fit best in *drivers/char* or *drivers/sound* or elsewhere.

Other bus-specific directories range from less known internal computer buses to widely used external interface standards.

The former class includes *drivers/sbus*, *drivers/nubus*, *drivers/zorro* (the bus used in Amiga computers), *drivers/dio* (the bus of the HP300 class of computers), and *drivers/tc* (Turbo Channel, used in MIPS DECstations). Whereas *sbus* includes both SBus support functions and drivers for some SBus devices, the others include only support functions. Hardware drivers based on all of these buses live in *drivers/net*, *drivers/scsi*, or wherever is appropriate for the actual hardware (with the exception of a few SBus drivers, as noted). A few of these buses are currently used by just one driver.

Directories devoted to external buses include *drivers/usb*, *drivers/pcmcia*, *drivers/parport* (generic cross-platform parallel port support, which defines a whole new class of device drivers), *drivers/isdn* (all ISDN controllers supported by Linux and their common support functions), *drivers/atm* (the same, for ATM network connections), and *drivers/ieee1394* (FireWire).

Platform-Specific Directories

Sometimes, a computer platform has its own directory tree in the *drivers* hierarchy. This has tended to happen when kernel development for that platform has proceeded alongside the main source tree without being merged for a while. In these cases, keeping the directory tree separate helped in maintaining the code. Examples include *drivers/acorn* (old ARM-based computers), *drivers/macintosh*, *drivers/sgi* (Silicon Graphics workstations), and *drivers/s390* (IBM mainframes). There is little of value, usually, in looking at that code, unless you are interested in the specific platform.

Other Subdirectories

There are other subdirectories in *drivers*, but they are, in our opinion, currently of minor interest and very specific use. *drivers/mtd* implements a Memory Technology Device layer, which is used to manage solid-state disks (flash memories and other kinds of EEPROM). *drivers/i2c* offers an implementation of the i2c protocol,

which is the "Inter Integrated Circuit" two-wire bus used internally by several modern peripherals, especially frame grabbers. *drivers/i2o*, similarly, handles I2O devices (a proprietary high-speed communication standard for certain PCI devices, which has been unveiled under pressure from the free software community). *drivers/pnp* is a collection of common ISA Plug-and-Play code from various drivers, but fortunately the PnP hack is not really used nowadays by manufacturers.

Under *drivers/* you also find initial support for new device classes that are currently implemented by a very small range of devices.

That's the case for fiber channel support (*drivers/fc4*) and *drivers/telephony*. There's even an empty directory *drivers/misc*, which claims to be "for misc devices that really don't fit anywhere else." The directory is empty of code, but hosts an (empty) makefile with the comment just quoted.

The Linux kernel is so huge that it's impossible to cover it all in a few pages. Moreover, it is a moving target, and once you think you are finished, you find that the new patch released by your preferred hackers includes a whole lot of new material. It may well be that the *misc* directory in 2.4 is not empty anymore as you read this.

Although we consider it unlikely, it may even happen that 2.6 or 3.0 will turn out to be pretty different from 2.4; unfortunately, this edition of the book won't automatically update itself to cover the new releases and will become obsolete over time. Despite our best efforts to cover the current version of the kernel, both in this chapter and in the whole book, there's no substitute for direct reference to the source code.