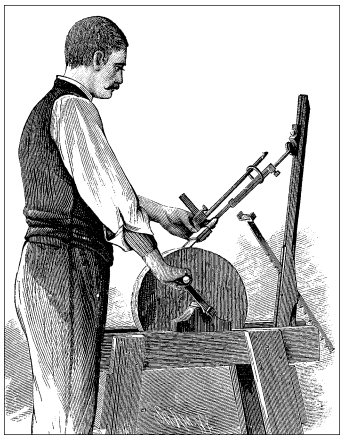
CHAPTER TWO

BUILDING AND RUNNING MODULES



It's high time now to begin programming. This chapter introduces all the essential concepts about modules and kernel programming. In these few pages, we build and run a complete module. Developing such expertise is an essential foundation for any kind of modularized driver. To avoid throwing in too many concepts at once, this chapter talks only about modules, without referring to any specific device class.

All the kernel items (functions, variables, header files, and macros) that are introduced here are described in a reference section at the end of the chapter.

For the impatient reader, the following code is a complete "Hello, World" module (which does nothing in particular). This code will compile and run under Linux kernel versions 2.0 through 2.4.*

```
#define MODULE
#include <linux/module.h>

int init_module(void)  { printk("<1>Hello, world\n"); return 0; }
void cleanup_module(void) { printk("<1>Goodbye cruel world\n"); }
```

The *printk* function is defined in the Linux kernel and behaves similarly to the standard C library function *printf*. The kernel needs its own printing function because it runs by itself, without the help of the C library. The module can call *printk* because, after *insmod* has loaded it, the module is linked to the kernel and can access the kernel's public symbols (functions and variables, as detailed in the next section). The string `<1>` is the priority of the message. We've specified a high priority (low cardinal number) in this module because a message with the default priority might not show on the console, depending on the kernel version you are

* This example, and all the others presented in this book, is available on the O'Reilly FTP site, as explained in Chapter 1.

running, the version of the *klogd* daemon, and your configuration. You can ignore this issue for now; we'll explain it in the section "printk" in Chapter 4.

You can test the module by calling *insmod* and *rmmod*, as shown in the screen dump in the following paragraph. Note that only the superuser can load and unload a module.

The source file shown earlier can be loaded and unloaded as shown only if the running kernel has module version support disabled; however, most distributions preinstall versioned kernels (versioning is discussed in "Version Control in Modules" in Chapter 11). Although older *modutils* allowed loading nonversioned modules to versioned kernels, this is no longer possible. To solve the problem with *hello.c*, the source in the *misc-modules* directory of the sample code includes a few more lines to be able to run both under versioned and nonversioned kernels. However, we strongly suggest you compile and run your own kernel (without version support) before you run the sample code.*

```
root# gcc -c hello.c
root# insmod ./hello.o
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

According to the mechanism your system uses to deliver the message lines, your output may be different. In particular, the previous screen dump was taken from a text console; if you are running *insmod* and *rmmod* from an *xterm*, you won't see anything on your TTY. Instead, it may go to one of the system log files, such as */var/log/messages* (the name of the actual file varies between Linux distributions). The mechanism used to deliver kernel messages is described in "How Messages Get Logged" in Chapter 4.

As you can see, writing a module is not as difficult as you might expect. The hard part is understanding your device and how to maximize performance. We'll go deeper into modularization throughout this chapter and leave device-specific issues to later chapters.

Kernel Modules Versus Applications

Before we go further, it's worth underlining the various differences between a kernel module and an application.

Whereas an application performs a single task from beginning to end, a module registers itself in order to serve future requests, and its "main" function terminates immediately. In other words, the task of the function *init_module* (the module's

* If you are new to building kernels, Alessandro has posted an article at *http://www.linux.it/kerneldocs/kconf* that should help you get started.

entry point) is to prepare for later invocation of the module's functions; it's as though the module were saying, "Here I am, and this is what I can do." The second entry point of a module, *cleanup_module*, gets invoked just before the module is unloaded. It should tell the kernel, "I'm not there anymore; don't ask me to do anything else." The ability to unload a module is one of the features of modularization that you'll most appreciate, because it helps cut down development time; you can test successive versions of your new driver without going through the lengthy shutdown/reboot cycle each time.

As a programmer, you know that an application can call functions it doesn't define: the linking stage resolves external references using the appropriate library of functions. *printf* is one of those callable functions and is defined in *libc*. A module, on the other hand, is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; there are no libraries to link to. The *printk* function used in *hello.c* earlier, for example, is the version of *printf* defined within the kernel and exported to modules. It behaves similarly to the original function, with a few minor differences, the main one being lack of floating-point support.*

Figure 2-1 shows how function calls and function pointers are used in a module to add new functionality to a running kernel.

Because no library is linked to modules, source files should *never* include the usual header files. Only functions that are actually part of the kernel itself may be used in kernel modules. Anything related to the kernel is declared in headers found in *include/linux* and *include/asm* inside the kernel sources (usually found in */usr/src/linux*). Older distributions (based on *libc* version 5 or earlier) used to carry symbolic links from */usr/include/linux* and */usr/include/asm* to the actual kernel sources, so your *libc* include tree could refer to the headers of the actual kernel source you had installed. These symbolic links made it convenient for user-space applications to include kernel header files, which they occasionally need to do.

Even though user-space headers are now separate from kernel-space headers, sometimes applications still include kernel headers, either before an old library is used or before new information is needed that is not available in the user-space headers. However, many of the declarations in the kernel header files are relevant only to the kernel itself and should not be seen by user-space applications. These declarations are therefore protected by `#ifdef __KERNEL__` blocks. That's why your driver, like other kernel code, will need to be compiled with the `__KERNEL__` preprocessor symbol defined.

The role of individual kernel headers will be introduced throughout the book as each of them is needed.

---

* The implementation found in Linux 2.0 and 2.2 has no support for the `L` and `Z` qualifiers. They have been introduced in 2.4, though.

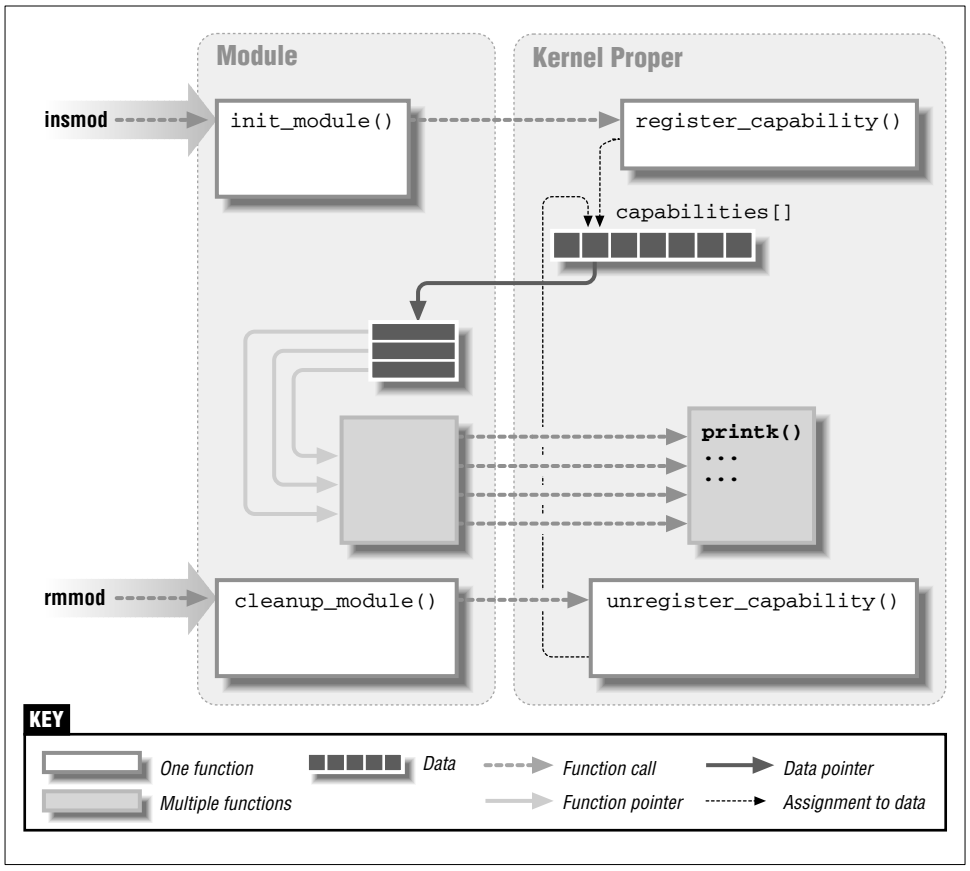


Figure 2-1. Linking a module to the kernel

Developers working on any large software system (such as the kernel) must be aware of and avoid *namespace pollution*. Namespace pollution is what happens when there are many functions and global variables whose names aren't meaningful enough to be easily distinguished. The programmer who is forced to deal with such an application expends much mental energy just to remember the "reserved" names and to find unique names for new symbols. Namespace collisions can create problems ranging from module loading failures to bizarre failures—which, perhaps, only happen to a remote user of your code who builds a kernel with a different set of configuration options.

Developers can't afford to fall into such an error when writing kernel code because even the smallest module will be linked to the whole kernel. The best approach for preventing namespace pollution is to declare all your symbols as `static` and to use a prefix that is unique within the kernel for the symbols you

leave global. Also note that you, as a module writer, can control the external visibility of your symbols, as described in "The Kernel Symbol Table" later in this chapter.*

Using the chosen prefix for private symbols within the module may be a good practice as well, as it may simplify debugging. While testing your driver, you could export all the symbols without polluting your namespace. Prefixes used in the kernel are, by convention, all lowercase, and we'll stick to the same convention.

The last difference between kernel programming and application programming is in how each environment handles faults: whereas a segmentation fault is harmless during application development and a debugger can always be used to trace the error to the problem in the source code, a kernel fault is fatal at least for the current process, if not for the whole system. We'll see how to trace kernel errors in Chapter 4, in the section "Debugging System Faults."

User Space and Kernel Space

A module runs in the so-called *kernel space*, whereas applications run in *user space*. This concept is at the base of operating systems theory.

The role of the operating system, in practice, is to provide programs with a consistent view of the computer's hardware. In addition, the operating system must account for independent operation of programs and protection against unauthorized access to resources. This nontrivial task is only possible if the CPU enforces protection of system software from the applications.

Every modern processor is able to enforce this behavior. The chosen approach is to implement different operating modalities (or levels) in the CPU itself. The levels have different roles, and some operations are disallowed at the lower levels; program code can switch from one level to another only through a limited number of gates. Unix systems are designed to take advantage of this hardware feature, using two such levels. All current processors have at least two protection levels, and some, like the x86 family, have more levels; when several levels exist, the highest and lowest levels are used. Under Unix, the kernel executes in the highest level (also called *supervisor mode*), where everything is allowed, whereas applications execute in the lowest level (the so-called *user mode*), where the processor regulates direct access to hardware and unauthorized access to memory.

We usually refer to the execution modes as *kernel space* and *user space*. These terms encompass not only the different privilege levels inherent in the two modes, but also the fact that each mode has its own memory mapping—its own address space—as well.

---

* Most versions of *insmod* (but not all of them) export all non-`static` symbols if they find no specific instruction in the module; that's why it's wise to declare as `static` all the symbols you are not willing to export.

Unix transfers execution from user space to kernel space whenever an application issues a system call or is suspended by a hardware interrupt. Kernel code executing a system call is working in the context of a process—it operates on behalf of the calling process and is able to access data in the process's address space. Code that handles interrupts, on the other hand, is asynchronous with respect to processes and is not related to any particular process.

The role of a module is to extend kernel functionality; modularized code runs in kernel space. Usually a driver performs both the tasks outlined previously: some functions in the module are executed as part of system calls, and some are in charge of interrupt handling.

Concurrency in the Kernel

One way in which device driver programming differs greatly from (most) application programming is the issue of concurrency. An application typically runs sequentially, from the beginning to the end, without any need to worry about what else might be happening to change its environment. Kernel code does not run in such a simple world and must be written with the idea that many things can be happening at once.

There are a few sources of concurrency in kernel programming. Naturally, Linux systems run multiple processes, more than one of which can be trying to use your driver at the same time. Most devices are capable of interrupting the processor; interrupt handlers run asynchronously and can be invoked at the same time that your driver is trying to do something else. Several software abstractions (such as kernel timers, introduced in Chapter 6) run asynchronously as well. Moreover, of course, Linux can run on symmetric multiprocessor (SMP) systems, with the result that your driver could be executing concurrently on more than one CPU.

As a result, Linux kernel code, including driver code, must be *reentrant*—it must be capable of running in more than one context at the same time. Data structures must be carefully designed to keep multiple threads of execution separate, and the code must take care to access shared data in ways that prevent corruption of the data. Writing code that handles concurrency and avoids race conditions (situations in which an unfortunate order of execution causes undesirable behavior) requires thought and can be tricky. Every sample driver in this book has been written with concurrency in mind, and we will explain the techniques we use as we come to them.

A common mistake made by driver programmers is to assume that concurrency is not a problem as long as a particular segment of code does not go to sleep (or "block"). It is true that the Linux kernel is nonpreemptive; with the important exception of servicing interrupts, it will not take the processor away from kernel

code that does not yield willingly. In past times, this nonpreemptive behavior was enough to prevent unwanted concurrency most of the time. On SMP systems, however, preemption is not required to cause concurrent execution.

If your code assumes that it will not be preempted, it will not run properly on SMP systems. Even if you do not have such a system, others who run your code may have one. In the future, it is also possible that the kernel will move to a preemptive mode of operation, at which point even uniprocessor systems will have to deal with concurrency everywhere (some variants of the kernel already implement it). Thus, a prudent programmer will always program as if he or she were working on an SMP system.

The Current Process

Although kernel modules don't execute sequentially as applications do, most actions performed by the kernel are related to a specific process. Kernel code can know the current process driving it by accessing the global item `current`, a pointer to `struct task_struct`, which as of version 2.4 of the kernel is declared in `<asm/current.h>`, included by `<linux/sched.h>`. The `current` pointer refers to the user process currently executing. During the execution of a system call, such as *open* or *read*, the current process is the one that invoked the call. Kernel code can use process-specific information by using `current`, if it needs to do so. An example of this technique is presented in "Access Control on a Device File," in Chapter 5.

Actually, `current` is not properly a global variable any more, like it was in the first Linux kernels. The developers optimized access to the structure describing the current process by hiding it in the stack page. You can look at the details of `current` in `<asm/current.h>`. While the code you'll look at might seem hairy, we must keep in mind that Linux is an SMP-compliant system, and a global variable simply won't work when you are dealing with multiple CPUs. The details of the implementation remain hidden to other kernel subsystems though, and a device driver can just include `<linux/sched.h>` and refer to the `current` process.

From a module's point of view, `current` is just like the external reference *printk*. A module can refer to `current` wherever it sees fit. For example, the following statement prints the process ID and the command name of the current process by accessing certain fields in `struct task_struct`:

```
printk("The process is \"%s\" (pid %i)\n",
 current->comm, current->pid);
```

The command name stored in `current->comm` is the base name of the program file that is being executed by the current process.

Compiling and Loading

The rest of this chapter is devoted to writing a complete, though typeless, module. That is, the module will not belong to any of the classes listed in "Classes of Devices and Modules" in Chapter 1. The sample driver shown in this chapter is called *skull*, short for Simple Kernel Utility for Loading Localities. You can reuse the *skull* source to load your own local code to the kernel, after removing the sample functionality it offers.*

Before we deal with the roles of *init_module* and *cleanup_module*, however, we'll write a makefile that builds object code that the kernel can load.

First, we need to define the `__KERNEL__` symbol in the preprocessor before we include any headers. As mentioned earlier, much of the kernel-specific content in the kernel headers is unavailable without this symbol.

Another important symbol is `MODULE`, which must be defined before including `<linux/module.h>` (except for drivers that are linked directly into the kernel). This book does not cover directly linked modules; thus, the `MODULE` symbol is always defined in our examples.

If you are compiling for an SMP machine, you also need to define `__SMP__` before including the kernel headers. In version 2.2, the "multiprocessor or uniprocessor" choice was promoted to a proper configuration item, so using these lines as the very first lines of your modules will do the task:

```
#include <linux/config.h>
#ifdef CONFIG_SMP
# define __SMP__
#endif
```

A module writer must also specify the *–O* flag to the compiler, because many functions are declared as `inline` in the header files. *gcc* doesn't expand inline functions unless optimization is enabled, but it can accept both the *–g* and *–O* options, allowing you to debug code that uses inline functions.† Because the kernel makes extensive use of inline functions, it is important that they be expanded properly.

You may also need to check that the compiler you are running matches the kernel you are compiling against, referring to the file *Documentation/Changes* in the kernel source tree. The kernel and the compiler are developed at the same time, though by different groups, so sometimes changes in one tool reveal bugs in the

* We use the word *local* here to denote personal changes to the system, in the good old Unix tradition of */usr/local*.

† Note, however, that using any optimization greater than *–O2* is risky, because the compiler might inline functions that are not declared as `inline` in the source. This may be a problem with kernel code, because some functions expect to find a standard stack layout when they are called.

other. Some distributions ship a version of the compiler that is too new to reliably build the kernel. In this case, they will usually provide a separate package (often called *kgcc*) with a compiler intended for kernel compilation.

Finally, in order to prevent unpleasant errors, we suggest that you use the *–Wall* (all warnings) compiler flag, and also that you fix all features in your code that cause compiler warnings, even if this requires changing your usual programming style. When writing kernel code, the preferred coding style is undoubtedly Linus's own style. *Documentation/CodingStyle* is amusing reading and a mandatory lesson for anyone interested in kernel hacking.

All the definitions and flags we have introduced so far are best located within the `CFLAGS` variable used by *make*.

In addition to a suitable `CFLAGS`, the makefile being built needs a rule for joining different object files. The rule is needed only if the module is split into different source files, but that is not uncommon with modules. The object files are joined by the *ld -r* command, which is not really a linking operation, even though it uses the linker. The output of *ld -r* is another object file, which incorporates all the code from the input files. The *–r* option means "relocatable;" the output file is relocatable in that it doesn't yet embed absolute addresses.

The following makefile is a minimal example showing how to build a module made up of two source files. If your module is made up of a single source file, just skip the entry containing *ld -r*.

```
# Change it here or specify it on the "make" command line
KERNELDIR = /usr/src/linux

include $(KERNELDIR)/.config

CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include \
  -O -Wall

ifdef CONFIG_SMP
 CFLAGS += -D__SMP__ -DSMP
endif

all: skull.o

skull.o: skull_init.o skull_clean.o
  $(LD) -r $^ -o $@

clean:
  rm -f *.o *~ core
```

If you are not familiar with *make*, you may wonder why no *.c* file and no compilation rule appear in the makefile shown. These declarations are unnecessary because *make* is smart enough to turn *.c* into *.o* without being instructed to, using the current (or default) choice for the compiler, `$(CC)`, and its flags, `$(CFLAGS)`.

After the module is built, the next step is loading it into the kernel. As we've already suggested, *insmod* does the job for you. The program is like *ld*, in that it links any unresolved symbol in the module to the symbol table of the running kernel. Unlike the linker, however, it doesn't modify the disk file, but rather an in-memory copy. *insmod* accepts a number of command-line options (for details, see the manpage), and it can assign values to integer and string variables in your module before linking it to the current kernel. Thus, if a module is correctly designed, it can be configured at load time; load-time configuration gives the user more flexibility than compile-time configuration, which is still used sometimes. Load-time configuration is explained in "Automatic and Manual Configuration" later in this chapter.

Interested readers may want to look at how the kernel supports *insmod*: it relies on a few system calls defined in *kernel/module.c*. The function *sys_create_module* allocates kernel memory to hold a module (this memory is allocated with *vmalloc*; see "vmalloc and Friends" in Chapter 7). The system call *get_kernel_syms* returns the kernel symbol table so that kernel references in the module can be resolved, and *sys_init_module* copies the relocated object code to kernel space and calls the module's initialization function.

If you actually look in the kernel source, you'll find that the names of the system calls are prefixed with `sys_`. This is true for all system calls and no other functions; it's useful to keep this in mind when grepping for the system calls in the sources.

Version Dependency

Bear in mind that your module's code has to be recompiled for each version of the kernel that it will be linked to. Each module defines a symbol called `__module_kernel_version`, which *insmod* matches against the version number of the current kernel. This symbol is placed in the `.modinfo` Executable Linking and Format (ELF) section, as explained in detail in Chapter 11. Please note that this description of the internals applies only to versions 2.2 and 2.4 of the kernel; Linux 2.0 did the same job in a different way.

The compiler will define the symbol for you whenever you include `<linux/module.h>` (that's why *hello.c* earlier didn't need to declare it). This also means that if your module is made up of multiple source files, you have to include `<linux/module.h>` from only one of your source files (unless you use `__NO_VERSION__`, which we'll introduce in a while).

In case of version mismatch, you can still try to load a module against a different kernel version by specifying the *–f* ("force") switch to *insmod*, but this operation isn't safe and can fail. It's also difficult to tell in advance what will happen. Loading can fail because of mismatching symbols, in which case you'll get an error

message, or it can fail because of an internal change in the kernel. If that happens, you'll get serious errors at runtime and possibly a system panic—a good reason to be wary of version mismatches. Version mismatches can be handled more gracefully by using versioning in the kernel (a topic that is more advanced and is introduced in "Version Control in Modules" in Chapter 11).

If you want to compile your module for a particular kernel version, you have to include the specific header files for that kernel (for example, by declaring a different `KERNELDIR`) in the makefile given previously. This situation is not uncommon when playing with the kernel sources, as most of the time you'll end up with several versions of the source tree. All of the sample modules accompanying this book use the `KERNELDIR` variable to point to the correct kernel sources; it can be set in your environment or passed on the command line of *make*.

When asked to load a module, *insmod* follows its own search path to look for the object file, looking in version-dependent directories under */lib/modules*. Although older versions of the program looked in the current directory, first, that behavior is now disabled for security reasons (it's the same problem of the `PATH` environment variable). Thus, if you need to load a module from the current directory you should use *./module.o*, which works with all known versions of the tool.

Sometimes, you'll encounter kernel interfaces that behave differently between versions 2.0.*x* and 2.4.*x* of Linux. In this case you'll need to resort to the macros defining the version number of the current source tree, which are defined in the header `<linux/version.h>`. We will point out cases where interfaces have changed as we come to them, either within the chapter or in a specific section about version dependencies at the end, to avoid complicating a 2.4-specific discussion.

The header, automatically included by *linux/module.h*, defines the following macros:

`UTS_RELEASE`
: The macro expands to a string describing the version of this kernel tree. For example, `"2.3.48"`.

`LINUX_VERSION_CODE`
: The macro expands to the binary representation of the kernel version, one byte for each part of the version release number. For example, the code for 2.3.48 is 131888 (i.e., 0x020330).* With this information, you can (almost) easily determine what version of the kernel you are dealing with.

`KERNEL_VERSION(major,minor,release)`
: This is the macro used to build a "kernel_version_code" from the individual numbers that build up a version number. For example, `KERNEL_VERSION(2,3,48)` expands to 131888. This macro is very useful when you

* This allows up to 256 development versions between stable versions.

need to compare the current version and a known checkpoint. We'll use this macro several times throughout the book.

The file *version.h* is included by *module.h*, so you won't usually need to include *version.h* explicitly. On the other hand, you can prevent *module.h* from including *version.h* by declaring `__NO_VERSION__` in advance. You'll use `__NO_VERSION__` if you need to include `<linux/module.h>` in several source files that will be linked together to form a single module—for example, if you need preprocessor macros declared in *module.h*. Declaring `__NO_VERSION__` before including *module.h* prevents automatic declaration of the string `__module_kernel_version` or its equivalent in source files where you don't want it (*ld -r* would complain about the multiple definition of the symbol). Sample modules in this book use `__NO_VERSION__` to this end.

Most dependencies based on the kernel version can be worked around with preprocessor conditionals by exploiting `KERNEL_VERSION` and `LINUX_VERSION_CODE`. Version dependency should, however, not clutter driver code with hairy `#ifdef` conditionals; the best way to deal with incompatibilities is by confining them to a specific header file. That's why our sample code includes a *sysdep.h* header, used to hide all incompatibilities in suitable macro definitions.

The first version dependency we are going to face is in the definition of a "`make install`" rule for our drivers. As you may expect, the installation directory, which varies according to the kernel version being used, is chosen by looking in *version.h*. The following fragment comes from the file *Rules.make*, which is included by all makefiles:

```
VERSIONFILE = $(INCLUDEDIR)/linux/version.h
VERSION  = $(shell awk -F\" '/REL/ {print $$2}' $(VERSIONFILE))
INSTALLDIR = /lib/modules/$(VERSION)/misc
```

We chose to install all of our drivers in the *misc* directory; this is both the right choice for miscellaneous add-ons and a good way to avoid dealing with the change in the directory structure under */lib/modules* that was introduced right before version 2.4 of the kernel was released. Even though the new directory structure is more complicated, the *misc* directory is used by both old and new versions of the *modutils* package.

With the definition of `INSTALLDIR` just given, the install rule of each makefile, then, is laid out like this:

```
install:
        install -d $(INSTALLDIR)
        install -c $(OBJS) $(INSTALLDIR)
```

Platform Dependency

Each computer platform has its peculiarities, and kernel designers are free to exploit all the peculiarities to achieve better performance in the target object file.

Unlike application developers, who must link their code with precompiled libraries and stick to conventions on parameter passing, kernel developers can dedicate some processor registers to specific roles, and they have done so. Moreover, kernel code can be optimized for a specific processor in a CPU family to get the best from the target platform: unlike applications that are often distributed in binary format, a custom compilation of the kernel can be optimized for a specific computer set.

Modularized code, in order to be interoperable with the kernel, needs to be compiled using the same options used in compiling the kernel (i.e., reserving the same registers for special use and performing the same optimizations). For this reason, our top-level *Rules.make* includes a platform-specific file that complements the makefiles with extra definitions. All of those files are called `Makefile.`*`platform`* and assign suitable values to *make* variables according to the current kernel configuration.

Another interesting feature of this layout of makefiles is that cross compilation is supported for the whole tree of sample files. Whenever you need to cross compile for your target platform, you'll need to replace all of your tools (*gcc*, *ld*, etc.) with another set of tools (for example, *m68k-linux-gcc*, *m68k-linux-ld*). The prefix to be used is defined as `$(CROSS_COMPILE)`, either in the *make* command line or in your environment.

The SPARC architecture is a special case that must be handled by the makefiles. User-space programs running on the SPARC64 (SPARC V9) platform are the same binaries you run on SPARC32 (SPARC V8). Therefore, the default compiler running on SPARC64 (*gcc*) generates SPARC32 object code. The kernel, on the other hand, must run SPARC V9 object code, so a cross compiler is needed. All GNU/Linux distributions for SPARC64 include a suitable cross compiler, which the makefiles select.

Although the complete list of version and platform dependencies is slightly more complicated than shown here, the previous description and the set of makefiles we provide is enough to get things going. The set of makefiles and the kernel sources can be browsed if you are looking for more detailed information.

The Kernel Symbol Table

We've seen how *insmod* resolves undefined symbols against the table of public kernel symbols. The table contains the addresses of global kernel items—

functions and variables—that are needed to implement modularized drivers. The public symbol table can be read in text form from the file */proc/ksyms* (assuming, of course, that your kernel has support for the */proc* filesystem—which it really should).

When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table, and you can see it appear in */proc/ksyms* or in the output of the *ksyms* command.

New modules can use symbols exported by your module, and you can stack new modules on top of other modules. Module stacking is implemented in the mainstream kernel sources as well: the *msdos* filesystem relies on symbols exported by the *fat* module, and each input USB device module stacks on the *usbcore* and *input* modules.

Module stacking is useful in complex projects. If a new abstraction is implemented in the form of a device driver, it might offer a plug for hardware-specific implementations. For example, the video-for-linux set of drivers is split into a generic module that exports symbols used by lower-level device drivers for specific hardware. According to your setup, you load the generic video module and the specific module for your installed hardware. Support for parallel ports and the wide variety of attachable devices is handled in the same way, as is the USB kernel subsystem. Stacking in the parallel port subsystem is shown in Figure 2-2; the arrows show the communications between the modules (with some example functions and data structures) and with the kernel programming interface.

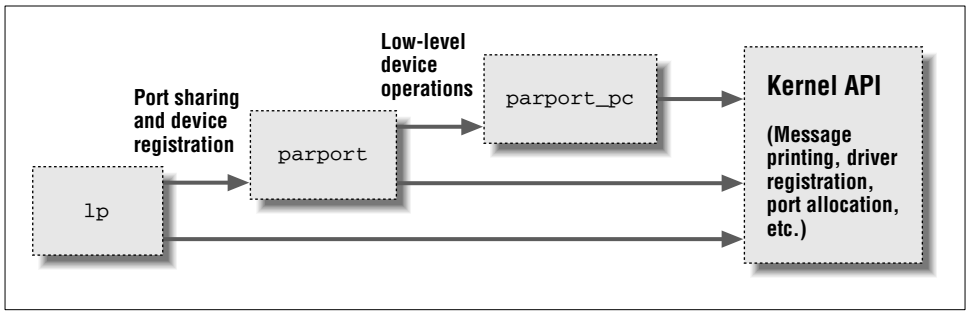


Figure 2-2. Stacking of parallel port driver modules

When using stacked modules, it is helpful to be aware of the *modprobe* utility. *modprobe* functions in much the same way as *insmod*, but it also loads any other modules that are required by the module you want to load. Thus, one *modprobe* command can sometimes replace several invocations of *insmod* (although you'll still need *insmod* when loading your own modules from the current directory, because *modprobe* only looks in the tree of installed modules).

Layered modularization can help reduce development time by simplifying each layer. This is similar to the separation between mechanism and policy that we discussed in Chapter 1.

In the usual case, a module implements its own functionality without the need to export any symbols at all. You will need to export symbols, however, whenever other modules may benefit from using them. You may also need to include specific instructions to avoid exporting all non-`static` symbols, as most versions (but not all) of *modutils* export all of them by default.

The Linux kernel header files provide a convenient way to manage the visibility of your symbols, thus reducing namespace pollution and promoting proper information hiding. The mechanism described in this section works with kernels 2.1.18 and later; the 2.0 kernel had a completely different mechanism, which is described at the end of the chapter.

If your module exports no symbols at all, you might want to make that explicit by placing a line with this macro call in your source file:

```
EXPORT_NO_SYMBOLS;
```

The macro expands to an assembler directive and may appear anywhere within the module. Portable code, however, should place it within the module initialization function (*init_module*), because the version of this macro defined in *sysdep.h* for older kernels will work only there.

If, on the other hand, you need to export a subset of symbols from your module, the first step is defining the preprocessor macro `EXPORT_SYMTAB`. This macro must be defined *before* including *module.h*. It is common to define it at compile time with the *–D* compiler flag in *Makefile*.

If `EXPORT_SYMTAB` is defined, individual symbols are exported with a couple of macros:

```
EXPORT_SYMBOL (name);
EXPORT_SYMBOL_NOVERS (name);
```

Either version of the macro will make the given symbol available outside the module; the second version (`EXPORT_SYMBOL_NOVERS`) exports the symbol with no versioning information (described in Chapter 11). Symbols must be exported outside of any function because the macros expand to the declaration of a variable. (Interested readers can look at `<linux/module.h>` for the details, even though the details are not needed to make things work.)

Initialization and Shutdown

As already mentioned, *init_module* registers any facility offered by the module. By *facility*, we mean a new functionality, be it a whole driver or a new software abstraction, that can be accessed by an application.

Modules can register many different types of facilities; for each facility, there is a specific kernel function that accomplishes this registration. The arguments passed to the kernel registration functions are usually a pointer to a data structure describing the new facility and the name of the facility being registered. The data structure usually embeds pointers to module functions, which is how functions in the module body get called.

The items that can be registered exceed the list of device types mentioned in Chapter 1. They include serial ports, miscellaneous devices, */proc* files, executable domains, and line disciplines. Many of those registrable items support functions that aren't directly related to hardware but remain in the "software abstractions" field. Those items can be registered because they are integrated into the driver's functionality anyway (like */proc* files and line disciplines for example).

There are other facilities that can be registered as add-ons for certain drivers, but their use is so specific that it's not worth talking about them; they use the stacking technique, as described earlier in "The Kernel Symbol Table." If you want to probe further, you can grep for EXPORT_SYMBOL in the kernel sources and find the entry points offered by different drivers. Most registration functions are prefixed with `register_`, so another possible way to find them is to grep for `register_` in */proc/ksyms*.

Error Handling in init_module

If any errors occur when you register utilities, you must undo any registration activities performed before the failure. An error can happen, for example, if there isn't enough memory in the system to allocate a new data structure or because a resource being requested is already being used by other drivers. Though unlikely, it might happen, and good program code must be prepared to handle this event.

Linux doesn't keep a per-module registry of facilities that have been registered, so the module must back out of everything itself if *init_module* fails at some point. If you ever fail to unregister what you obtained, the kernel is left in an unstable state: you can't register your facilities again by reloading the module because they will appear to be busy, and you can't unregister them because you'd need the same pointer you used to register and you're not likely to be able to figure out the address. Recovery from such situations is tricky, and you'll be often forced to reboot in order to be able to load a newer revision of your module.

Error recovery is sometimes best handled with the `goto` statement. We normally hate to use `goto`, but in our opinion this is one situation (well, the *only* situation) where it is useful. In the kernel, `goto` is often used as shown here to deal with errors.

The following sample code (using fictitious registration and unregistration functions) behaves correctly if initialization fails at any point.

```
int init_module(void)
{
int err;

 /* registration takes a pointer and a name */
 err = register_this(ptr1, "skull");
 if (err) goto fail_this;
 err = register_that(ptr2, "skull");
 if (err) goto fail_that;
 err = register_those(ptr3, "skull");
 if (err) goto fail_those;

 return 0; /* success */

 fail_those: unregister_that(ptr2, "skull");
 fail_that: unregister_this(ptr1, "skull");
 fail_this: return err; /* propagate the error */
}
```

This code attempts to register three (fictitious) facilities. The `goto` statement is used in case of failure to cause the unregistration of only the facilities that had been successfully registered before things went bad.

Another option, requiring no hairy `goto` statements, is keeping track of what has been successfully registered and calling *cleanup_module* in case of any error. The cleanup function will only unroll the steps that have been successfully accomplished. This alternative, however, requires more code and more CPU time, so in fast paths you'll still resort to `goto` as the best error-recovery tool. The return value of *init_module*, `err`, is an error code. In the Linux kernel, error codes are negative numbers belonging to the set defined in `<linux/errno.h>`. If you want to generate your own error codes instead of returning what you get from other functions, you should include `<linux/errno.h>` in order to use symbolic values such as `-ENODEV`, `-ENOMEM`, and so on. It is always good practice to return appropriate error codes, because user programs can turn them to meaningful strings using *perror* or similar means. (However, it's interesting to note that several versions of *modutils* returned a "Device busy" message for any error returned by *init_module*; the problem has only been fixed in recent releases.)

Obviously, *cleanup_module* must undo any registration performed by *init_module*, and it is customary (but not mandatory) to unregister facilities in the reverse order used to register them:

```
void cleanup_module(void)
{
 unregister_those(ptr3, "skull");
 unregister_that(ptr2, "skull");
 unregister_this(ptr1, "skull");
 return;
}
```

If your initialization and cleanup are more complex than dealing with a few items, the `goto` approach may become difficult to manage, because all the cleanup code must be repeated within *init_module*, with several labels intermixed. Sometimes, therefore, a different layout of the code proves more successful.

What you'd do to minimize code duplication and keep everything streamlined is to call *cleanup_module* from within *init_module* whenever an error occurs. The cleanup function, then, must check the status of each item before undoing its registration. In its simplest form, the code looks like the following:

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;

void cleanup_module(void)
{
 if (item1)
        release_thing(item1);
 if (item2)
  release_thing2(item2);
 if (stuff_ok)
  unregister_stuff();
 return;
}

int init_module(void)
{
 int err = -ENOMEM;

 item1 = allocate_thing(arguments);
 item2 = allocate_thing2(arguments2);
 if (!item2 || !item2)
  goto fail;
 err = register_stuff(item1, item2);
 if (!err)
  stuff_ok = 1;
 else
  goto fail;
 return 0; /* success */

 fail:
 cleanup_module();
 return err;
}
```

As shown in this code, you may or may not need external flags to mark success of the initialization step, depending on the semantics of the registration/allocation function you call. Whether or not flags are needed, this kind of initialization scales well to a large number of items and is often better than the technique shown earlier.

The Usage Count

The system keeps a usage count for every module in order to determine whether the module can be safely removed. The system needs this information because a module can't be unloaded if it is busy: you can't remove a filesystem type while the filesystem is mounted, and you can't drop a char device while a process is using it, or you'll experience some sort of segmentation fault or kernel panic when wild pointers get dereferenced.

In modern kernels, the system can automatically track the usage count for you, using a mechanism that we will see in the next chapter. There are still times, however, when you will need to adjust the usage count manually. Code that must be portable to older kernels must still use manual usage count maintenance as well. To work with the usage count, use these three macros:

`MOD_INC_USE_COUNT`
: Increments the count for the current module

`MOD_DEC_USE_COUNT`
: Decrements the count

`MOD_IN_USE`
: Evaluates to true if the count is not zero

The macros are defined in `<linux/module.h>`, and they act on internal data structures that shouldn't be accessed directly by the programmer. The internals of module management changed a lot during 2.1 development and were completely rewritten in 2.1.18, but the use of these macros did not change.

Note that there's no need to check for `MOD_IN_USE` from within *cleanup_module*, because the check is performed by the system call *sys_delete_module* (defined in *kernel/module.c*) in advance.

Proper management of the module usage count is critical for system stability. Remember that the kernel can decide to try to unload your module at absolutely any time. A common module programming error is to start a series of operations (in response, say, to an *open* request) and increment the usage count at the end. If the kernel unloads the module halfway through those operations, chaos is ensured. To avoid this kind of problem, you should call `MOD_INC_USE_COUNT` *before* doing almost anything else in a module.

You won't be able to unload a module if you lose track of the usage count. This situation may very well happen during development, so you should keep it in mind. For example, if a process gets destroyed because your driver dereferenced a NULL pointer, the driver won't be able to close the device, and the usage count won't fall back to zero. One possible solution is to completely disable the usage count during the debugging cycle by redefining both `MOD_INC_USE_COUNT` and

`MOD_DEC_USE_COUNT` to no-ops. Another solution is to use some other method to force the counter to zero (you'll see this done in the section "Using the ioctl Argument" in Chapter 5). Sanity checks should never be circumvented in a production module. For debugging, however, sometimes a brute-force attitude helps save development time and is therefore acceptable.

The current value of the usage count is found in the third field of each entry in */proc/modules*. This file shows the modules currently loaded in the system, with one entry for each module. The fields are the name of the module, the number of bytes of memory it uses, and the current usage count. This is a typical */proc/modules* file:

```
parport_pc    7604 1 (autoclean)
lp      4800 0 (unused)
parport     8084 1 [parport_probe parport_pc lp]
lockd     33256 1 (autoclean)
sunrpc     56612 1 (autoclean) [lockd]
ds      6252 1
i82365     22304 1
pcmcia_core   41280 0 [ds i82365]
```

Here we see several modules in the system. Among other things, the parallel port modules have been loaded in a stacked manner, as we saw in Figure 2-2. The `(autoclean)` marker identifies modules managed by *kmod* or *kerneld* (see Chapter 11); the `(unused)` marker means exactly that. Other flags exist as well. In Linux 2.0, the second (size) field was expressed in pages (4 KB each on most platforms) rather than bytes.

Unloading

To unload a module, use the *rmmod* command. Its task is much simpler than loading, since no linking has to be performed. The command invokes the *delete_module* system call, which calls *cleanup_module* in the module itself if the usage count is zero or returns an error otherwise.

The *cleanup_module* implementation is in charge of unregistering every item that was registered by the module. Only the exported symbols are removed automatically.

Explicit Initialization and Cleanup Functions

As we have seen, the kernel calls *init_module* to initialize a newly loaded module, and calls *cleanup_module* just before module removal. In modern kernels, however, these functions often have different names. As of kernel 2.3.13, a facility exists for explicitly naming the module initialization and cleanup routines; using this facility is the preferred programming style.

Consider an example. If your module names its initialization routine *my_init* (instead of *init_module*) and its cleanup routine *my_cleanup*, you would mark them with the following two lines (usually at the end of the source file):

```
module_init(my_init);
module_exit(my_cleanup);
```

Note that your code must include `<linux/init.h>` to use *module_init* and *module_exit*.

The advantage of doing things this way is that each initialization and cleanup function in the kernel can have a unique name, which helps with debugging. These functions also make life easier for those writing drivers that work either as a module or built directly into the kernel. However, use of *module_init* and *module_exit* is not required if your initialization and cleanup functions use the old names. In fact, for modules, the only thing they do is define *init_module* and *cleanup_module* as new names for the given functions.

If you dig through the kernel source (in versions 2.2 and later), you will likely see a slightly different form of declaration for module initialization and cleanup functions, which looks like the following:

```
static int __init my_init(void)
{
 ....
}

static void __exit my_cleanup(void)
{
 ....
}
```

The attribute `__init`, when used in this way, will cause the initialization function to be discarded, and its memory reclaimed, after initialization is complete. It only works, however, for built-in drivers; it has no effect on modules. `__exit`, instead, causes the omission of the marked function when the driver is not built as a module; again, in modules, it has no effect.

The use of `__init` (and `__initdata` for data items) can reduce the amount of memory used by the kernel. There is no harm in marking module initialization functions with `__init`, even though currently there is no benefit either. Management of initialization sections has not been implemented yet for modules, but it's a possible enhancement for the future.

Using Resources

A module can't accomplish its task without using system resources such as

memory, I/O ports, I/O memory, and interrupt lines, as well as DMA channels if you use old-fashioned DMA controllers like the Industry Standard Architecture (ISA) one.

As a programmer, you are already accustomed to managing memory allocation; writing kernel code is no different in this regard. Your program obtains a memory area using *kmalloc* and releases it using *kfree*. These functions behave like *malloc* and *free*, except that *kmalloc* takes an additional argument, the priority. Usually, a priority of `GFP_KERNEL` or `GFP_USER` will do. The `GFP` acronym stands for "get free page." (Memory allocation is covered in detail in Chapter 7.)

Beginning driver programmers may initially be surprised at the need to allocate I/O ports, I/O memory,* and interrupt lines explicitly. After all, it is possible for a kernel module to simply access these resources without telling the operating system about it. Although system memory is anonymous and may be allocated from anywhere, I/O memory, ports, and interrupts have very specific roles. For instance, a driver needs to be able to allocate the exact ports it needs, not just *some* ports. But drivers cannot just go about making use of these system resources without first ensuring that they are not already in use elsewhere.

I/O Ports and I/O Memory

The job of a typical driver is, for the most part, writing and reading I/O ports and I/O memory. Access to I/O ports and I/O memory (collectively called *I/O regions*) happens both at initialization time and during normal operations.

Unfortunately, not all bus architectures offer a clean way to identify I/O regions belonging to each device, and sometimes the driver must guess where its I/O regions live, or even probe for the devices by reading and writing to "possible" address ranges. This problem is especially true of the ISA bus, which is still in use for simple devices to plug in a personal computer and is very popular in the industrial world in its PC/104 implementation (see PC/104 and PC/104+ in Chapter 15).

Despite the features (or lack of features) of the bus being used by a hardware device, the device driver should be guaranteed exclusive access to its I/O regions in order to prevent interference from other drivers. For example, if a module probing for its hardware should happen to write to ports owned by another device, weird things would undoubtedly happen.

The developers of Linux chose to implement a request/free mechanism for I/O regions, mainly as a way to prevent collisions between different devices. The mechanism has long been in use for I/O ports and was recently generalized to manage resource allocation at large. Note that this mechanism is just a software

* The memory areas that reside on the peripheral device are commonly called *I/O memory* to differentiate them from system RAM, which is customarily called memory).

abstraction that helps system housekeeping, and may or may not be enforced by hardware features. For example, unauthorized access to I/O ports doesn't produce any error condition equivalent to "segmentation fault"—the hardware can't enforce port registration.

Information about registered resources is available in text form in the files */proc/ioports* and */proc/iomem*, although the latter was only introduced during 2.3 development. We'll discuss version 2.4 now, introducing portability issues at the end of the chapter.

Ports

A typical */proc/ioports* file on a recent PC that is running version 2.4 of the kernel will look like the following:

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : serial(set)
0300-031f : NE2000
0376-0376 : ide1
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(set)
1000-103f : Intel Corporation 82371AB PIIX4 ACPI
 1000-1003 : acpi
 1004-1005 : acpi
 1008-100b : acpi
 100c-100f : acpi
1100-110f : Intel Corporation 82371AB PIIX4 IDE
1300-131f : pcnet_cs
1400-141f : Intel Corporation 82371AB PIIX4 ACPI
1800-18ff : PCI CardBus #02
1c00-1cff : PCI CardBus #04
5800-581f : Intel Corporation 82371AB PIIX4 USB
d000-dfff : PCI Bus #01
 d000-d0ff : ATI Technologies Inc 3D Rage LT Pro AGP-133
```

Each entry in the file specifies (in hexadecimal) a range of ports locked by a driver or owned by a hardware device. In earlier versions of the kernel the file had the same format, but without the "layered" structure that is shown through indentation.

The file can be used to avoid port collisions when a new device is added to the system and an I/O range must be selected by moving jumpers: the user can check what ports are already in use and set up the new device to use an available I/O range. Although you might object that most modern hardware doesn't use jumpers any more, the issue is still relevant for custom devices and industrial components.

But what is more important than the *ioports* file itself is the data structure behind it. When the software driver for a device initializes itself, it can know what port ranges are already in use; if the driver needs to probe I/O ports to detect the new device, it will be able to avoid probing those ports that are already in use by other drivers.

ISA probing is in fact a risky task, and several drivers distributed with the official Linux kernel refuse to perform probing when loaded as modules, to avoid the risk of destroying a running system by poking around in ports where some yet-unknown hardware may live. Fortunately, modern (as well as old-but-well-thought-out) bus architectures are immune to all these problems.

The programming interface used to access the I/O registry is made up of three functions:

```
int check_region(unsigned long start, unsigned long len);
struct resource *request_region(unsigned long start,
unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
```

check_region may be called to see if a range of ports is available for allocation; it returns a negative error code (such as `-EBUSY` or `-EINVAL`) if the answer is no. *request_region* will actually allocate the port range, returning a non-`NULL` pointer value if the allocation succeeds. Drivers don't need to use or save the actual pointer returned—checking against `NULL` is all you need to do.* Code that needs to work only with 2.4 kernels need not call *check_region* at all; in fact, it's better not to, since things can change between the calls to *check_region* and *request_region*. If you want to be portable to older kernels, however, you must use *check_region* because *request_region* used to return `void` before 2.4. Your driver should call *release_region*, of course, to release the ports when it is done with them.

The three functions are actually macros, and they are declared in `<linux/ioport.h>`.

The typical sequence for registering ports is the following, as it appears in the *skull* sample driver. (The function *skull_probe_hw* is not shown here because it contains device-specific code.)

---

* The actual pointer is used only when the function is called internally by the resource management subsystem of the kernel.

```
#include <linux/ioport.h>
#include <linux/errno.h>
static int skull_detect(unsigned int port, unsigned int range)
{
 int err;

 if ((err = check_region(port,range)) < 0) return err; /* busy */
 if (skull_probe_hw(port,range) != 0) return -ENODEV; /* not found */
 request_region(port,range,"skull");     /* "Can't fail" */
 return 0;
}
```

This code first looks to see if the required range of ports is available; if the ports cannot be allocated, there is no point in looking for the hardware. The actual allocation of the ports is deferred until after the device is known to exist. The *request_region* call should never fail; the kernel only loads a single module at a time, so there should not be a problem with other modules slipping in and stealing the ports during the detection phase. Paranoid code can check, but bear in mind that kernels prior to 2.4 define *request_region* as returning `void`.

Any I/O ports allocated by the driver must eventually be released; *skull* does it from within *cleanup_module*:

```
static void skull_release(unsigned int port, unsigned int range)
{
 release_region(port,range);
}
```

The request/free approach to resources is similar to the register/unregister sequence described earlier for facilities and fits well in the `goto`-based implementation scheme already outlined.

Memory

Similar to what happens for I/O ports, I/O memory information is available in the */proc/iomem* file. This is a fraction of the file as it appears on a personal computer:

```
00000000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000fffff : System ROM
00100000-03feffff : System RAM
 00100000-0022c557 : Kernel code
 0022c558-0024455f : Kernel data
20000000-2fffffff : Intel Corporation 440BX/ZX - 82443BX/ZX Host bridge
68000000-68000fff : Texas Instruments PCI1225
68001000-68001fff : Texas Instruments PCI1225 (#2)
e0000000-e3ffffff : PCI Bus #01
e4000000-e7ffffff : PCI Bus #01
 e4000000-e4ffffff : ATI Technologies Inc 3D Rage LT Pro AGP-133
```

```
 e6000000-e6000fff : ATI Technologies Inc 3D Rage LT Pro AGP-133
fffc0000-ffffffff : reserved
```

Once again, the values shown are hexadecimal ranges, and the string after the colon is the name of the "owner" of the I/O region.

As far as driver writing is concerned, the registry for I/O memory is accessed in the same way as for I/O ports, since they are actually based on the same internal mechanism.

To obtain and relinquish access to a certain I/O memory region, the driver should use the following calls:

```
int check_mem_region(unsigned long start, unsigned long len);
int request_mem_region(unsigned long start, unsigned long len,
   char *name);
int release_mem_region(unsigned long start, unsigned long len);
```

A typical driver will already know its own I/O memory range, and the sequence shown previously for I/O ports will reduce to the following:

```
if (check_mem_region(mem_addr, mem_size)) { printk("drivername:
 memory already in use\n"); return -EBUSY; }
 request_mem_region(mem_addr, mem_size, "drivername");
```

Resource Allocation in Linux 2.4

The current resource allocation mechanism was introduced in Linux 2.3.11 and provides a flexible way of controlling system resources. This section briefly describes the mechanism. However, the basic resource allocation functions (*request_region* and the rest) are still implemented (via macros) and are still universally used because they are backward compatible with earlier kernel versions. Most module programmers will not need to know about what is really happening under the hood, but those working on more complex drivers may be interested.

Linux resource management is able to control arbitrary resources, and it can do so in a hierarchical manner. Globally known resources (the range of I/O ports, say) can be subdivided into smaller subsets—for example, the resources associated with a particular bus slot. Individual drivers can then further subdivide their range if need be.

Resource ranges are described via a *resource* structure, declared in `<linux/ioport.h>`:

```
struct resource {
 const char *name;
 unsigned long start, end;
 unsigned long flags;
 struct resource *parent, *sibling, *child;
};
```

Top-level (root) resources are created at boot time. For example, the resource structure describing the I/O port range is created as follows:

```
struct resource ioport_resource =
   { "PCI IO", 0x0000, IO_SPACE_LIMIT, IORESOURCE_IO };
```

Thus, the name of the resource is `PCI IO`, and it covers a range from zero through `IO_SPACE_LIMIT`, which, according to the hardware platform being run, can be `0xffff` (16 bits of address space, as happens on the x86, IA-64, Alpha, M68k, and MIPS), `0xffffffff` (32 bits: SPARC, PPC, SH) or `0xffffffffffffffff` (64 bits: SPARC64).

Subranges of a given resource may be created with *allocate_resource*. For example, during PCI initialization a new resource is created for a region that is actually assigned to a physical device. When the PCI code reads those port or memory assignments, it creates a new resource for just those regions, and allocates them under `ioport_resource` or `iomem_resource`.

A driver can then request a subset of a particular resource (actually a subrange of a global resource) and mark it as busy by calling *__request_region*, which returns a pointer to a new `struct resource` data structure that describes the resource being requested (or returns `NULL` in case of error). The structure is already part of the global resource tree, and the driver is not allowed to use it at will.

An interested reader may enjoy looking at the details by browsing the source in *kernel/resource.c* and looking at the use of the resource management scheme in the rest of the kernel. Most driver writers, however, will be more than adequately served by *request_region* and the other functions introduced in the previous section.

This layered mechanism brings a couple of benefits. One is that it makes the I/O structure of the system apparent within the data structures of the kernel. The result shows up in */proc/ioports*, for example:

```
e800-e8ff : Adaptec AHA-2940U2/W / 7890
e800-e8be : aic7xxx
```

The range `e800-e8ff` is allocated to an Adaptec card, which has identified itself to the PCI bus driver. The *aic7xxx* driver has then requested most of that range—in this case, the part corresponding to real ports on the card.

The other advantage to controlling resources in this way is that it partitions the port space into distinct subranges that reflect the hardware of the underlying system. Since the resource allocator will not allow an allocation to cross subranges, it can block a buggy driver (or one looking for hardware that does not exist on the system) from allocating ports that belong to more than range—even if some of those ports are unallocated at the time.

Automatic and Manual Configuration

Several parameters that a driver needs to know can change from system to system. For instance, the driver must know the hardware's actual I/O addresses, or memory range (this is not a problem with well-designed bus interfaces and only applies to ISA devices). Sometimes you'll need to pass parameters to a driver to help it in finding its own device or to enable/disable specific features.

Depending on the device, there may be other parameters in addition to the I/O address that affect the driver's behavior, such as device brand and release number. It's essential for the driver to know the value of these parameters in order to work correctly. Setting up the driver with the correct values (i.e., configuring it) is one of the tricky tasks that need to be performed during driver initialization.

Basically, there are two ways to obtain the correct values: either the user specifies them explicitly or the driver autodetects them. Although autodetection is undoubtedly the best approach to driver configuration, user configuration is much easier to implement. A suitable trade-off for a driver writer is to implement automatic configuration whenever possible, while allowing user configuration as an option to override autodetection. An additional advantage of this approach to configuration is that the initial development can be done without autodetection, by specifying the parameters at load time, and autodetection can be implemented later.

Many drivers also have configuration options that control other aspects of their operation. For example, drivers for SCSI adapters often have options controlling the use of tagged command queuing, and the Integrated Device Electronics (IDE) drivers allow user control of DMA operations. Thus, even if your driver relies entirely on autodetection to locate hardware, you may want to make other configuration options available to the user.

Parameter values can be assigned at load time by *insmod* or *modprobe*; the latter can also read parameter assignment from a configuration file (typically */etc/modules.conf*). The commands accept the specification of integer and string values on the command line. Thus, if your module were to provide an integer parameter called *skull_ival* and a string parameter *skull_sval*, the parameters could be set at module load time with an *insmod* command like:

```
insmod skull skull_ival=666 skull_sval="the beast"
```

However, before *insmod* can change module parameters, the module must make them available. Parameters are declared with the `MODULE_PARM` macro, which is defined in *module.h*. `MODULE_PARM` takes two parameters: the name of the variable, and a string describing its type. The macro should be placed outside of any function and is typically found near the head of the source file. The two parameters mentioned earlier could be declared with the following lines:

```
int skull_ival=0;
char *skull_sval;

MODULE_PARM (skull_ival, "i");
MODULE_PARM (skull_sval, "s");
```

Five types are currently supported for module parameters: `b`, one byte; `h`, a short (two bytes); `i`, an integer; `l`, a long; and `s`, a string. In the case of string values, a pointer variable should be declared; *insmod* will allocate the memory for the user-supplied parameter and set the variable accordingly. An integer value preceding the type indicates an array of a given length; two numbers, separated by a hyphen, give a minimum and maximum number of values. If you want to find the author's description of this feature, you should refer to the header file `<linux/module.h>`.

As an example, an array that must have at least two and no more than four values could be declared as:

```
int skull_array[4];
MODULE_PARM (skull_array, "2-4i");
```

There is also a macro `MODULE_PARM_DESC`, which allows the programmer to provide a description for a module parameter. This description is stored in the object file; it can be viewed with a tool like *objdump*, and can also be displayed by automated system administration tools. An example might be as follows:

```
int base_port = 0x300;
MODULE_PARM (base_port, "i");
MODULE_PARM_DESC (base_port, "The base I/O port (default 0x300)");
```

All module parameters should be given a default value; *insmod* will change the value only if explicitly told to by the user. The module can check for explicit parameters by testing parameters against their default values. Automatic configuration, then, can be designed to work this way: if the configuration variables have the default value, perform autodetection; otherwise, keep the current value. In order for this technique to work, the "default" value should be one that the user would never actually want to specify at load time.

The following code shows how *skull* autodetects the port address of a device. In this example, autodetection is used to look for multiple devices, while manual configuration is restricted to a single device. The function *skull_detect* occurred earlier, in "Ports," while *skull_init_board* is in charge of device-specific initialization and thus is not shown.

```
/*
 * port ranges: the device can reside between
 * 0x280 and 0x300, in steps of 0x10. It uses 0x10 ports.
 */
#define SKULL_PORT_FLOOR 0x280
#define SKULL_PORT_CEIL 0x300
#define SKULL_PORT_RANGE 0x010
```

```
/*
 * the following function performs autodetection, unless a specific
 * value was assigned by insmod to "skull_port_base"
 */

static int skull_port_base=0; /* 0 forces autodetection */
MODULE_PARM (skull_port_base, "i");
MODULE_PARM_DESC (skull_port_base, "Base I/O port for skull");


static int skull_find_hw(void) /* returns the # of devices */
{
 /* base is either the load-time value or the first trial */
 int base = skull_port_base ? skull_port_base
        : SKULL_PORT_FLOOR;
 int result = 0;

 /* loop one time if value assigned; try them all if autodetecting */
 do {
       if (skull_detect(base, SKULL_PORT_RANGE) == 0) {
        skull_init_board(base);
        result++;
       }
       base += SKULL_PORT_RANGE; /* prepare for next trial */
 }
 while (skull_port_base == 0 && base < SKULL_PORT_CEIL);

 return result;
}
```

If the configuration variables are used only within the driver (they are not published in the kernel's symbol table), the driver writer can make life a little easier for the user by leaving off the prefix on the variable names (in this case, `skull_`). Prefixes usually mean little to users except extra typing.

For completeness, there are three other macros that place documentation into the object file. They are as follows:

`MODULE_AUTHOR (name)`
: Puts the author's name into the object file.

`MODULE_DESCRIPTION (desc)`
: Puts a description of the module into the object file.

`MODULE_SUPPORTED_DEVICE (dev)`
: Places an entry describing what device is supported by this module. Comments in the kernel source suggest that this parameter may eventually be used to help with automated module loading, but no such use is made at this time.

Doing It in User Space

A Unix programmer who's addressing kernel issues for the first time might well be nervous about writing a module. Writing a user program that reads and writes directly to the device ports is much easier.

Indeed, there are some arguments in favor of user-space programming, and sometimes writing a so-called user-space device driver is a wise alternative to kernel hacking.

The advantages of user-space drivers can be summarized as follows:

- The full C library can be linked in. The driver can perform many exotic tasks without resorting to external programs (the utility programs implementing usage policies that are usually distributed along with the driver itself).
- The programmer can run a conventional debugger on the driver code without having to go through contortions to debug a running kernel.
- If a user-space driver hangs, you can simply kill it. Problems with the driver are unlikely to hang the entire system, unless the hardware being controlled is *really* misbehaving.
- User memory is swappable, unlike kernel memory. An infrequently used device with a huge driver won't occupy RAM that other programs could be using, except when it is actually in use.
- A well-designed driver program can still allow concurrent access to a device.

An example of a user-space driver is the X server: it knows exactly what the hardware can do and what it can't, and it offers the graphic resources to all X clients. Note, however, that there is a slow but steady drift toward frame-buffer-based graphics environments, where the X server acts only as a server based on a real kernel-space device driver for actual graphic manipulation.

Usually, the writer of a user-space driver implements a server process, taking over from the kernel the task of being the single agent in charge of hardware control. Client applications can then connect to the server to perform actual communication with the device; a smart driver process can thus allow concurrent access to the device. This is exactly how the X server works.

Another example of a user-space driver is the *gpm* mouse server: it performs arbitration of the mouse device between clients, so that several mouse-sensitive applications can run on different virtual consoles.

Sometimes, though, the user-space driver grants device access to a single program. This is how *libsvga* works. The library, which turns a TTY into a graphics display, gets linked to the application, thus supplementing the application's capabilities

without resorting to a central authority (e.g., a server). This approach usually gives you better performance because it skips the communication overhead, but it requires the application to run as a privileged user (this is one of the problems being solved by the frame buffer device driver running in kernel space).

But the user-space approach to device driving has a number of drawbacks. The most important are as follows:

- Interrupts are not available in user space. The only way around this (on the x86) is to use the *vm86* system call, which imposes a performance penalty.*
- Direct access to memory is possible only by *mmap*ping */dev/mem*, and only a privileged user can do that.
- Access to I/O ports is available only after calling *ioperm* or *iopl*. Moreover, not all platforms support these system calls, and access to */dev/port* can be too slow to be effective. Both the system calls and the device file are reserved to a privileged user.
- Response time is slower, because a context switch is required to transfer information or actions between the client and the hardware.
- Worse yet, if the driver has been swapped to disk, response time is unacceptably long. Using the *mlock* system call might help, but usually you'll need to lock several memory pages, because a user-space program depends on a lot of library code. *mlock*, too, is limited to privileged users.
- The most important devices can't be handled in user space, including, but not limited to, network interfaces and block devices.

As you see, user-space drivers can't do that much after all. Interesting applications nonetheless exist: for example, support for SCSI scanner devices (implemented by the *SANE* package) and CD writers (implemented by *cdrecord* and other tools). In both cases, user-level device drivers rely on the "SCSI generic" kernel driver, which exports low-level SCSI functionality to user-space programs so they can drive their own hardware.

In order to write a user-space driver, some hardware knowledge is sufficient, and there's no need to understand the subtleties of kernel software. We won't discuss user-space drivers any further in this book, but will concentrate on kernel code instead.

One case in which working in user space might make sense is when you are beginning to deal with new and unusual hardware. This way you can learn to manage your hardware without the risk of hanging the whole system. Once you've

---

* The system call is not discussed in this book because the subject matter of the text is kernel drivers; moreover, *vm86* is too platform specific to be really interesting.

done that, encapsulating the software in a kernel module should be a painless operation.

Backward Compatibility

The Linux kernel is a moving target—many things change over time as new features are developed. The interface that we have described in this chapter is that provided by the 2.4 kernel; if your code needs to work on older releases, you will need to take various steps to make that happen.

This is the first of many "backward compatibility" sections in this book. At the end of each chapter we'll cover the things that have changed since version 2.0 of the kernel, and what needs to be done to make your code portable.

For starters, the `KERNEL_VERSION` macro was introduced in kernel 2.1.90. The *sysdep.h* header file contains a replacement for kernels that need it.

Changes in Resource Management

The new resource management scheme brings in a few portability problems if you want to write a driver that can run with kernel versions older than 2.4. This section discusses the portability problems you'll encounter and how the *sysdep.h* header tries to hide them.

The most apparent change brought about by the new resource management code is the addition of *request_mem_region* and related functions. Their role is limited to accessing the I/O memory database, without performing specific operations on any hardware. What you can do with earlier kernels, thus, is to simply not call the functions. The *sysdep.h* header easily accomplishes that by defining the functions as macros that return `0` for kernels earlier than 2.4.

Another difference between 2.4 and earlier kernel versions is in the actual prototypes of *request_region* and related functions.

Kernels earlier than 2.4 declared both *request_region* and *release_region* as functions returning `void` (thus forcing the use of *check_region* beforehand). The new implementation, more correctly, has functions that return a pointer value so that an error condition can be signaled (thus making *check_region* pretty useless). The actual pointer value will not generally be useful to driver code for anything other than a test for `NULL`, which means that the request failed.

If you want to save a few lines of code in your drivers and are not concerned about backward portability, you could exploit the new function calls and avoid using *check_region* in your code. Actually, *check_region* is now implemented on top of *request_region*, releasing the I/O region and returning success if the request is fulfilled; the overhead is negligible because none of these functions is ever called from a time-critical code section.

If you prefer to be portable, you can stick to the call sequence we suggested earlier in this chapter and ignore the return values of *request_region* and *release_region*. Anyway, *sysdep.h* declares both functions as macros returning 0 (success), so you can both be portable and check the return value of every function you call.

The last difference in the I/O registry between version 2.4 and earlier versions of the kernel is in the data types used for the `start` and `len` arguments. Whereas new kernels always use `unsigned long`, older kernels used shorter types. This change has no effect on driver portability, though.

Compiling for Multiprocessor Systems

Version 2.0 of the kernel didn't use the `CONFIG_SMP` configuration option to build for SMP systems; instead, choice was made a global assignment in the main kernel *makefile*. Note that modules compiled for an SMP machine will not work in a uniprocessor kernel, and vice versa, so it is important to get this one right.

The sample code accompanying this book automatically deals with SMP in the makefiles, so the code shown earlier need not be copied in each module. However, we do not support SMP under version 2.0 of the kernel. This should not be a problem because multiprocessor support was not very robust in Linux 2.0, and everyone running SMP systems should be using 2.2 or 2.4. Version 2.0 is covered by this book because it's still the platform of choice for small embedded systems (especially in its no-MMU implementation), but no such system has multiple processors.

Exporting Symbols in Linux 2.0

The Linux 2.0 symbol export mechanism was built around a function called *register_symtab*. A Linux 2.0 module would build a table describing all of the symbols to be exported, then would call *register_symtab* from its initialization function. Only symbols that were listed in the explicit symbol table were exported to the kernel. If, instead, the function was not called at all, all global symbols were exported.

If your module doesn't need to export any symbols, and you don't want to declare everything as `static`, just hide global symbols by adding the following line to *init_module*. This call to *register_symtab* simply overwrites the module's default symbol table with an empty one:

```
register_symtab(NULL);
```

This is exactly how *sysdep.h* defines `EXPORT_NO_SYMBOLS` when compiling for version 2.0. This is also why `EXPORT_NO_SYMBOLS` must appear within *init_module* to work properly under Linux 2.0.

If you do need to export symbols from your module, you will need to create a symbol table structure describing these symbols. Filling a Linux 2.0 symbol table structure is a tricky task, but kernel developers have provided header files to simplify things. The following lines of code show how a symbol table is declared and exported using the facilities offered by the headers of Linux 2.0:

```
static struct symbol_table skull_syms = {

#include <linux/symtab_begin.h>
  X(skull_fn1),
  X(skull_fn2),
  X(skull_variable),
#include <linux/symtab_end.h>
        };

 register_symtab(&skull_syms);
```

Writing portable code that controls symbol visibility takes an explicit effort from the device driver programmer. This is a case where it is not sufficient to define a few compatibility macros; instead, portability requires a fair amount of conditional preprocessor code, but the concepts are simple. The first step is to identify the kernel version in use and to define some symbols accordingly. What we chose to do in *sysdep.h* is define a macro `REGISTER_SYMTAB()` that expands to nothing on version 2.2 and later and expands to *register_symtab* on version 2.0. Also, `__USE_OLD_SYMTAB__` is defined if the old code must be used.

By making use of this code, a module that exports symbols may now do so portably. In the sample code is a module, called *misc-modules/export.c*, that does nothing except export one symbol. The module, covered in more detail in "Version Control in Modules" in Chapter 11, includes the following lines to export the symbol portably:

```
#ifdef __USE_OLD_SYMTAB__
 static struct symbol_table export_syms = {
  #include <linux/symtab_begin.h>
  X(export_function),
  #include <linux/symtab_end.h>
 };
#else
 EXPORT_SYMBOL(export_function);
#endif

int export_init(void)
{
 REGISTER_SYMTAB(&export_syms);
 return 0;
}
```

If `__USE_OLD_SYMTAB__` is set (meaning that you are dealing with a 2.0 kernel), the *symbol_table* structure is defined as needed; otherwise, `EXPORT_SYMBOL` is used to export the symbol directly. Then, in *init_module*, `REGISTER_SYMTAB` is called; on anything but a 2.0 kernel, it will expand to nothing.

Module Configuration Parameters

`MODULE_PARM` was introduced in kernel version 2.1.18. With the 2.0 kernel, no parameters were declared explicitly; instead, *insmod* was able to change the value of any variable within the module. This method had the disadvantage of providing user access to variables for which this mode of access had not been intended; there was also no type checking of parameters. `MODULE_PARM` makes module parameters much cleaner and safer, but also makes Linux 2.2 modules incompatible with 2.0 kernels.

If 2.0 compatibility is a concern, a simple preprocessor test can be used to define the various `MODULE_` macros to do nothing. The header file *sysdep.h* in the sample code defines these macros when needed.

Quick Reference

This section summarizes the kernel functions, variables, macros, and */proc* files that we've touched on in this chapter. It is meant to act as a reference. Each item is listed after the relevant header file, if any. A similar section appears at the end of every chapter from here on, summarizing the new symbols introduced in the chapter.

`__KERNEL__`
`MODULE`
: Preprocessor symbols, which must both be defined to compile modularized kernel code.

`__SMP__`
: A preprocessor symbol that must be defined when compiling modules for symmetric multiprocessor systems.

```
int init_module(void);
void cleanup_module(void);
```
: Module entry points, which must be defined in the module object file.

```
#include <linux/init.h>
module_init(init_function);
module_exit(cleanup_function);
```
: The modern mechanism for marking a module's initialization and cleanup functions.

`#include <linux/module.h>`
Required header. It must be included by a module source.

`MOD_INC_USE_COUNT;`
`MOD_DEC_USE_COUNT;`
`MOD_IN_USE;`
Macros that act on the usage count.

/proc/modules
The list of currently loaded modules. Entries contain the module name, the amount of memory each module occupies, and the usage count. Extra strings are appended to each line to specify flags that are currently active for the module.

`EXPORT_SYMTAB;`
Preprocessor macro, required for modules that export symbols.

`EXPORT_NO_SYMBOLS;`
Macro used to specify that the module exports no symbols to the kernel.

`EXPORT_SYMBOL (symbol);`
`EXPORT_SYMBOL_NOVERS (symbol);`
Macro used to export a symbol to the kernel. The second form exports without using versioning information.

`int register_symtab(struct symbol_table *);`
Function used to specify the set of public symbols in the module. Used in 2.0 kernels only.

`#include <linux/symtab_begin.h>`
`X(symbol),`
`#include <linux/symtab_end.h>`
Headers and preprocessor macro used to declare a symbol table in the 2.0 kernel.

`MODULE_PARM(variable, type);`
`MODULE_PARM_DESC (variable, description);`
Macros that make a module variable available as a parameter that may be adjusted by the user at module load time.

`MODULE_AUTHOR(author);`
`MODULE_DESCRIPTION(description);`
`MODULE_SUPPORTED_DEVICE(device);`
Place documentation on the module in the object file.

`#include <linux/version.h>`
Required header. It is included by `<linux/module.h>`, unless `__NO_VERSION__` is defined (see later in this list).

`LINUX_VERSION_CODE`
Integer macro, useful to `#ifdef` version dependencies.

`char kernel_version[] = UTS_RELEASE;`
Required variable in every module. `<linux/module.h>` defines it, unless `__NO_VERSION__` is defined (see the following entry).

`__NO_VERSION__`
Preprocessor symbol. Prevents declaration of `kernel_version` in `<linux/module.h>`.

`#include <linux/sched.h>`
One of the most important header files. This file contains definitions of much of the kernel API used by the driver, including functions for sleeping and numerous variable declarations.

`struct task_struct *current;`
The current process.

`current->pid`
`current->comm`
The process ID and command name for the current process.

`#include <linux/kernel.h>`
`int printk(const char * fmt, ...);`
The analogue of *printf* for kernel code.

```
#include <linux/malloc.h>
void *kmalloc(unsigned int size, int priority);
void kfree(void *obj);
```
Analogue of *malloc* and *free* for kernel code. Use the value of `GFP_KERNEL` as the priority.

```
#include <linux/ioport.h>
int check_region(unsigned long from, unsigned long extent);
struct resource *request_region(unsigned long from, unsigned
      long extent, const char *name);
void release_region(unsigned long from, unsigned long
      extent);
```
Functions used to register and release I/O ports.

```
int check_mem_region (unsigned long start, unsigned long
      extent);
struct resource *request_mem_region (unsigned long start,
      unsigned long extent, const char *name);
void release_mem_region (unsigned long start, unsigned long
      extent);
```

Macros used to register and release I/O memory regions.

/proc/ksyms
The public kernel symbol table.

/proc/ioports
The list of ports used by installed devices.

/proc/iomem
The list of allocated memory regions.