CHAPTER 7

Kernel Modules and Device Drivers

"If you think I'm a bad driver, you should see me putt."
—Snoopy

For me, the fun part of embedded programming is seeing the computer interact with its physical environment, i.e., actually do something. In this chapter we'll use the parallel port on our target machine to provide some limited I/O capability for the thermostat program developed in the previous chapter. But first some background.

Kernel Modules

Installable kernel modules offer a very useful way to extend the functionality of the basic Linux kernel and add new features without having to rebuild the kernel. A key feature is that modules may be dynamically loaded when their functionality is required and subsequently unloaded when no longer needed. Modules are particularly useful for things like device drivers and `/proc` files.

Given the range and diversity of hardware that Linux supports, it would be impractical to build a kernel image that included all of the possible device drivers. Instead the kernel includes only drivers for boot devices and other common hardware such as serial and parallel ports, IDE and SCSI drives, and so on. Other devices are supported as loadable modules and only the modules needed in a given system are actually loaded.

Loadable modules are unlikely to be used in a production embedded environment because we know in advance exactly what hardware the system must support and so we simply build that support into the kernel image. Nevertheless, modules are still useful when testing a new driver. You don't need to build a new kernel image every time you find and fix a problem in your driver code. Just load it as a module and try it.

Keep in mind that modules execute in Kernel Space at Privilege Level 0 and thus are capable of bringing down the entire system.

A Module Example

cd /$HOME/BlueCat/demo.x86/apps/module and open the file hello.c. Note first of all the two include files required for a kernel module. This is a trivial example of a loadable kernel module. It contains two very simple functions; hello_init() and hello_exit(). The last two lines:

```
module_init(hello_init);
module_exit(hello_exit);
```

identify these functions to the module loader and unloader utilities. Every module must include an init function and an exit function. The function specified to the module_init() macro, in this case hello_init(), is called by insmod, the shell command that installs a module. The function specified to module_exit(), hello_exit(), is called by rmmod, the command that removes a module.

In this example both functions simply print a message on the console using printk, the kernel equivalent of printf. C library functions like printf are intended to run from user space making use of operating system features like redirection. These facilities aren't available to kernel code. Rather than writing directly to the console, printk writes to a circular buffer and then wakes up the klogd process to deal with the message by either printing it to the console and/or writing it to the system log.

Note the KERN_ALERT at the beginning of the printk format strings. This is the symbolic representation of a *loglevel* that determines whether or not the message appears on the console. Loglevels range from 0 to 7 with lower numbers having higher priority. If the loglevel is numerically less than the kernel integer variable console_loglevel then the message appears on the console. In any case, printk messages always show up in the file /var/log/messages regardless of the loglevel.

The hello example also shows how to specify module parameters, local variables whose values can be entered on the command line that loads the module. This is done with the module_param() macro. The first argument to module_param() is a variable name, the second is the variable type, and the third is a "permission flag" that controls access to the representation of the module parameter in *sysfs*, a new feature of the 2.6 kernel that offers cleaner access to driver internals than the /proc filesystem. A safe value for now is S_IRUGO meaning the value is read-only.

Variable types can be charp—pointer to a character string, or int, short, long—various size integers or their unsigned equivalent with "u" prepended.

Make hello with a simple make command and then try it out. As root user, enter the command:

```
insmod hello.ko my_string="name" my_int=47
```

If you're running from the command line, i.e., not in X windows, you should see the message printed by hello_init(). Oddly enough, printk messages do not appear in shell windows running under X windows, KDE for example, regardless of the loglevel. You can see what printk did with the command:

```
tail /var/log/messages
```

This prints out the last few lines of the messages log file. A useful variation on this command is:

```
tail –f /var/log/messages
```

This version continues running and outputs new text sent to messages.

If you initially logged on under your normal user name and used su to become root, you'll have to preface all of the module commands with /sbin/ because /sbin is not normally part of the path for a normal user. Now try the command lsmod. This lists the currently loaded modules in the reverse order in which they were loaded. hello should be the first entry. lsmod also gives a "usage count" for each module and shows what modules depend on other modules. The same information is available in /proc/modules.

Now execute the command rmmod hello. You should see the message printed by hello_exit(). Finally execute lsmod again to see that hello is no longer listed. Note incidentally that rmmod does not require the ".ko" extension. Once loaded, modules are identified by their base name.

A module may, and usually does, contain references to external symbols such as printk. How do these external references get resolved? insmod resolves them against the kernel's symbol table, which is loaded in memory as part of the kernel boot process. Furthermore, any nonstatic symbols defined in the module are added to the kernel symbol table and are available for use by subsequently loaded modules. So the only external symbols a module can reference are those built into the kernel image or previously loaded modules. The kernel symbol table is available in /proc/ksyms.

"Tainting" the Kernel

You should have seen another message when you installed hello, just before the message from **hello_init()**:

```
hello: module license 'unspecified' taints kernel
```

What the heck does that mean? Apparently, kernel developers were getting tired of trying to cope with bug reports involving kernel modules for which no source was available, that is, modules not released under an Open Source license such as GPL. Their solution to the problem was to invent a MODULE_LICENSE() macro whereby you can declare that a module is indeed Open Source. The format is:

```
MODULE_LICENSE ("<approved string>")
```

Where <approved_string> is one of the ASCII text strings found in linux/include/linux/module.h. Among these, not surprisingly, is "GPL." If you distribute your module in accordance with the terms of an Open Source license such as GPL, then you are permitted to include the corresponding MODULE_LICENSE() invocation in your code and loading your module will not produce any complaints.

If you install a module that produces the above warning and the system subsequently crashes, the crash documentation (core dump) will reveal the fact that a non-Open Source module was loaded. Your kernel has been "tainted" by code that no one has access to. If you submit the crash documentation to the kernel developer group it will be ignored[1].

Add the following line to hello.c just below the two **module_param()** statements:

```
MODULE_LICENSE("GPL");
```

Remake hello and verify that installing the new version does not generate the warning.

Building Kernel Modules

The build process for kernel modules is somewhat more involved than the single line Makefile we encountered in the last chapter. Take a look at the Makefile for hello.

[1] What's not clear to me is how the tainted kernel provision is enforced. An unscrupulous device vendor could very easily include a MODULE_LICENSE() entry in his driver code but still not release the source. What happens when that module causes a kernel fault? I suspect the Open Source community is relying on public approbation to "out" vendors who don't play by the rules. What else is there?

The complication arises because modules must be built within the context of the kernel itself. The upshot of this is that, if you're building a module outside of the kernel tree as we are now, the Makefile has to change into the top-level kernel source directory and invoke the Makefile there. That in turn invokes the Makefile in the directory we started from.

To keep things simple and minimize the typing required, the kernel developers created a clever mechanism that in effect invokes the Makefile twice. The essence of this Makefile is a conditional statement based on the environment variable KERNELRELEASE, which is defined in the kernel build context and not defined otherwise.

The top half of the conditional shows what happens when KERNELRELEASE is not defined, that is when we invoke the Makefile from our own directory. This case creates two environment variables; KERNELDIR, pointing to the top-level kernel directory, and PWD, the current directory. The line

```
$(MAKE) -C $(KERNELDIR) SUBDIRS=$(PWD) modules
```

invokes make again telling it to change (-C) to the kernel directory and invoke the Makefile there. The SUBDIRS option says include the Makefile in directory PWD and build the target modules. It's left as the proverbial exercise for the reader to figure out how the two environment variables get set correctly.

The next time our Makefile is invoked, KERNELRELEASE is defined and we just execute the line:

```
obj-m := hello.o
```

This turns out to be sufficient to direct the kernel build system to compile hello.c and then turn it into the loadable module hello.ko. You'll note that in the process, it creates a number of temporary files and a directory. Hence the clean target. If a module is built from two or more source files, the line above expands to something like:

```
obj-m := module.o
module-objs := file1.o file2.o
```

When creating makefiles for any kind of project, whether it be a kernel module or a user space application, it's generally a good idea to start with an existing model. The Makefile described here is a good starting point for other kernel module projects.

The Role of a Module

As an extension of the kernel, a module's role is to provide some service or set of services to applications. Unlike an application program, a module does not execute on its own. Rather, it patiently waits until some application program invokes its service.

But how does that application program gain access to the module's services? That's the role of the module_ init() function. By calling one or more kernel functions, module_ init () "registers" the module's "capabilities" with the kernel. In effect the module is saying "Here I am and here's what I can do."

Figure 7-1 illustrates graphically how this registration process works. The init_module() function calls some register_capability() function passing as an argument a pointer to a structure containing pointers to functions within the module. The register_capability() function puts the pointer argument into an internal "capabilities" data structure. The system then defines a protocol for how applications get access to the information in the capabilities data structure through system calls.

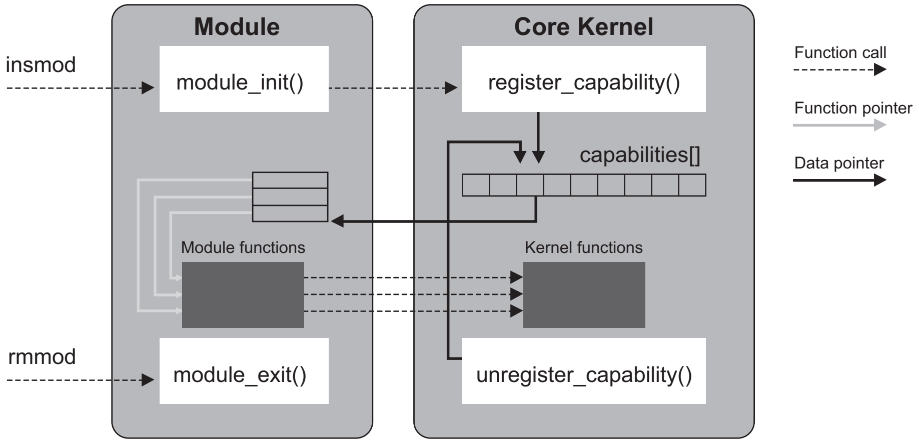

Figure 7-1: Linking a Module to the Kernel[2]

A specific example of registering a capability is the pair of functions register_chrdev_region() and alloc_chrdev_region() that register the capabilities of a "character" device driver. The way in which an application program gains access to the services of a character device driver is through the open() system call. This is described in more detail shortly.

[2] Rubini, Allesandro and Jonathan Corbet, *Linux Device Drivers, 2nd Ed*, p. 18.

If a module is no longer needed and is removed with the `rmmod` command, the `module_exit()` function should "unregister" the module's capabilities, freeing up the entry in the capabilities data structure.

What's a Device Driver Anyway?

There are probably as many definitions of a *device driver* as there are programmers who write them. Fundamentally, a device driver is simply a way to "abstract out" the details of peripheral hardware so the application programmer doesn't have to worry about them.

In simple systems a driver may be nothing more than a library of functions linked directly to the application. In general purpose operating systems, device drivers are often independently loaded programs that communicate with applications through some OS-specific protocol. In multitasking systems like Linux the driver should be capable of establishing multiple "channels" to the device originating from different application processes. In all cases though the driver is described in terms of an API that defines the services the driver is expected to support.

The device driver paradigm takes on additional significance in a protected mode environment such as Linux. There are two reasons for this. First, User Space application code is normally not allowed to execute I/O instructions. This can only be done in Kernel Space at Privilege Level 0. So a set of device driver functions linked directly to the application simply won't work. The driver must execute in Kernel Space. Actually, there are some hoops you can jump through to allow I/O access from User Space but it's better to avoid them.

The second problem is that User Space is swappable. This means that while an I/O operation is in process, the user's data buffer could get swapped out to disk. And when that page gets swapped back in, it will very likely be at a different physical address. So data to be transferred to or from a peripheral device must reside in Kernel Space, which is not swappable. The driver then has to take care of transferring that data between Kernel Space and User Space.

Linux Device Drivers

Unix, and by extension Linux, divides the world of peripheral devices into three categories:

1. Character
2. Block
3. Network

The principal distinction between character and block is that the latter, such as disks, are randomly accessible, that is, you can move back and forth within a stream of characters. Furthermore data on block devices is usually transferred in one or more blocks at a time and prefetched and cached. With character devices the stream moves in one direction only. You can't for example go back and reread a character from a serial port. Block devices generally have a filesystem associated with them whereas character devices don't.

Network devices are different in that they handle "packets" of data for multiple protocol clients rather than a "stream" of data for a single client. Furthermore, data arrives at a network device asynchronously from sources outside the system. These differences necessitate a different interface between the kernel and the device driver.

The /dev Directory

While other OSes treat devices as files, Linux goes one step further in actually creating a directory for devices. Typically this is `/dev`. In a shell window on the workstation, `cd /dev` and `ls –l hd*`. You'll get a very large number of entries. Notice that the first character in the flags field for all these entries is "b". This indicates that the directory entry represents a "block" device. In fact all of these entries are hard disk devices.

Between the Group field and the date stamp, where the file size normally appears, is a pair of numbers separated by a comma. These are referred to as the "major device number" and the "minor device number" respectively. The major device number identifies the device driver. The minor device number is used only by the driver itself to distinguish among possibly different types of devices the driver can handle.

Now do `ls –l tty*`. The first character in the flags field is now "c" indicating that these are character devices. Character devices and block devices each have their own table of device driver pointers. Links allow the creation of logical devices that can then be mapped to system-specific physical devices. Try `ls –l mouse`. In my system this is a link to `psaux`, the device for a PS2-style mouse.

Prior to the 2.6 series, the major and minor device numbers were each eight bits and there was a one-to-one correspondence between major numbers and device drivers. In effect, the major number was an index into a table of pointers to device drivers. Now a device number is a 32-bit integer, `typedef`'d as `dev_t`, where the high-order 12 bits are the major number and the remaining bits are the minor number. The registration mechanism now allows multiple drivers to attach to the same major number.

Entries in `/dev` are created with the `mknod` command as, for example,

```
mknod /dev/ttyS1 c 4 65
```

This creates a `/dev` entry named `ttyS1`. It's a character device with major device number 4 and minor device number 65. There's no reference to a device driver here. All we've done with `mknod` is create an entry in the filesystem so application programs can "open" the device. But before that can happen we'll have to "register" a character device driver with major number 4 to create the connection.

The Low Level I/O API

The set of User Space system functions closest to the device drivers is termed "low level I/O." For the kind of device we're dealing with in this chapter, low level I/O is probably the most useful because the calls are inherently synchronous. That is, the call doesn't return until the data transfer has completed.

The drawback to the low-level API is that every call requires a switch from User Space to Kernel Space and back again. This can lead to a fair amount of overhead. By contrast, the "stream" I/O API buffers data until there's enough to justify the overhead of the system call. For general file transfers the stream API usually makes more sense. The downside for embedded applications is that, for example, when a write operation returns, you don't know if the data has gotten to the device yet.

Figure 7-2 shows the basic elements of the low level I/O API.

- *OPEN*. Establishes a connection between the calling process and the file or device. `path` is the directory entry to be opened. `oflags` is a bitwise set of flags specifying access mode and must include one of the following:

– O_RDONLY	Open for read-only
– O_WRONLY	Open for write-only
– O_RDWR	Open for both reading and writing

  Additionally oflags may include one or more of the following modes:

– O_APPEND	Place written data at the end of the file
– O_TRUNC	Set the file length to zero, discarding existing contents
– O_CREAT	Create the file if necessary. Requires the function call with three arguments where mode is the initial permissions.

If OPEN is successful it returns a nonnegative integer representing a "file descriptor." This value is then used in all subsequent I/O operations to identify this specific connection.

- *READ and WRITE*. These functions transfer data between the process and the file or device. filedes is the file descriptor returned by OPEN. buf is a pointer to the data to be transferred and count is the size of buf in bytes. If the return value is nonnegative it is the number of bytes actually transferred, which may be less than count.
- *CLOSE*. When a process is finished with a particular device or file, it can close the connection, which invalidates the file descriptor and frees up any resources to be used for another process/file connection. It is good practice to close any unneeded connections because there is typically a limited number of file descriptors available.
- *IOCTL*. This is the "escape hatch" to deal with specific device idiosyncrasies. For example a serial port has a baud rate and may have a modem attached to it. The manner in which these features are controlled is specific to the device. So each device driver can establish its own protocol for the IOCTL function.

```
int open (const char *path, int oflags);
int open (const char *path, int oflags, mode_t mode);
size_t read (int filedes, void *buf, size_t count);
size_t write (int filedes, void *buf, size_t count);
int close (int filedes);
int ioctl (int filedes, int cmd, ...);
```

Figure 7-2: Stream I/O API

A characteristic of most Linux system calls is that, in case of an error, the function return value is –1 and doesn't directly indicate the source of the error. The actual error code is placed in the global variable errno. So you should always test the function return for a negative value and then inspect errno to find out what really happened. Or better yet call perror(), which prints a sensible error message on the console.

There are a few other low level I/O functions but they're not particularly relevant to this discussion.

Internal Driver Structure

The easiest way to get a feel for the structure of a device driver is to take a look at the code for a relatively simple one. In demo.x86/apps/parport is a simple driver called parport.c. This simply reads and writes the PC's parallel port in a way that supports our thermostat example. Open the file with your favorite editor.

Basically, a driver consists of a set of functions that mirror the functions in the low level API. However the driver's functions are called by the kernel, in Kernel Space, in response to an application program calling a low-level I/O function.

init() and exit()

Let's start near the end of the file with the parport_init() function. The first thing the driver must do is gain exclusive access to the relevant I/O ports. That's done with the function request_region(), which returns a non-NULL value if it succeeds. The arguments to request_region() are a base port number, the number of ports to allocate and the name of the device.

For the parallel port BASE represents the data register and is normally 0x378. BASE + 1 is the status register. For the thermostat example the data port will drive a pair of LEDs representing the heater and the alarm. The status port will monitor a set of pushbuttons that allow us to raise or lower the measured "temperature."

Next we need to allocate a major device number that serves as the link to our driver. Many of the "lower" device numbers have been allocated to specific device classes and, in particular, the parallel port is assigned major number 6. If we were building the driver to run on our workstation, we would probably want to select a different major number to avoid interfering with the standard parallel port driver.

But since we intend to run the driver on a target that we have complete control over, we'll go ahead and use major number 6 for convenience. The macro MKDEV(int, int) makes a dev_t out of a major and minor device number. There are also macros for retrieving the major and minor elements of a dev_t: MAJOR(dev_t) and MINOR(dev_t). Good practice dictates using these macros rather than manipulating the bit fields explicitly. There's no guarantee that the bit assignments for major and minor will remain the same in later kernel revisions. Besides, the code is more readable.

The function register_chrdev_region() registers a contiguous range of device numbers starting from a specified base. In this case we're only asking for one number. The string argument is the name of the device. The function returns zero if it succeeds. If it doesn't succeed, you don't have access to the device number range.

Having allocated a major device number, we can now register our driver with the kernel. This involves two steps:

1. Allocate or initialize a cdev data structure with cdev_alloc() or cdev_init()
2. Tell the kernel about it with cdev_add()

cdev_alloc() dynamically allocates and initializes a cdev structure. There are two fields you need to initialize before calling cdev_add().

1. The "owner," which is almost always the macro THIS_MODULE.
2. A pointer to a file_operations structure. This contains pointers to the functions that implement the driver API for this device. The file_operations structure is just above parport_init(). Our driver is quite simple and only implements four of the driver functions. The kernel takes some sensible default action for all of the driver functions that are not specified. Note by the way that file_operations also contains an owner field.

Finally, we register the device driver with a call to cdev_add(). The arguments are:

- Pointer to the cdev structure
- The based device number we got from alloc_chrdev_region()
- The number of devices we're registering, in this case one

In many cases, you're device will have its own structure into which you may want to embedded the cdev structure. Then you call cdev_init() passing in the address of the cdev structure and a pointer to the file_operations structure. You still need to initialize the owner field.

If parport_init() succeeds, the return value is zero. A nonzero return value indicates an error condition and the module is not loaded. Note that if parport_init() fails, any resources successfully allocated up to the point of failure must be returned before exiting. In this case, failure to register the device requires us to release the I/O region.

As might be expected, parport_exit() simply reverses the actions of parport_init(). It unregisters the device and releases the I/O region.

open() and release()

Move back to the top of the file around line 29. The kernel calls parport_open() when an application calls open() specifying /dev/parport as the path. The arguments to open both represent a "file" but in a slightly different way. struct inode represents a file on a disk or a physical device. Among the fields in inode are the major and minor device numbers. struct file represents an *open* file. Every open file in the system has an associated file structure that is created by the kernel on open() and passed to every function that operates on the file until the final close(). struct file maintains information like the access mode, readable or writable, and the current file position. There's even a void *private_data field that allows the driver to add its own information if necessary.

Neither inode nor file is relevant to our simple parport driver. In fact parport_open() does nothing. We could have left it out and simply let the kernel take the default action, but it's useful to see the function prototype since open is used in most drivers.

Interestingly enough, the driver equivalent of close is called release. Again, in this case there's nothing to do.

read() and write()

Moving down the file we come to the interesting functions, parport_read() and parport_write(). The arguments to read and write are:

- A struct file (see above).
- A pointer to a data buffer in *User Space*.
- A count representing the size of the buffer.
- A pointer to a file position argument. If the function is operating on a file, then any data transferred will change the file position. It is the function's responsibility to update this field appropriately.

parport_read() is a little unusual in that it will never return more than two bytes of data regardless of the count value. We invert the status byte because a pressed button reads as a logic 0 on the status lines but we would like to see a button press appear as a 1 in the program. However the MSB of the status register is already inverted so we don't invert it here.

The port registers are read into a local char array in Kernel Space. copy_to_user() copies this local array to the application's buffer in User Space.

parport_write() takes the more conventional approach of writing out an arbitrary size buffer even though only the last character written will be visible on the LEDs. If count is greater than 2 we dynamically allocate a Kernel Space buffer of count bytes.

Normally count is 1 and it's rather silly to allocate a buffer of one byte so we just use a local variable. In either case, `copy_from_user()` copies the application's buffer into the Kernel Space buffer. Then just write it out and return the dynamically allocated buffer if necessary.

Figure 7-3 attempts to capture graphically the essence of driver processing for a write operation. The user process calls `write()`. This invokes the kernel through an INT instruction where the write operation is conveyed as an index into the `syscall[]` table. The `filedes` argument contains, among other things, a major device number so that the kernel's write function can access the driver's `file_operations` structure. The kernel calls the driver's write function, which copies the User Space buffer into Kernel Space.

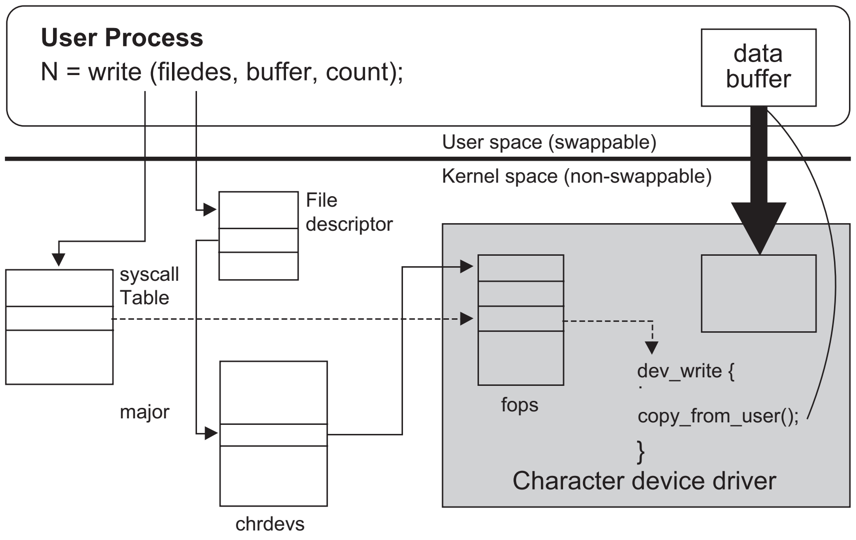

Figure 7-3: Driver Processing – Write

Building and Running the Driver

As usual, it's a good idea to try out a driver in the workstation environment before moving it to the target. The Makefile in `parport/` is slightly more complicated than the one in `module/` owing to the requirement to build the driver for either the workstation or target environment. When we're building for the target, the `SETUP.sh` script will define KERNELDIR so as to start the Makefile in the source tree for the target kernel, not the one running on the workstation. When building for the workstation, KERNELDIR is defined within the Makefile. Hence the line:

```
ifeq ($(KERNELDIR),)
```

Go ahead and build the driver module. Chances are your workstation kernel has printer support enabled, probably as a module. Actually it's a set of three modules. Before you can successfully install your parport module, you need to remove the currently running parport module that supports printing. As root user, execute the following commands in the order given:

```
/sbin/rmmod lp
/sbin/rmmod parport_pc
/sbin/rmmod parport
```

Do `cat /proc/ioports` to verify that the port range of 0x378 to 0x37f is not allocated. Now execute:

```
/sbin/insmod parport.ko
```

Now `/proc/ioports` should show 0x378 to 0x37f allocated to parport. We won't be able to properly exercise parport until after the next section.

Now rename parport.ko to parport.ko.ws (for workstation version) and rebuild parport for the target. Be sure to run `SETUP.sh` script first.

Before we can actually load and exercise the driver on the target, we have to create a device node on the target filesystem. `cd ../../shell` and edit `shell.spec`. Below the line that begins "`mknod /dev/ttyS1`" add the line

```
mknod /dev/parport  c 6 0
```

You'll also need to remove printer support from the Bluecat target kernel. From the `shell/` directory run `make xconfig`. Under "Device Drivers," "Parallel port support," deselect "Parallel port support."

Now rebuild the kernel with `make kernel` and the root filesystem with `make rootfs`, and copy both to `/home`. Reboot your target.

One more "gotcha." Bluecat lite doesn't include the module utilities. We're going to "cheat" just a little. Because both the workstation and the target are the same architecture, running the same kernel series, 2.6, we can get away with copying the workstation's module utilities into the `Bluecat/` directory, which is visible to the target as `/usr`.

User space processes aren't as sensitive kernel version as kernel modules and indeed kernel version doesn't even factor into application building. The module utilities that came with your Red Hat distribution will execute just fine on the target. The module utilities are normally found in the directory `/sbin`. Copy the following files from `/sbin to $HOME/Bluecat/sbin`: `depmod, insmod, insmod.static, lsmod, modinfo, modprobe, rmmod`.

In the target window on your workstation, `ls –l /dev`. The `parport` device should show up. Now `cat /proc/ioports`. Several I/O port ranges will be listed but you should *not* see one that begins "0378". `cat /proc/devices`. You'll see a number of character devices but you shouldn't see one for major number 6.

`cd` back to the `parport/` directory and `insmod parport.ko`. If the module installed successfully, repeat the `cat` commands for `/proc/ioports` and `/proc/devices`. `parport` should show up in both listings. The driver is now installed and ready to use. But before we can really do anything with it, we need to take a little side trip into the exciting world of hardware.

The Hardware

OK, it's time to dust off the old soldering iron. Yes, I know, this is a book about software and operating systems, but any successful embedded developer needs to get his or her hands dirty on real hardware once in a while.

Figure 7-4 is a schematic of the device we need to build. It's pretty simple really consisting of just two LEDs and three pushbuttons connected to a DB-25P connector, the mate to the parallel port connector on the target PC. The LEDs are connected to the least significant bits of the data port, the pushbuttons to the most significant bits of the status port. Feel free to add more LEDs to display the rest of the data port on pins 4 through 9. There are also two additional status bits available on pins 13 and 15.

The schematic assumes the use of LEDs with built-in dropping resistors so they can be safely connected to a 5-volt source. If you use LEDs without the built-in dropping resistor, you should add a resistor of between 1k and 4.7k in series with each LED.

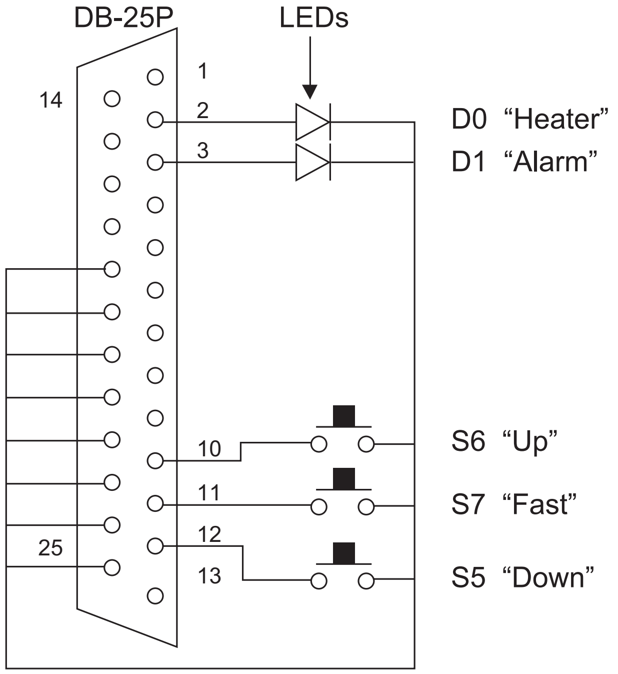

Figure 7-4: Parallel Port Interface

As shown in Figure 7-4, the three pushbuttons represent "Up," "Down," and "Fast". In the context of thermostat, pressing "Up" causes the measured temperature to increase by 2° each time it is read. Pressing "Down" decreases the temperature by two degrees. Pressing "Fast" together with either "Up" or "Down" changes the temperature by 5° per scan.

This is admittedly somewhat awkward but is the result of not having enough digital input bits. If we had, say, 6 bits we could attach a DIP switch to each bit and just read a number directly.

The Target Version of Thermostat

Now we're ready to try building and running thermostat on the target with "real" hardware. Take a look at `trgdrive.c`. It has the same set of functions as `simdrive.c` but these interact with the parallel port through the `parport` device driver. This is a set of "wrapper" functions that translate the raw I/O of the parallel port into something that makes sense to the application. Later on, if we were to plug a real A/D converter into the target, we would simply write a new set of trgdrive functions that talk to its device driver with absolutely no change to thermostat.c.

make target. With the parport module loaded, execute ./thermostat.t (.t for the target version) in the target's shell window. Now play with the pushbuttons and watch the heater and alarm respond accordingly.

Debugging Kernel Code

Kernel Space code presents some problems in terms of debugging. To begin with, DDD and GDB rely on kernel services. If we stop the kernel, at a breakpoint for example, those services aren't available. Consequently some other approaches tend to be more useful for kernel and module code.

printk

Personally I've never been a big fan of the printf debugging strategy. On occasion it's proven useful but more often than not the print statement just isn't in the right place. So you have to move it, rebuild the code and try again. I find it much easier to probe around the code and data with a high quality source level debugger.

Nevertheless, it seems that many Unix/Linux programmers rely heavily on printf and its kernel equivalent printk. At the kernel level this makes good sense. Keep in mind of course that printk exacts a performance hit.

```
#ifdef DEBUG
#define PDEBUG(fmt, args...) printk (<1> fmt, ## args)
#else
#define PDEBUG(fmt, args...)      // nothing
#endif
```

Listing 7-1

While printk statements are useful during development, they should probably be taken out before shipping production code. Of course as soon as you take them out, someone will discover a bug or you'll add a new feature and you'll need them back again. An easy way to manage this problem is to encapsulate the printk's in a macro as illustrated in Listing 7-1.

While you're debugging, define the DEBUG macro on the compiler command line. When you build production code, DEBUG is not defined and the printk statements are compiled out.

/proc Files

We first encountered /proc files back in Chapter 2. The /proc filesystem serves as a window into the kernel and its device drivers. /proc files can provide useful runtime information and can also be a valuable debugging tool. In fact many Linux utilities, lsmod for example, get their information from /proc files. Some device drivers export internal information via /proc files and so can yours.

/proc files are harder to implement than printk statements but the advantage is that you only get the information when you ask for it. Once a /proc file is created, there's virtually no run time overhead until you actually ask for the data. printk statements on the other hand always execute.

A simple read-only /proc file can be implemented with the two function calls shown in Listing 7-2. A module that uses /proc must include <linux/proc_fs.h>. The function create_proc_read_entry() creates a new entry in the /proc directory. This would typically be called from the module initialization function. The arguments are:

- *Name*. The name of the new file.
- *File permissions, mode_t*. Who's allowed to read it. The value 0 is treated as a special case that allows everyone to read the file.
- *struct proc_dir_entry*. Where the file shows up in the /proc hierarchy. A NULL value puts the file in /proc.
- *Read function*. The function that will be called to read the file.
- *Private data*. An argument that will be passed to the read function.

```
#include <linux/proc_fs.h>

struct proc_dir_entry *create_proc_read_entry (char *name,
        mode_t mode, struct proc_dir_entry *base,
        read_proc_t *read_proc, void *data);

int read_proc (char *page, char **start, off_t offset,
        int count, int *eof, void *data);
```

Listing 7-2

The read_proc() function is called as a result of some process invoking read() on the /proc file. Its arguments are:

- *page*. Pointer to a page (4 Kbytes) of memory allocated by the kernel. read_proc() writes its data here.
- *start*. Where in page read_proc() starts writing data. If read_proc() returns less than a page of data you can ignore this and just start writing at the beginning.
- *offset*. This many bytes of page were written by a previous call to read_proc(). If read_proc() returns less than a page of data you can ignore this.
- *count*. The size of page.
- *eof*. If you're writing more than one page, set this nonzero when you're finished. Otherwise you can ignore it.
- *data*. The private data element passed to create_proc_read_entry().

Like any good read function, read_proc() returns the number of bytes read.

As a trivial example, we might want to see how many times the read and write functions in parport are called. Listing 7-3 shows how this might be done with a /proc file.

```
Add to parport.c
#include <linux/proc_fs.h>

int read_count, write_count; // These are incremented each time the
                             // read and write functions are called.

int parport_read_proc (char *page, char **start, off_t offset, int count,
int *eof, void *data)
{
    return sprintf (page, "parport.  Read calls: %d, Write calls: %d\n",
        read_count, write_count);
}

In parport_init ()

    create_proc_read_entry ("parport", 0, NULL, parport_read_proc, NULL);
```

Listing 7-3

Using gdb on the Kernel

Actually it is possible to debug the kernel with GDB. The basic requirement is that you have two systems; a host machine on which GDB runs and a target machine on which the kernel code you're debugging runs. These two machines are connected through a dedicated serial port distinct from the one you're using as the console. This means both the workstation and the target need a second serial port.

Bluecat includes support for kernel debugging with GDB using a kernel patch called kgdb. Debugging support is enabled as part of the configuration process. In the `shell/` directory do `make xconfig` and find the "Kernel hacking" menu down near the bottom of the left hand navigation panel. Enable "Kernel debugging" and "Bluecat kernel debugger." Then select the serial port for the debugger, probably `ttyS1` if `ttyS0` is your console port. Also enable "Include debugging information in kernel binary."

There are a number of additional options to turn on specific debugging features in the kernel. We won't deal with those here since our primary interest is in debugging kernel module code.

Rebuild the kernel. Before copying `shell.kernel` to your `/home` directory, you should rename the `shell.kernel` that's already there. Call it `shell.kernel.norm`, for example. You'll want to keep a nondebug version of the kernel around because the debug version *only* works with GDB.

Copy the new `shell.kernel` to `/home`, reboot your target machine, and boot the shell kernel. In a shell window on the workstation, `cd Bluecat/usr/src/linux` and start GDB specifying the kernel image as a parameter:

```
gdb vmlinux
```

When you get the `(gdb)` prompt, enter:

```
target remote /dev/ttyS1
```

This connects GDB to the target machine. GDB responds with:

```
Remote debugging using /dev/ttyS1
```

The kernel debugger stops the kernel fairly early in the boot up process to create a synchronization point with GDB. You'll see something like:

```
kgdb_breakinst () at arch/i386/kernel/kdbg/i386-stub.c:58
58 }
```

```
warning: shared library handler failed to enable
breakpoint
(gdb)
```

For now, just enter the command continue and watch the kernel boot up. You can break into the target kernel anytime by typing control-C in the GDB window.

Debugging Kernel Modules with GDB

Let's take a crack at running parport under GDB. First of all, we'll have to recompile it with the –g option to create a symbol table for GDB. In your target console window, `cd /usr/demo.x86/apps/parport` and load the module. Now execute:

```
getsyms.sh parport.ko
```

This is a shell script that retrieves information on the specified module from the kernel's symbol table. It will return something like:

```
add-symbol-file parport.ko 0xd00b4060 -s .rodata 0xd00bc9bc -s .bss
0x -s .data 0xd00bf4e0
```

In the GDB window, break into the debugger with control-C and enter the command:

```
cd ../../../demo.x86/apps/parport
```

Now enter the line that was returned by the getsyms.sh script. This loads the symbol file for the parport module and sets appropriate base addresses so that the symbols resolve correctly to memory addresses. We can now set breakpoints in parport and watch it execute.

Building Your Driver into the Kernel

In a production embedded environment there is little reason at all to use loadable kernel modules. So when you have your new driver working you'll probably want to build it into the kernel executable image. This means integrating your driver into the kernel's source tree. It's not as hard as you might think.

To begin with you need to move your driver source files into the kernel source tree. Assuming your source code really is a "device driver," it probably belongs somewhere under usr/src/linux/drivers. In the absence of any definitive documentation or recommendations on this issue, I recommend you create your own directory under usr/src/linux/drivers/char, assuming you're creating some sort of character driver. I called mine usr/src/linux/drivers/char/doug. Copy your driver source code to this directory and create a Makefile like the one shown in Listing 7-4. This is for a driver that consists of a single object file. If your driver contains multiple object files, just add the remainder of the object files as a space delimited list.

```
#
# Makefile for Doug's parport driver.
#
obj-y += parport.o
```

Listing 7-4

The source file requires a couple of minor modifications In the declaration of the module initialization function, add "__init" just before parport_init so that it looks like this:

```
int __init parport_init (void)
```

__init causes the function to be compiled into a special segment of initialization functions that are called as part of the boot up process. Once boot up is complete, this segment can be discarded and the memory reused.

When the module is built into the kernel, the function identified by module_init() is added to a table of initialization functions that are called at boot up. The upshot is that you don't have to modify main.c to add a new initialization function every time you add a new driver. It happens automatically.

Also, we no longer need parport_cleanup() because the device will never be removed. You can bracket that function, and the exit_module() macro with:

```
#ifdef MODULE
#endif
```

Next you'll need to make your device directory visible to the Makefile in the directory just above yours, drivers/char. Open usr/src/linux/drivers/char/Makefile take a look at the section starting around line 76. There are a number of lines of the form:

```
obj-$(CONFIG_yyyy) += yyyy/
```

where CONFIG_yyyy represents an environment variable set by the make xconfig process and yyyy is a subdirectory of char/. All of these environment variables end up with one of three values:

"y" Yes. Build this feature into the kernel.

"n" No. Don't build this feature into the kernel.

"m" Module. Build this feature as a kernel loadable module.

The "/" tells the Makefile that this entry in obj-y (or obj-m) is a subdirectory rather than a file. The build system automatically visits each directory added to obj-y. If you want to add your driver directory unconditionally, add a line like this:

```
obj-y += <your directory>/
```

You can also make your driver a kernel configuration option by defining an environment variable of the form CONFIG_yyyy. Then, instead of the line in the previous paragraph, you would add to the Makefile in /char a line of the form:

```
obj-$(CONFIG_yyyy) += <name>/
```

In your own Makefile, obj-y also changes to obj-$(CONFIG_yyyy).

You also need to add an entry into the configuration menus. Refer back to the section in Chapter 4, *Behind the Scenes—What's really happening.*

When the top level Makefile completes its traverse of the kernel source tree, the environment variable obj-y contains a list of all the object files that should be linked into the kernel image. obj-m is a list of the object files that should be built as kernel loadable modules and obj-n represents those files that aren't needed at all. It's not at all clear to me why the kernel build process needs to know about source files that won't be built.

Try it out. Follow the instructions above and then execute make kernel in demo.x86/shell/. Boot the new kernel either from a diskette or with osloader. You should see parport show up in both /proc/ioports and /proc/devices.

In a true production kernel you would also want to remove loadable module support from the kernel to reduce size.

An Alternative—uCLinux

Much of the complexity and runtime overhead of conventional Linux device drivers is the result of the protected memory environment. In calling a device driver, the system switches from User Space to Kernel Space and back again. Data to be transferred to/from a peripheral device must be copied from/to User Space. All of this chews up run time.

uCLinux is a variant of Linux designed to run on processors without memory management. The uC stands for microcontroller. Without memory management there's no User Space and Kernel Space, no privilege levels. Everything runs in one flat memory space effectively at Privilege Level 0.There's no virtual memory and no swapping.

The lack of privilege levels and memory protection has several implications. First of all, any process can directly execute I/O instructions. This further implies that device drivers aren't really necessary. Of course as a structuring and abstraction tool drivers are still useful, but now they can be as simple as functions linked directly to a process executable image.

The other major consequence is that any process can bring down the whole system, just like kernel loadable modules in conventional Linux.

uClinux has been ported to a number of Motorola microcontrollers including the Dragonball (M68EZ328) and other 68k derivatives as well as Coldfire. Other ports include the Intel i960, ARM7TDMI and NEC V850E. For more information on uClinux, go to *www.uclinux.org*.

The 2.6 series kernels fold uClinux into the main kernel tree as an option when the appropriate target architecture is selected.

Arcturus Networks offers a number of hardware development kits based on uClinux. Check them out at *www.arcturusnetworks.com*.

The "Old Way"

There were substantial changes to the device driver model between version 2.4 and version 2.6 of the kernel. The earlier driver mechanisms have by and large been deprecated[3], but are nevertheless worth a quick look because they're still used in many

[3] I never liked that word "deprecate" until I recently looked it up on wikipedia (*www.wikipedia.org*). In the context of computer software, it signifies the "gradual phasing-out of a software or programming language feature." The term alerts users that while the feature still works in the current software version, perhaps with warning messages, it is subject to removal at any time. The term derives from the Latin verb deprecare, meaning "to ward off (a disaster) by prayer." Hmmm, very appropriate.

existing drivers that haven't been upgraded. The older protocols are still supported under 2.6, but any new drivers should use the newer mechanisms.

Device Registration

```
int check_region (int base, int nports);
int register_chrdev (int major, char *name, file_operations *fops);
int register_blkdev (int major, char *name, file_operations *fops);
```

The 2.4 series used the functions register_chrdev() and register_blkdev() to register character and block devices respectively. There was no cdev structure to allocate and register. The other interesting feature of 2.4 was the check_region() function, which validated the availability of a range of I/O ports before trying to allocate it with request_region(). There's an obvious problem with this approach that we'll look at in more detail in Chapter 9 when we discuss real-time issues.

Briefly, what happens if two drivers "simultaneously" try to allocate the same range of I/O ports? They each get an affirmative response from check_region() that the I/O range is available. Each then calls request_region() thinking that it has exclusive access. But in reality, only one of them does. In 2.6 the functionality of check_region() was folded into request_region() so that the check is done "atomically" with the allocation. Only one driver will get an affirmative response from request_region().

Module Initialization

The other major difference is that the 2.4 and earlier kernels didn't have the module_init() and module_exit() macros. Instead, each module had a pair of functions named init_module() and cleanup_module() that handled initialization and cleanup respectively. Clearly these function names were not exported but were instead handled as statics by the insmod and rmmod commands.

Resources

The subject of module and device driver programming is way more extensive than we've been able to cover here. Hopefully, this introduction has piqued your interest and you'll want to pursue the topic further. An excellent book on the topic is:

Rubini, Alessandro, Jonathan Corbet, and Greg Kroah-Hartman, *Linux Device Drivers, 3nd Ed.*, O'Reilly, 2005.

In fact, I would go so far as to say this is one of the best computer science books I've read. It's very readable and quite thorough.

For more information on using kgdb for kernel-level debugging, visit *http://kgdb.linsyssoft.com/*.