Device Drivers Demystified

They really aren't all that mysterious

Class ESC-117
Monday, March26, 1:00 pm

Doug Abbott
Silver City, NM
doug@intellimetrix.us

Intellimetrix
COMPUTING FOR SCIENCE AND INDUSTRY

www.intellimetrix.us

Abstract

Application programmers often view device drivers as some sort of "black magic". All that bowing and scraping to the operating system! The objective of this class is to make the case that drivers really aren't all that mysterious. At its most fundamental, a device driver is nothing more than a set of functions that presents a uniform, high-level API to the application programmer that abstracts out the messy details of the hardware. The apparent complexity of drivers comes from the hoops one has to jump through to link the driver with the application.

Introduction

So what the heck is a device driver anyway? There are probably as many definitions of a *device driver* as there are programmers who write them. In simple systems a driver may be nothing more than a library of functions linked either statically or dynamically to the application. In the context of general purpose operating systems, device drivers are often independently loaded programs that communicate with applications through some OS-specific protocol.

But the whole point of a device driver is to hide the messy details of accessing a peripheral device behind some sort of higher level abstraction that makes sense to an application programmer who shouldn't be concerned with where the serial port data register is located or what bit indicates that a character has been received.

Abstract Device Driver Model

Let's start by developing some sort of abstract model of a device driver in terms of APIs. What's the most fundamental thing a driver does? Well, it transfers data. So we'll need functions to read (get data from the device) and write (send data to the device). These functions might look like this:

```
int read (void *buffer, int length, int *actual);
int write (void *buffer, int length, int *actual);
```

buffer is an area of length bytes in memory to which or from which data is transferred. The functions store the number of bytes transferred in actual. The functions' return value is a status—did the operation succeed or not.

So far, so good, but there's something important missing here. We haven't specified the device we're talking to. There are a couple of ways we might do that. We could create unique pairs of read and write functions for each device. So, for example, we might have:

```
read_serial ()      write_serial ()
read_AD ()          write_AD ()
read_display ()     write_display ()
```

I suggest this has a certain "inelegance" to it. Every time we add a new type of device, we need new functions. But it's even worse than that because every time we encounter a new implementation of a given device type, we need a new pair of read and write functions. This could lead to something like:

```
read_serial_8250 ()     write_serial_8250 ()
read_serial_68328 ()    write_serial_68328 ()
read_serial_atmel ()    write_serial_atmel ()
```

We're starting to develop quite a library here. There's also the issue that you might have more than one of any peripheral device. You could very well have multiple serial ports for example. How do you distinguish them?

Perhaps it would be preferable to identify the device with a function parameter like this:

```
int read (device_t *dev, void *buffer, int length, int *actual);
int write (device_t *dev, void *buffer, int length, int *actual);
```

where device_t is a typedef struct that characterizes the device. Now we're back to having only one pair of read and write functions. The device_t structure could very well contain pointers to the device-specific read and write functions. The application calls a generic read (), which turns around and calls the device-specific read () specified by the device_t.

The next problem is, how do we make the connection between the application and the driver? That is, how do we fill out the device_t structure? We could just directly set the fields of dev, such as:

```
dev->read = read_serial_8250;
dev->write = write_serial_8250;
```

I would suggest that this requires too much detailed knowledge on the part of the application programmer who really doesn't care what serial device is present, she just knows it's a serial device, or maybe she just knows that it's a source of command data. Maybe the solution is to let the driver itself fill in the dev structure.

How about if we do this? We'll invent a mechanism that ties a device to an ASCII string. We could arrange for each device driver to have an initialization function called at boot time that, among other things, would call a system function that connects a string with, say, a filled in device_t structure. It might look something like this:

```
int device_init ()
{
        device_t dev;
        .
        .
        /* fill in the dev struct */
        .
        .
        device_register ("serial", &dev);
        return 0;                       // success
}
```

device_register() would have to copy the dev structure into some global table. An application that needs to access the serial device would do something like this:

```
device_t serial_dev;
char command[80];
        .
        .
        .
status = connect ("serial", &serial_dev);
        .
        .
status = read (&serial_dev, command, sizeof (command), &actual);
```

We've reduced our ever-expanding library of access functions back down to a basic set that has the ability to access any device with virtually no specific device knowledge on the part of the application programmer.

So far we've concerned ourselves only with reading and writing data. Typically that's the primary focus of accessing a peripheral device. But that's not necessarily the full story. Many, if not most, peripheral devices have various control mechanisms that don't fit neatly in the stream of bytes model that we've developed for transferring data. For example, a serial port has a baud rate; an A to D converter may have a programmable range. We need some other mechanism for dealing with these peculiarities. This is often called an "out of band" mechanism.

Let's invent a *control* function for our devices. Its prototype might look something like this;

```
int control (device_t *device, int command, …);
```

where the variable length argument list represents optional parameters for the command. It would be difficult to anticipate all of the possible commands and their parameters, so we should probably just leave that up to the driver writer.

At this point, our collection of driver APIs consists of the following functions:

```
int connect (char *name, device_t *dev);
int read (device_t *dev, void *buffer, int len, int *transferred);
int write (device_t *dev, void *buffer, int len, int *transferred);
int control (device_t *dev, int command, …);
```

These four functions provide just about all the functionality we need. We might want to add a disconnect function to release the connection to a device when we're done with it. That allows any resources that the connection is using to be recovered.

Blocking I/O

Let's consider another problem. What if there's no data available to read from the device or it's not able to accept write data. The read and write functions could simply return a 0 transfer length, but that's generally not a useful thing to do. Consider reading a disk drive for example. You typically have to wait for the head to move to the right track and for the right sector to come around. Except in the most simple systems, you don't want the program "twiddling its thumbs" all that time. You want it to be doing something useful while it waits to satisfy the read request.

That's why multi-tasking operating systems were invented. The read function reaches a point where it has to wait for some condition to be satisfied before it can fulfill the user's read request. We want the task that called read to politely "go to sleep" until the read operation can be completed. This allows another task to do something useful in the meantime.

Generally, the condition indicating that the operation is ready to proceed is signaled by an interrupt. So it's the interrupt service routine (ISR) that "wakes up" the task. Note incidentally that these operations are internal to the device driver and aren't seen by the application programmer. In effect, the application programmer doesn't have to deal with interrupts.

So we'll need a pair of functions called perhaps sleep() and wake_up() that might look like this:

```
int sleep (wait_t *wait);
int wake_up (wait_t *wait);
```

The purpose of the wait_t structure is to differentiate among various conditions that multiple drivers may be waiting on. Realize that as soon as we do this, the driver functions must be reentrant. They're subject to being called by multiple tasks simultaneously. That in turn means that any statically allocated resources must be protected by synchronizing mechanisms such as mutexes.

The Linux Model

Everything is a file

Linux tries to treat everything as a file. And while other operating systems at least pretend to treat peripheral devices as files, Linux goes the final step and actually creates nodes in the file system to represent peripherals. Actually this isn't entirely true. Network devices have a different interface to the kernel and are not represented by nodes in the file system.

By convention, file system nodes that represent peripherals are placed in the /dev directory. File system nodes that represent devices are characterized by two numbers; a *major* device number and a *minor* device number. The major number serves to associate a file system node with a device driver. The minor number is simply passed to the driver for its use.

Devices come in four "flavors": Character, Block, Pipe, and 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. With character devices the stream typically moves in one direction only. Block devices must have file systems associated with them whereas character devices don't. In both cases, I/O data is viewed as a "stream" of bytes.

Pipes are pseudo-devices that establish uni-directional links between processes. One process writes into one end of the pipe and another process reads from the other end.

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. This necessitates a different interface between the kernel and the device driver. Network devices are not nodes in the /dev directory.

The Driver APIs

The application's view of I/O in Linux is fairly similar to the model we developed above. The APIs are:

```
int open (char *path, int flags);
size_t read (int file_des, void *buffer, size_t length);
size_t write (int file_des, void *buffer, size_t length);
int close (int file_des);
int ioctl (int file_des, int cmd, …);
```

open() serves the same function as connect(). A positive return value is a "file descriptor" that identifies the connection to all the other functions. An error is indicated by a return value of -1. flags identifies the type of access requested. A positive return value from read() and write() is the number of bytes transferred. close() is the equivalent of disconnect() and ioctl() is the same as control(). There are other I/O functions, but they're used much less frequently.

Kernel Loadable Modules

There are a couple of different ways that device drivers can be installed, or integrated with the Linux kernel. A driver can be compiled and linked directly into the kernel image. It becomes a part of the kernel. Drivers for the most common peripherals and those required to boot the system are built this way.

But it would be virtually impossible, not to mention relatively pointless, to build drivers for all possible devices into the kernel. So most device drivers are in fact built as *installable kernel modules* to be dynamically integrated into the kernel when their functionality is required.

Fundamentally, installable kernel modules offer a way to extend the functionality of the kernel without having to rebuild it. When a module's functionality is required, it is dynamically "installed", or integrated into the kernel. Later, if its functionality is no longer required, the module can be removed from the kernel.

Being part of the kernel, modules run at Privilege Level 0 and are thus capable of bringing down the entire system.

A module is *installed* into the kernel by executing the insmod command. When the module's functionality is no longer needed, it can be removed, and its memory space reclaimed, by executing rmmod. Note that insmod allows parameters to be passed to the module. These parameters can be strings or integers of various sizes.

The basic requirement for a kernel module (other than being thoroughly debugged) is that it include at least two functions, one of which is called as part of insmod to initialize the module and the other is called as part of rmmod to "clean up" when the module is removed.

The initialization function for a device driver module typically does the following:

- Initialize the device
- Allocate resources such as dynamic memory
- Register itself with the kernel. This creates the connection that allows the kernel, and ultimately user space processes, to use the functionality of the module.

The cleanup function just reverses what the initialization function did. It de-registers the module, returns any allocated resources and perhaps quiesces the device.

A loaded module can make use of services provided by the kernel because it has access to the kernel's symbol table. Furthermore, symbols exported by the module are added to the symbol table for possible use by subsequently loaded modules.

Modules are kernel version specific. That is, they are built to run with a specific kernel version. insmod will object if you try to load a module in a different kernel than the one it was built for. This is because kernel APIs change over time. A module built for one kernel version may fail under a later version because some APIs are different or just plain missing. Note that you can force a module to load in a different kernel but it's not recommended.

Char driver basics

The very first thing a driver needs to do is allocate one or more device numbers for the devices it manages. This would usually be done in the module init function. Device numbers in the 2.6 series kernels are 32 bits—the high 12 bits represent the major number and the low 20 bits the minor number. This is a substantial expansion over previous kernel versions where both major and minor numbers were limited to 8 bits. Of course, good programming practice says you should never rely on a specific bit allocation of something like a device number, so the kernel provides a set of macros for creating and dissecting device numbers.

There are two ways to get a range of device numbers for your driver. If you need a specific range of numbers, you use the function register_chrdev_region(), which may return an error if the requested region is not available. The preferred mechanism is to ask the kernel to dynamically allocate the required range using the function alloc_chrdev_region().

We have one or more device numbers, but we don't necessarily have nodes in the /dev directory to represent them. If we allocated the device numbers dynamically, we won't know until run time what the major number is. There's another, newer technique for getting device numbers that also automatically creates the /dev nodes. Create a "miscellaneous class" device by calling misc_register(), passing in a stuct miscdevice that includes a minor number, a name, and a file operations structure. Miscellaneous devices all use major device number 10.

With a device number in hand, we can now connect our driver to the kernel. This requires a cdev structure, which can be allocated dynamically through the function cdev_alloc(), or statically, often as part of some device-specific structure.

Before we can add the driver to the kernel, there are two fields that must be initialized. The ops field must point to a file operations structure, and the owner field is normally set to THIS_MODULE.

The function cdev_add() makes the driver known to the kernel. If it succeeds (it almost always does), the driver is "alive" and ready to be accessed.

The most important field in the cdev data structure is the *File Operations Table*, struct file_operations. With the exception of the first field, it is a list of pointers to functions that implement the driver APIs. There is a very large number of functions in this table, many of which are only rarely used. For the most part, there is a one-to-one correspondence between these kernel space functions and a set of user space functions.

Tools

Let's look at a couple of tools that can be of immense help to driver writers. The printk() function is the kernel's equivalent of printf() and is used extensively by kernel and driver developers for debugging. printk() does not write directly to the console terminal but rather writes to a circular buffer in memory and then wakes up any process waiting on the *syslog* system call.

To make a long story short, the process waiting on the syslog call is klogd and its default behavior is to write anything in the circular buffer to the log file /var/log/messages and, if the *loglevel* is of sufficient priority, meaning a low enough number, print it to the current console.

Incidentally, one consequence of this strategy is that printk() is non-blocking and thus can be invoked from within interrupt service routines whereas most I/O operations can't.

The /proc file system, that is the files in the /proc directory, is a valuable tool for gaining visibility into what's happening in kernel code. It acts just like an ordinary file system. You can list the files in the /proc directory, you can read and write the files, but they don't really exist. The information in a /proc file is generated on the fly when the file is read. The kernel module that registered a given /proc file contains the functions that generate read data and accept write data.

/proc files provide dynamic information about the state of a kernel module in a way that is easily accessible to user-level tasks and the shell. The first thing to notice when you look at the /proc directory is the large number of subdirectories named with numbers. Each running process gets a directory under /proc named for its process ID. Under this are several directories and files describing the state of the process.

/proc files can be a useful mechanism for gaining visibility into what's happening in your device driver. Unlike printk(), which always prints if it's compiled in, a /proc file only returns information when you ask for it.

In its simplest form, a /proc file is nothing more than a read method that is registered as a /proc file. When a User space process reads the file, the registered method is called and passed a page of memory (4k bytes on an x86) to return its data.

System Resources

Device drivers need access to certain system resources to carry out their task. Among these are: device numbers, I/O ports and interrupts. These resources must be allocated to the driver on an exclusive basis by calling the corresponding request or register function. We've already seen how that works with the device number dev_t. In virtually all cases, one of the arguments passed to the registration function is a "name" string that identifies who owns the resource. This then becomes an entry in a /proc file.

Consider registering an interrupt handler. The function is request_irq() and its arguments are:

- The interrupt number you're registering
- A pointer to the handler function, which takes two arguments to be described shortly
- A flags bit mask
- The device name. This is used by /proc/interrupts to identify the devices holding interrupts
- A pointer used for shared interrupt lines. Typically this would point to the driver's private data area as a way of identifying which device caused the interrupt. If the interrupt isn't shared, this can be NULL.

As usual, the return value is 0 if the function succeeds and a negative error code if it fails. The most likely error is –EBUSY, meaning that another device has registered this non-shared interrupt.

The flags argument has several bits including these three:

- IRQF_DISABLED—indicates a "fast" handler when set. A fast handler executes with interrupts disabled. This bit should not be set unless it's absolutely necessary.
- IRQF_SHARED—when set, indicates that the interrupt line is shared. Multiple registrations of a shared interrupt succeed.
- IRQF_SAMPLE_RANDOM-- indicates that the generated interrupts can contribute to the "entropy pool" used by /dev/random and /dev/urandom for generating truly random numbers.

There are fundamentally two approaches that operating systems use to handle interrupts, which I choose to call the "direct" approach and the "indirect" approach. All contemporary processor architectures have some kind of interrupt vector table with pointers to interrupt service routines. When an interrupt occurs, its number serves as an index into this table to invoke the corresponding ISR.

In the direct approach, which tends to be used with smaller, simpler kernels and systems with no OS, the entries in the vector table point directly at the ISRs. The ISR is thus responsible for saving the processor state on entry and restoring the state prior to exiting.

In the indirect approach, the vector table points to someplace in the OS that takes care of saving the state and possibly some other housekeeping, and then calls the ISR (called an *interrupt handler* in this case) as an ordinary func-

tion. When the handler returns, the OS takes care of restoring the processor state and returning to the background code. Linux uses the indirect approach.

Blocking I/O in Linux

Earlier we considered what happens if the driver's read or write functions are invoked and data transfer is not possible. The usual behavior is to block the calling process until transfer is possible. Linux implements this behavior with a mechanism called a *wait queue*.

In the case of Linux, if two or more processes happen to be waiting on the same queue, they *all* wake up. One of them will get to run first, as determined by the scheduler, and will find the condition it's waiting for to be true. When that first process is finished, the condition is probably no longer true and the processes have to wait again. The reason for doing it this way is to let the scheduler determine the highest priority ready process and run it first.

I/O Ports

So far in our discussion of Linux device drivers we've done a lot of bowing and scraping to the operating system. Now it's time to actually diddle some hardware bits. But before we can do that, there are some kernel and compiler issues to be considered.

The first issue is that different processors treat I/O differently. Some, like the x86, put I/O registers in a totally different address space accessed by unique instructions. Others simply reserve a part of the memory space for I/O ports. In this case, I/O access is no different from memory access.

Unlike memory, I/O access usually has "side effects." That is to say that the act of reading or writing an I/O port often does something other than simply transfer data. Memory access has no side effects. The compiler takes advantage of that to implement optimizations that may change the order in which events occur, or even eliminate them altogether. We must be very careful about compiler optimizations when it comes to I/O port access.

And of course I/O ports are allocated exclusively to one driver at a time. Before your driver can use a range of I/O ports, it must ask the kernel's permission.

Summary

In our brief tour of Linux device drivers, we've learned the following:

- Everything is a file – almost
- Kernel loadable modules are a way to dynamically extend the kernel at run time
- The basic structure of a character driver and its most important object, the File Operations Table
- Tools that aid in developing drivers
- What resources a driver needs and how they are allocated
- Considerations when accessing hardware registers

Fundamentally, a device driver is just a set of functions that are called on behalf of, and in the context of, a user space process. What tends to make a driver complicated is that it is usually an independently loaded program and so there must exist some mechanism for linking that program to the rest of the system. Also, in the case of Linux and other protected mode operating systems, there is the process of transitioning between user space and kernel space.

Resources

http://lxr.linux.no/ -- This is one of the neatest Linux sites I've come across. It is the Linux kernel source code fully cross-referenced and hyperlinked. You can search for any identifier in the source and see where it's defined and where it's used.

http://kernelnewbies.org – Lots of resources for those new to the wonderful world of kernel programming

Rubini, Alessandro, Jonathan Corbet, and Greg Kroah-Hartman, *Linux Device Drivers, 3nd Ed.*, O'Reilly, 2005. This is a very thorough and readable treatment of device drivers and programming at the level of kernel APIs, although it is getting a bit dated.

Venkateswaran, Sreekrishnan, *Essential Linux Device Drivers*, Prentice Hall, 2008. Larger and more up to date than Rubini, there's a lot here, but I find the organization a bit odd.