Chapter 1

What is a Device Driver?

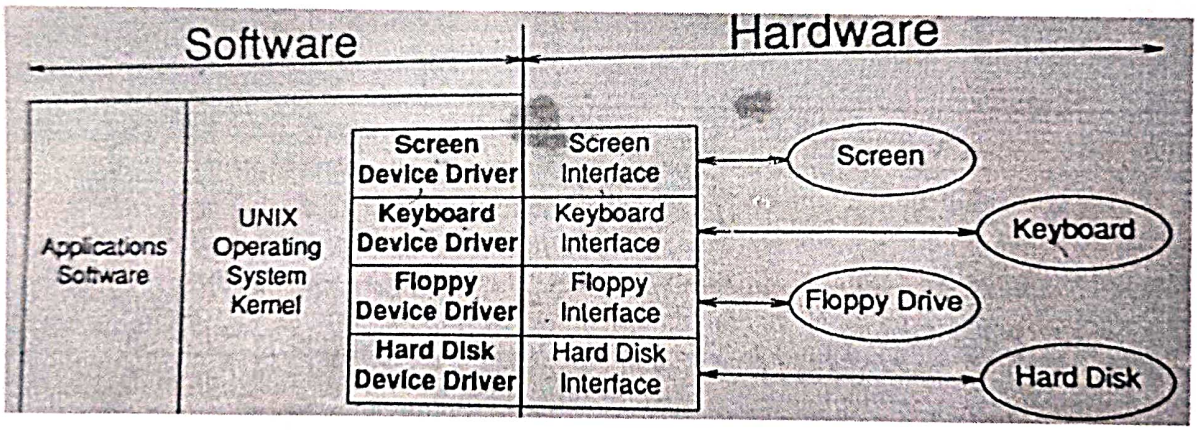
The Grand Design

(A device driver is the glue between an operating system and its I/O devices. Device drivers act as translators, converting the generic requests received from the operating system into commands that specific peripheral controllers can understand.

This relationship is illustrated in Figure 1-1. The applications software makes system calls to the operating system requesting services (for exam- ple, write this data to file x).

The operating system analyzes these requests and, when necessary, issues requests to the appropriate device driver (for example, write this data to disk 2, block 23654).

The device driver in turn analyzes the request from the operating system and, when necessary, issues commands to the hardware interface to perform the operations needed to service the request (for example: transfer 1024 bytes starting at address 235400 to disk 2, cylinder 173, head 7, sector 8).



Although this process may seem unnecessarily complex, the existence of device drivers actually simplifies the operating system considerably. Without device drivers, the operating system would be responsible for talking directly to the hardware. This would require the operating system's designer to include support for all of the devices that users might want to connect to their computer. It would also mean that adding sup- port for a new device would mean modifying the operating system itself.

By separating the device driver's functions from the operating system itself, the designer of the operating system can concern himself with issues that relate to the operation of the system as a whole. Furthermore, details related to individual hardware devices can be ignored and generic requests for I/O operations can be issued by the operating system to the device driver.

The device driver writer, on the other hand, does not have to worry about the many issues related to general I/O management, as these are handled by the operating system. All the device driver writer has to do is to take detailed device-independent requests from the operating system and to manipul..te the hardware in order to fulfil the request.

Finally, the operating system can be written without any knowledge of the specific devices that will be connected. And the device driver writer can connect any I/O device to the system without having to modify the operating system. The operating system views all hard through the same interface, and the device driver writer can connect almost any type of disk to the system by providing a driver so that the operating system is happy.

The result is a clean separation of responsibilities and the ability to add device drivers for new devices without changing the operating system. Device drivers provide the operating system with a standard interface to non-standard I/O devices.

The Major Design Issues

To make studying the design of device drivers easier, we have divided the issues to be considered into three broad categories:

1. **Operating System/Driver Communications**

2. **Driver/Hardware Communications**

3. **Driver Operations**

The first category covers all of the issues related to the exchange of information (commands and data) between the device driver and the operating system. It also includes support functions that the kernel provides for the benefit of the device driver.

The second covers those issues related to the exchange of information (commands and data) between the device driver and the device it controls) (i.e., the hardware). This also includes issues such as how software talks to hardware-and how the hardware talks back.

The third covers issues related to the actual internal operation of the driver itself. This includes:

- interpreting commands received from the operating system;
- scheduling multiple outstanding requests for service;
- managing the transfer of data across both interfaces (operating system and hardware);
- accepting and processing hardware interrupts; and maintaining the integrity of the driver's and the kernel's data structures.

When we proceed to study the actual drivers presented in this book, we shall examine each in the context of these three general design issues.

Types of Device Drivers

UNIX device drivers can be divided into four different types based entirely on differences in the way they communicate with the UNIX oper- ating system. The types are: block, character, terminal, and STREAMS. The kernel data structures that are accessed and the entry points that the driver can provide vary between the various types of drivers. These differ- ences affect the type of devices that can be supported with each interface.

The following sections will describe these differences briefly and the types of devices that each driver can support

Block Drivers

(Block drivers communicate with the operating system through a collec tion of fixed-sized buffers as illustrated in Figure 1-2. The operating sys- tem manages a cache of these buffers and attempts to satisfy user requests for data by accessing buffers in the cache. The driver is invoked only when the requested data is not in the cache, or when buffers in the cache have been changed and must be written out.

Because of this buffer cache the driver is insulated from many of the details of the users' requests and need only handle requests from the operating system to fill or empty fixed-sized buffers.

Block drivers are used primarily to support devices that can contain file systems (such as hard disks))

Character Drivers

Character drivers can handle I/O requests of arbitrary size and can be used to support almost any type of device Usually, character drivers are used for devices that either deal with data a byte at a time (such as line printers) or work best with data in chunks smaller or larger than the standard fixed-sized buffers used by block drivers (such as analog-to-digi- tal converters or tape drives).

One of the major differences between block and character drivers is that while user processes interact with block drivers only indirectly through the buffer cache, their relationship with character drivers is very direct. This is shown schematically in Figure 1-3. The I/O request is passed essentially unchanged to the driver to process and the character driver is responsible for transferring the data directly to and from the user process's memory.

The Gross Anatomy of a Device Driver

Each of the following chapters will present a complete driver for study. But before we turn the page and start considering our first driver, it might be helpful to consider the major components of a driver.

Recall that a driver is a set of entry points (routines) that can be called by the operating system. A driver can also contain: data structures private to the driver, references to kernel data structures external to the driver; and routines private to the driver (i.e., not entry points).

Most device drivers are written as a single source file. The initial part of the driver is sometimes called the prologue. The prologue is everything before the first routine and like most C programs contains:

- #include directives referencing header files which define various kernel data types and structures;

- #define directives that provide mnemonic names for various con- stants used in the driver (in particular constants related to the location and definition of the hardware registers); and

- declarations of variables and data structures.

The remaining parts of the driver are the entry points (C functions referenced by the operating system) and routines (C functions private to the driver.)

When we examine our drivers we shall consider each of these parts in turn.

General Programming Considerations

As we shall see in the many examples in this book, writing device drivers is somewhat different from writing applications programs in C. While explanation of the subtle differences can wait until we encounter them, it will be helpful to discuss the significant differences now.

As obvious as it sounds, the main difference is that device drivers are part of the kernel and not normal user processes. This means that many of the things that a normal C program can do a driver cannot. For starters, for- get about using any of the normal C library functions described in the programmer's reference for your UNIX system. These functions, normally documented in sections 2 and 3 of your manual (or section S if you have an SCO XENIX or UNIX operating system) are not supported by the ker- nel. The functions that are supported by your kernel may be found in the device driver manual for your system. Note that although some of the ker- nel routines have the same name as standard C library functions (e.g.. printf) they are somewhat different.

In addition, make frugal use of the stack (do not use recursive functions and do not declare local arrays). The stack space available to the kernel is limited and is not expandable on most UNIX systems. Also, do not use floating-point arithmetic. Although your machine may have a floating- point unit, the kernel does not save the contents of the FPU's registers unless it is planning to switch processes. Therefore using the FPU at best can cause incorrect results both in your code and some poor innocent user's program and at worst can cause the system to crash.

Do not busy wait (spin) within your driver waiting for an event to occur unless the expected time to wait is less than the time to leave your driver and re-enter it when an interrupt occurs (ie, less than 200 microseconds on a fast 386). User processes that spin will merely have the CPU taken away from them and given to other processes in turn. The kernel, how- ever, always takes priority over any user process and a driver that is exe- cutting a spin loop will prevent the system from doing anything but responding to interrupts.

Character Driver I: A Test Data Generator

Instead of spending countless chapters droning on about the UNIX oper- ating system and the theory of drivers, let's look at a real driver right now!

The driver we are about to examine is actually a pseudo-device driver in that it does not control any hardware. This will allow us to introduce the basic concepts of a UNIX device driver without the complexity of having to deal with a real device.

Read system calls on this pseudo-device will return the "Quick Brown Fox" message repeated infinitely. Writes are just ignored.

This device driver can therefore be used to generate infinite amounts of data for testing without consuming any disk space. For example, to test a terminal, one could type:

```
cat /dev/testdata
to test a printer)
cat /dev/testdata > /dev/1p
or to generate a one-megabyte data file:
dd if=/dev/testdata of bigfile bs-lk count=1024
```

In the first two cases, the test would continue until the process was stopped by typing the interrupt character on your terminal (DEL, C, or whatever you have it set to).

Since this driver will handle data byte by byte and reads of arbitrary size, it is best structured as a character driver.

The Design Issues

Since this driver does not control any hardware, we need only concern ourselves with two of the three categories of design issues: the UNIX/driver interface and the internal driver operation. There are no driver/hardware interface issues./

The Operating System/Driver Interface

How does the UNIX kernel tell the driver what it wants it to do?

As we shall see shortly, this is done in two steps. Firstly, the operating system calls one of the device driver's entry points (functions). This causes control to pass to the device driver.

Secondly, the device driver examines the parameters passed and kernel data structures for information on exactly what to do.

Each of the four types of device drivers has its own set of entry points that the operating system expects to find and its own conventions for the exchange of data and commands. A character driver may have any or all of the following entry points:

init()

The init entry point is called by the kernel immediately after the system is booted. It provides the driver with an opportunity to initialize the

driver and the hardware as well as to display messages announcing the presence of the driver and hardware/

start()

The start entry point is called by the kernel late in the bootstrap sequence when more system services are available. It provides the driver with an opportunity to perform initialization that require more system services than are available at the time when init is called.

open(dev, flag, id)

The open entry point is called by the kernel whenever a user process per forms an open system call on a special file that is related to the driver. It provides the driver with an opportunity to perform initialization that need to occur prior to handling read and write system calls.

close(dev, flag, id)

The close entry point is called by the kernel when the last user process that has the driver open performs a close system call (or exits, causing the operating system to close all open files automatically). Note that the driver's open entry point is called for every user open, while the driver's close entry point is called only for the last user close. It provides the driver with an opportunity to release resources that may be needed only while the device is open and to reset the device or otherwise place it in a quiescent mode.

halt()

The halt entry point is called by the kernel just before the system is shut down. It provides the driver with an opportunity to prepare the hardware for the shutdown and to flush any data that may still be resident in the driver.

intr(vector)

The intr entry point is called by the kernel whenever an interrupt is received from the hardware, Interrupts are a signal from the hardware that a significant event has occurred (usually the completion of the last I/O operation) which requires the attention of the driver.

read(dev)

The read entry point is called by the kernel whenever a user process performs a read system call on a special file that is related to the driver. The driver is required to accept the details of the I/O request, perform the nec essary operations to obtain the requested data, and arrange for the trans- fer of the data to the user's process.

write(dev)

The write entry point is called by the kernel whenever a user process performs a write system call on a special file that is related to the driver. The driver is required to accept the details of the I/O request, arrange for the transfer of the data from the user's process to the driver (or directly to the device), and perform the necessary operations to write the data to the device.

poll(pri)

The poll entry point is called by the kernel 25 to 100 times a second (depending on the version of UNIX). It provides the driver with an oppor- tunity to perform operations on a periodic basis or to check on the status of the device on a regular basis.

ioctl(dev, cmd, arg, mode)

The ioctl entry point is called by the kernel whenever a user process performs an ioctl system call on a special file that is related to the driver. The driver is required either to accept the details of the ioctl request and per- form

the necessary operations, or reject the request with an error. Ioctl calls are used to pass special requests to the driver or to obtain informa- tion on the configuration or status of the device and driver.

The actual names of a particular driver's entry points are the name of the entry point (as above) plus a two- to four-letter prefix. For example, the prefix we have chosen for this sample driver is chr1 (character driver #1). Therefore the init entry point is named chrlinit and the read entry point chrIread. Each of these entry points will be discussed in detail as we encounter them in the drivers we are about to study.

The Driver Prologue

The prologue of a device driver includes all of the definitions and declara- tions required by the rest of the driver. In particular, most prologues con. sist of three parts: references to the header files necessary to define various kernel data types and structures; definitions that are local to the driver; and declarations local to the driver.

The prologue for this driver contains the references to the header files and local declarations. It has no local definitions.

```
1 #include <sys/types.h>
2 #include <sys/param.h>
3 #include <sys/dir.h>
4 #include <sys/signal.h>
5 #include <sys/user.h>
6 #include <sys/errno.h>
7 static char foxmessage[] =
*THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG\n";
```

The first six lines include header files that define various data types and structures that are used within the operating system and/or driver. These header files are found in the directory /usr/include/sys and ought to be read before writing your first driver. Briefly:

▪sys/types.h
contains the definitions of various data types that are used by other include files and by the driver itself (i.e., dev_t);

▪sys/param.h
contains the definitions of various kernel parameters and macros that are needed by sys/user.h to size certain arrays;

sys/dir.h
contains the definitions of the directory structure needed by sys/user.h;
Chapter 2: Character Driver I: A Test Data Generato

sys/signal.h
contains the definitions related to signals needed by sys/user.h;

sys/user.h
contains the definition of the user structure; and

sys/errno.h

contains the definitions of the various error codes that may be returned from a driver. The last line of our prologue declares the character array that contains our "QUICK BROWN FOX..." message. Note that variables used only within the driver are declared static so as not to risk conflict with global variables defined in the kernel.

The remainder of the driver consists of the driver's entry points.

The Init Entry Point

This entry point is called by the operating system shortly after the system. is booted. Few system services are available to the driver at this time (i.e., it must not attempt to read a disk file). The primary purpose of calling the init routine is for the device driver:

to check that the device is actually installed on the machine;

- to print a message indicating the presence (or absence) of the device and driver; to initialize the device (if necessary) prior to the first open; and

- to initialize the driver and allocate local memory (if necessary) prior to the first open.

Since the system itself is not completely initialized it is important not to use certain system services that are not yet ready. In particular:

do not call the sleep, delay, or wakeup kernel support routines; and

do not attempt to reference any members of the user structure (the u. area).

Note also that interrupts are not available at init time.

The init entry point is optional and need not be included in a driver, although most drivers, even those that do not need to perform any initial-ization, provide one to announce their presence. Such is the case with the init entry point for this driver.

The Read Entry Point

The read entry point of a character driver is called when the user's pro- cess has requested data from the device using the read system call. This routine must coordinate the transfer from the driver to the user's process.

Although the read entry point is optional, it is necessary for any driver that wishes to transfer data from a device (or driver) to a process (i.e., sat- isfy a read system call). Since our driver's sole purpose is to transfer "QUICK BROWN FOX..." messages from the driver to a process, it has a read entry point.
The pseudo-code for this entry point is below, followed immediately by the actual driver code. This sequence will be used in this book whenever the entry point is sufficiently complex to warrant a pseudo-code overview. In this manner readers wishing an overview of the driver can skim the pseudo-code and avoid the details of the actual driver. Since the line num- bers in the pseudo-code listing refer to the lines in the actual driver code, the commentary on the code can be followed by referring to either the pseudo-code or the actual code.