3

An Overview Of The Hardware And Runtime Environment

One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man.

— Elbert Hubbard

3.1 Introduction

Because it deals with the details of devices, processors, and memory, an operating system cannot be designed without knowledge of the capabilities and features of the underlying hardware. Throughout the text, we will use two example hardware platforms: a *Galileo* and a *BeagleBone Black*. The platforms each consist of a small, low-cost circuit board that includes a processor, memory, and a few I/O devices. The processors on the boards employ well-known instruction sets: the Galileo uses Intel's instruction set that is known as *x86*, and the BeagleBone Black uses an *ARM* instruction set. We will see that most operating system functions are identical on the two architectures. However, the use of two platforms will allow a reader to compare how low-level OS system functions, such as a context switch, differ between a CISC (Complex Instruction Set Computer) architecture and a RISC (Reduced Instruction Set Computer) architecture.

3.4.2 BeagleBone Black (ARM)

The BeagleBone Black follows the traditional 32-bit ARM architecture, which has 15 general-purpose registers plus a *program counter*. The program counter contains the address of the instruction that will be executed next, and is only changed during a jump. Figure 3.3 lists the 32-bit registers on an ARM processor, aliases, and their typical use.†

Name	Alias	Use
R0 – R3	a1 – a4	Argument registers
R4 – R11	v1 – v8	Variables and temporaries
R9	sb	Static base register
R12	ip	Intra procedure call scratch register
R13	sp	Stack pointer
R14	lr	Link register used for return address
R15	pc	Program counter

Figure 3.3 The general-purpose registers and the program counter in the BeagleBone Black (ARM) and the meaning of each.

3.5 I/O Buses And The Fetch-Store Paradigm

We use the term *bus* to refer to a mechanism that provides the primary path between the processor and other components, namely the memory, I/O devices, and other interface controllers. Bus hardware uses a *fetch-store paradigm* in which the only operations are *fetch*, to move data from a component across the bus to the processor, and *store*, to move data from the processor across the bus to a component. For example, when it needs to access memory, the processor places a memory address on the bus and issues a *fetch* request to obtain the corresponding value. The memory hardware responds to the request by looking up the address in memory, placing the data value on the bus, and signalling the processor that the value is ready. Similarly, to store a value in memory, the processor places an address and value on the bus and issues a *store* request; the memory hardware extracts the value and stores a copy in the specified memory location. Bus hardware handles many details of the fetch-store paradigm, including signals that the processor and other components use to communicate and control access to the bus. We will see that an operating system can use a bus without knowing many details of the underlying hardware.

The example systems use *memory-mapped I/O*, which means that each I/O device is assigned a set of addresses in the bus address space. The processor uses the same

†The ARM architecture has eight additional floating point registers that are not listed here because they are not directly relevant to process management.

fetch-store paradigm to communicate with I/O devices as with memory. We will see that communication with a memory-mapped I/O device resembles data access. First, the processor computes the address associated with a device. Second, to access the device, the processor either stores a value to the address or fetches a value from the address.

3.6 Direct Memory Access

Higher-speed I/O devices (e.g., an Ethernet device) offer *Direct Memory Access* (*DMA*), which means the device contains hardware that can use the bus to communicate directly with memory. The key idea is that DMA allows I/O to proceed quickly because it does not interrupt the processor frequently nor does it require the processor to perform each data transfer. Instead, a processor can give the I/O device a list of operations, and the device proceeds from one to the next. Thus, DMA allows a processor to continue running processes while a device operates.

As an example, consider how an Ethernet device uses DMA. To receive a packet, the operating system allocates a buffer in memory and starts the Ethernet device. When a packet arrives, the device hardware accepts the packet and makes multiple bus transfers to move a copy into the buffer in memory. Finally, once the entire packet has been transferred, the device interrupts the processor. Sending a packet is equally efficient: the operating system places the packet in a buffer in memory and starts the device. The device uses the bus multiple times to fetch the packet from the buffer in memory. The device then transmits the packet on the network and interrupts the processor after the entire packet has been transferred.

We will see that the DMA hardware on the example platforms allows a processor to request multiple operations. In essence, the processor creates a list of packets to be sent and a list of buffers to be used for incoming packets. The network interface hardware uses the lists to send and receive packets without requiring the processor to restart the device after each operation. As long as the processor consumes incoming packets faster than they arrive and adds fresh buffers to the list, the network hardware device will continue to read packets. Similarly, as long as the processor continues to generate packets and add them to the list, the network hardware device will continue to transmit the packets. Later chapters explain additional DMA details, and the example code illustrates how a device driver in the operating system allocates I/O buffers and controls DMA operations.

3.7 The Bus Address Space

Each of the platforms uses a 32-bit bus address space, with addresses ranging from 0x00000000 through 0xFFFFFFFF. Some of the addresses in the bus address space correspond to memory, some to FlashROM, and others to I/O devices. The next sections provide more detail.

Memory. On each system, memory is divided into 8-bit *bytes*, with a byte being the smallest addressable unit. The C language uses the term *character* in place of *byte* because each byte can hold one ASCII character. Although a 32-bit bus can address 4 Gbytes total, not all possible addresses are assigned. The Galileo contains an 8 Mbyte Legacy SPI Flash (used to store firmware, such as a bootstrap loader), 512 Kbytes of SRAM, 256 Mbytes of DRAM, and between 256 and 512 Kbytes of storage for Arduino sketches. Note that the largest item, the DRAM, occupies only 6.25% of the address space. The BeagleBone Black contains a 4 Gbyte Flash memory (used to store firmware), and 512 Mbytes of DRAM. In the case of a BeagleBone Black, the DRAM occupies 12.5% of the bus address space, meaning that many addresses are not assigned.

Do unassigned bus addresses cause a problem? The hardware allows addresses to remain unassigned provided the addresses are not referenced. If the processor attempts to access an unassigned address, however, the hardware raises an *exception*.† For example, a bus exception can occur if a Xinu process overflows an array or generates an incorrect pointer and then tries to dereference the pointer. The important point is:

Referencing an unassigned bus address, such as an address beyond the physical memory, will cause the hardware to create an exception.

3.10 Interrupts And Interrupt Processing

Modern processors provide mechanisms that allow external I/O devices to *interrupt* the processor when they need service. In most cases, processor hardware has a closely related *exception* mechanism that is used to inform the software when an error or fault occurs (e.g., an application attempts division by zero or references a page in virtual memory that is not present in memory). From an operating system's point of view, interrupts are fundamental because they allow the processor to perform computation at the same time I/O proceeds.†

Any of the I/O devices connected to a bus can interrupt the processor when the device needs service. To do so, the device places a signal on one of the bus control lines. During normal execution of the fetch-execute cycle, hardware in the processor monitors the control line and initiates interrupt processing when the control line has been signaled. In the case of a RISC processor, the main processor does not usually contain the hardware to handle interrupts. Instead, a co-processor interacts with the bus on behalf of the main processor.

Whatever interrupt mechanisms a processor uses, the hardware or the operating system must guarantee that:

- The entire state of the processor, including the program counter and status registers, is saved when an interrupt occurs
- The processor runs the appropriate interrupt handler processor, which must have been placed in memory before the interrupt occurs
- When an interrupt finishes, the operating system and hardware provide mechanisms that restore the entire state of the processor and continue processing at the point of interruption

Interrupts introduce a fundamental idea that pervades an entire operating system. An interrupt can occur at any time, and an operating system can switch from one process to another during an interrupt. The consequence is that other processes can run at any time.

To prevent problems caused by concurrent processes trying to manipulate shared data, an operating system must take steps to avoid switching context. The simplest way to prevent other processes from executing consists of disabling interrupts. That is, the hardware includes an *interrupt mask* mechanism that an operating system can use to control interrupts. On many hardware systems, if the interrupt mask is assigned a value of zero, the processor ignores all interrupts; if the mask is assigned a non zero value, the hardware allows interrupts to occur. On some processors, the hardware has individual interrupt bits for each device, and on others, the mask provides a set of eight or sixteen levels, and each device is assigned a level. We will see that many operating system functions disable interrupts while they manipulate global data structures and I/O queues.

---

†Later chapters explain how an operating system manages interrupt and exception processing, and show how the high-level I/O operations a user performs relate to low-level device hardware mechanisms.

3.11 Vectored Interrupts

When a device interrupts, how does the hardware know the location of the code that handles the interrupt? The hardware on most processors uses a mechanism known as *vectored interrupts*. The basic idea is straightforward: each device is assigned a small integer number: 0, 1, 2, and so on. The integers are known as *interrupt level numbers* or *interrupt request numbers*. The operating system creates an array of pointers in memory known as an *interrupt vector*, where the i^{th} entry in the interrupt vector array points to the code that handles interrupts for the device with vector number i. When it interrupts, a device sends its vector number over the bus to the processor. Depending on the processor details, either the hardware or the operating system uses the vector number as an index into the interrupt vector, obtains a pointer, and uses the pointer as the address of the code to run.

Because it must be configured before any interrupts occur, an operating system initializes the interrupt vector at the same time devices are assigned addresses on the bus. The assignment of interrupt level numbers usually employs the same paradigm as address assignment. A manual assignment means a human assigns a unique interrupt level number to each device and then configures the interrupt vector addresses accordingly. An automatic approach requires bus and device hardware that can assign interrupt levels at runtime. To use the automatic approach, an operating system polls devices at startup, assigns a unique interrupt level number to each device, and initializes the interrupt vector accordingly. Automatic assignment is safer (i.e., less prone to human error), but requires more complex hardware in both the devices and the bus. We will see examples of static and automatic interrupt vector assignment.

3.12 Exception Vectors And Exception Processing

Many processors follow the same vectored approach for *exceptions* as they use for interrupts. That is, each exception is assigned a unique number: 0, 1, 2, and so on. When an exception occurs, the hardware places the exception number in a register. The operating system extracts the exception number, and uses the number as the index into an *exception vector*. A minor difference occurs between the way processor hardware handles interrupts and exceptions. We think of an interrupt as occurring *between* two instructions. Thus, one instruction has completed and the next instruction has not begun. However, an exception occurs *during* an instruction. Thus, when the processor returns from the exception, the program counter has not advanced, and the instruction can be restarted. Restarting is especially important for page fault exceptions — when a page fault occurs, the operating system must read the missing page from memory, set the page table, and then execute the instruction that caused the fault a second time.

3.13 Clock Hardware

In addition to I/O devices that transfer data, most computers include hardware that can be used to manage timed events. There are two basic forms:

- Real-time clock
- Interval timer

Real-time clock. A real-time clock circuit consists of hardware that generates a pulse regularly (e.g., 1000 times per second). To turn a real-time clock circuit into a real-time clock device, the hardware is configured to interrupt the processor on each pulse. A real-time clock device does not keep any counters, does not store the time of day, and may not have an adjustable cycle time (e.g., the pulse rate may be determined by a crystal that must be replaced to change the rate).

Interval timer. Conceptually, an interval timer consists of a real-time clock circuit that pulses at regular intervals connected to a counter that computes a tally of pulses plus a comparator circuit that compares the tally to a threshold value. An operating system can specify a threshold value and can reset the counter to zero. When the counter reaches the threshold value, the interval timer interrupts the processor. The advantage of an interval timer lies in its efficiency. Instead of interrupting continuously, an interval timer can be configured to wait until an event should occur. Of course, interval timer hardware is more complex than a real-time clock.

3.14 Serial Communication

Serial communication devices are among the simplest I/O devices available, and have been used on computers for decades. Each of the example platforms contains an RS-232 serial communication device that is used as a system console. The serial hardware handles both input and output (i.e., the transmission and reception of characters). When an interrupt occurs, the processor examines a device hardware register to determine whether the output side has completed transmission or the input side has received a character. Chapter 15 examines serial devices, and shows how interrupts are processed.

3.15 Polled vs. Interrupt-driven I/O

Most I/O performed by an operating system uses the interrupt mechanism. The operating system interacts with the device to start an operation (either input or output), and then proceeds with computation. When the I/O operation completes, the device interrupts the processor, and the operating system can choose to start another operation.

Although they optimize concurrency and permit multiple devices to proceed in parallel with computation, interrupts cannot always be used to perform I/O. For exam-

ple, consider displaying a startup message for a user before the operating system has initialized interrupts and I/O. Also consider a programmer writing operating systems code. It may be desirable to allow I/O even though it may be necessary to leave interrupts disabled during debugging. In either case, interrupts cannot be used.

The alternative to *interrupt-driven I/O* is known as *polled I/O*. When using polled I/O, the processor starts an I/O operation, but does not enable interrupts. Instead, the processor enters a loop that repeatedly checks a device status register to determine whether the operation has completed. We have already seen an example of how an operating system designer can use polled I/O when we examined functions *kputc* and *kprintf* in Chapter 2.

3.19 Perspective

The hardware specifications for a processor or I/O device contain so many details that studying them can seem overwhelming. Fortunately, many of the differences among processors are superficial — fundamental concepts apply across most hardware platforms. Therefore, when learning about hardware, it is important to focus on the overall architecture and design principles rather than on tiny details.

In terms of operating systems, many hardware details affect the overall design. In particular, the hardware interrupt mechanism and interrupt processing dominate many parts of the design. Intellectually, however, the most significant item to appreciate is the stunning disparity between the primitive facilities the hardware offers and the high-level abstractions an operating system supplies.

3.20 Hardware References

General information about the BeagleBone Black, including a photo and links to a *Getting Started* page and other materials can be found on the web site:

http://beagleboard.org/black

Details about the processor and SoC can be found on:

http://www.ti.com/product/am3358

and a data sheet for the SoC is available at:

http://www.ti.com/lit/pdf/spruh73

A general introduction to the Galileo and other platforms that use the Quark SoC can be found at:

http://www.intel.com/galileo

Information about the Galileo development board, including a photo and links to sites with more detail, can be found at:

http://www.intel.com/content/www/us/en/intelligent-systems/galileo/galileo-overview.html

14

Device–independent Input And Output

We have been left so much to our own devices — after a while, one welcomes the uncertainty of being left to other people's.

— Tom Stoppard

14.1 Introduction

Earlier chapters explain concurrent process support and memory management. Chapter 12 discusses the key concept of interrupts. The chapter describes interrupt processing, gives an architecture for interrupt code, and explains the relationship between interrupt handling and concurrent processes. Chapter 13 expands the discussion of interrupts by showing how real-time clock interrupts can be used to implement preemption and process delay.

This chapter takes a broader look at how an operating system implements I/O. The chapter explains the conceptual basis for building I/O abstractions, and presents an architecture for a general-purpose I/O facility. The chapter shows how processes can transfer data to or from a device without understanding the underlying hardware. It defines a general model, and explains how the model incorporates device-independent I/O functions. Finally, the chapter examines an efficient implementation of an I/O subsystem.

14.2 Conceptual Organization Of I/O And Device Drivers

Operating systems control and manage input and output (I/O) devices for three reasons. First, because most device hardware uses a low-level interface, the software interface is complex. Second, because a device is a shared resource, an operating system provides access according to policies that make sharing fair and safe. Third, an operating system defines a high-level interface that hides details and allows a programmer to use a consistent and uniform set of operations when interacting with devices.

The I/O subsystem can be divided into three conceptual pieces: an abstract interface consisting of high-level I/O functions that processes use to perform I/O, a set of physical devices, and *device driver* software that connects the two. Figure 14.1 illustrates the organization.

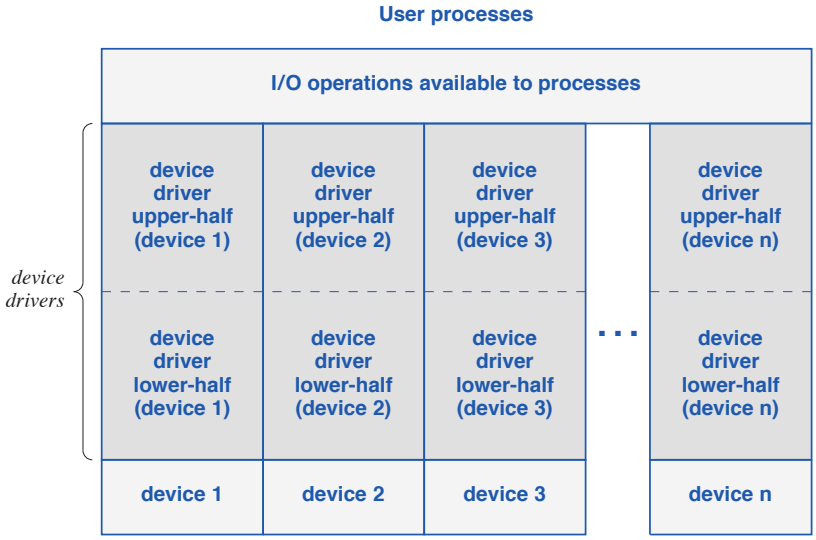

Figure 14.1 The conceptual organization of the I/O subsystem with device driver software between processes and the underlying device hardware.

As the figure indicates, device driver software bridges the gap between high-level, concurrent processes and low-level hardware. We will see that each driver is divided into two conceptual pieces: an *upper half* and a *lower half*. Functions in the upper half are invoked when a process requests I/O. Upper-half functions implement operations such as *read* and *write* by transferring data to or from a process. The lower half contains functions that are invoked by interrupts. When a device interrupts, the interrupt dispatcher invokes a lower-half handler function. The handler services the interrupt, interacts with the device to transfer data, and may start an additional I/O operation.

14.3 Interface And Driver Abstractions

The ultimate goal of an operating system designer lies in creating convenient programming abstractions and finding efficient implementations. With respect to I/O, there are two aspects:

- Interface abstraction
- Driver abstractions

Interface abstraction. The question arises: what I/O interface should an operating system supply to processes? There are several possibilities, and the choice represents a tradeoff among goals of flexibility, simplicity, efficiency, and generality. To understand the scope of the problem, consider Figure 14.2, which lists a set of example devices and the types of operations appropriate for each.

Device	I/O paradigm
hard drive	move to a position and transfer a block of data
keyboard	accept individual characters as entered
printer	transfer an entire document to be printed
audio output	transfer a continuous stream of encoded audio
wireless network	send or receive a single network packet

Figure 14.2 Example devices and the paradigm that each device uses.

Early operating systems provided a separate set of I/O operations for each individual hardware device. Unfortunately, building device-specific information into software means the software must be changed when an I/O device is replaced by an equivalent device from another vendor. A slightly more general approach defines a set of operations for each type of device, and requires the operating system to perform the appropriate low-level operations on a given device. For example, an operating system can offer abstract functions *send_network_packet* and *receive_network_packet* that can be used to transfer network packets over any type of network. A third approach originated in Multics and was popularized by Unix: choose a small set of abstract I/O operations that are sufficient for all I/O.

Driver abstractions. We think of the second category of abstraction as providing semantics. One of the most important semantic design questions focuses on synchrony: does a process block while waiting for an I/O operation to complete? A *synchronous* interface, similar to the one described earlier, provides blocking operations. For example, to request data from a keyboard in a synchronous system, a process invokes an

upper-half routine that blocks the process until a user presses a key. Once a user makes a keystroke, the device interrupts and the dispatcher invokes a lower-half routine that acts as a handler. The handler unblocks a waiting process and reschedules to allow the process to run. In contrast, an *asynchronous* I/O interface allows a process to continue executing after the process initiates an I/O operation. When the I/O completes, the driver must inform the requesting process (e.g., by invoking the *event handler* function associated with the process). We can summarize:

> *When using a synchronous I/O interface, a process is blocked until the operation completes. When using an asynchronous I/O interface, a process continues to execute and is notified when the operation completes.*

Each approach has advantages. An asynchronous interface is useful in situations where a programmer needs to control the overlap of I/O and computation. A synchronous approach has the advantage of being easier to program.

Another design issue arises from the format of data and the size of transfers. Two questions arise. First, will data be transferred in blocks or bytes? Second, how much data can be transferred in a single operation? Observe that some devices transfer individual data bytes, some transfer a variable-size chunk of data (such as a network packet or a line of text), and others, such as disks, transfer fixed-size blocks of data. Because a general-purpose operating system must handle a variety of I/O devices, an I/O interface may require both single-byte transfers as well as multi-byte transfers.

A final design question arises from the parameters that a driver supplies and the way a driver interprets individual operations. For example, does a process specify a location on disk and then repeatedly request the next disk block, or does the process specify a block number in each request? The example device driver presented in the next chapter illustrates the use of parameters.

The key idea is:

> *In a modern operating system, the I/O interface and device drivers are designed to hide device details and present a programmer with convenient, high-level abstractions.*

14.4 An Example I/O Interface

Experience has shown that a small set of I/O functions is both sufficient and convenient. Thus, our example system contains an I/O subsystem with nine abstract I/O operations that are used for all input and output. The operations have been derived from the I/O facilities in the Unix operating system. Figure 14.3 lists the operations and the purpose of each.

Operation	Purpose
close	Terminate use of a device
control	Perform operations other than data transfer
getc	Input a single byte of data
init	Initialize the device at system startup
open	Prepare the device for use
putc	Output a single byte of data
read	Input multiple bytes of data
seek	Move to specific data (usually a disk)
write	Output multiple bytes of data

Figure 14.3 The set of abstract I/O interface operations used in Xinu.

14.5 The Open-Read-Write-Close Paradigm

Like the programming interface in many operating systems, the example I/O interface follows an *open-read-write-close* paradigm. That is, before it can perform I/O, a process must *open* a specific device. Once it has been opened, a device is ready for the process to call *read* to obtain input or *write* to send output. Finally, once it has finished using a device, the process calls *close* to terminate use.

To summarize:

> *The open-read-write-close paradigm requires a process to open a device before use and close a device after use.*

Open and *close* allow the operating system to manage devices that require exclusive use, prepare a device for data transfer, and stop a device after transfer has ended. Closing a device may be useful, for example, if a device needs to be powered down or placed in a standby state when not in use. *Read* and *write* handle the transfer of multiple data bytes to or from a buffer in memory. *Getc* and *putc* form a counterpart for the transfer of a single byte (usually a character). *Control* allows a program to control a device or a device driver (e.g., to check supplies in a printer or select the channel on a wireless radio). *Seek* is a special case of *control* that applies to randomly accessible storage devices, such as disks. Finally, *init* initializes the device and driver at system startup.

Consider how the operations apply to a console window. *Getc* reads the next character from the keyboard, and *putc* displays one character in the console window. *Write* can display multiple characters with one call, and *read* can read a specified number of characters (or all that have been entered, depending on its arguments). Finally, *control* allows the program to change parameters in the driver to control such things as whether the system stops echoing characters as a password is entered.

14.6 Bindings For I/O Operations And Device Names

How can an abstract operation such as *read* act on an underlying hardware device? The answer lies in a binding. When a process invokes a high-level operation, the operating system maps the call to a device driver function. For example, if a process calls *read* on a console device, the operating system passes the call to a function in the console device driver that implements *read*. In doing so, the operating system hides both the hardware and device driver details from application processes and presents an abstract version of devices. By using a single abstract device for a keyboard and a window on the display, an operating system can hide the fact that the underlying hardware consists of two separate devices. Furthermore, an operating system can hide device details by presenting the same high-level abstraction for the hardware from multiple vendors. The point is:

> *An operating system creates a virtual I/O environment — a process can only perceive peripheral devices through the abstractions that the interface and device drivers provide.*

In addition to mapping abstract I/O operations onto driver routines, an operating system must map device names onto devices. A variety of mappings have been used. Early systems required a programmer to embed device names in source code. Later systems arranged for programs to use small integers to identify devices, and allowed a command interpreter to link each integer with a specific device when the application was launched. Many modern systems embed devices in a file naming hierarchy, allowing an application to use a symbolic name for each device.

Early and late binding each have advantages. An operating system that waits until runtime to bind the name of an abstract device to a real device and a set of abstract operations to device driver functions is flexible. However, such late binding systems incur more computational overhead, making them impractical in the smallest embedded systems. At the other extreme, early binding requires device information to be specified when an application is written. Thus, the essence of I/O design consists of synthesizing a binding mechanism that allows maximum flexibility within the required performance bounds.

Our example system uses an approach that is typical of small embedded systems: information about devices is specified before the operating system is compiled. For each device, the operating system knows exactly which driver functions correspond to

each of the abstract I/O operations. In addition, the operating system knows the underlying hardware device to which each abstract device corresponds. Thus, the operating system must be recompiled whenever a new device is added or when an existing device is removed. Because application code does not contain information about specific device hardware, application code can be ported from one system to another easily. For example, an application that only performs I/O operations on a *CONSOLE* serial port will work on any Xinu system that offers a *CONSOLE* device and the appropriate driver, independent of the physical device hardware and interrupt structure.

14.7 Device Names In Xinu

In Xinu, the system designer must specify a set of abstract devices when the system is configured. The configuration program assigns each device name a unique integer value known as a *device descriptor*. For example, if a designer specifies a device name *CONSOLE*, the configuration program might assign descriptor zero. The configuration program produces a header file that contains *#define* statements for each name. Thus, once the header file has been included, a programmer can reference *CONSOLE* in the code. For example, if *CONSOLE* has been assigned descriptor zero, the call:

```
read(CONSOLE, buf, 100);
```

is equivalent to:

```
read(0, buf, 100);
```

To summarize:

> *Xinu uses a static binding for device names. Each device name is bound to an integer descriptor at configuration time before the operating system is compiled.*

14.8 The Concept Of A Device Switch Table

Each time a process invokes a high-level I/O operation such as *read* or *write*, the operating system must forward the call to the appropriate driver function. To make the implementation efficient, Xinu uses an array known as a *device switch table*. The integer descriptor assigned to a device is an index into the device switch table. To understand the arrangement, imagine a two-dimensional array. Conceptually, each row of the array corresponds to a device, and each column corresponds to an abstract operation. An entry in the array specifies the driver function to use to perform the operation.

For example, suppose a system contains three devices defined as follows:

- CONSOLE, a serial device used to send and receive characters
- ETHER, an Ethernet interface device
- DISK, a hard drive

Figure 14.4 illustrates part of a device switch table that has a row for each of the three devices and a column for each I/O operation. Items in the table represent the names of driver functions that perform the operation given by the column on the device given by the row.

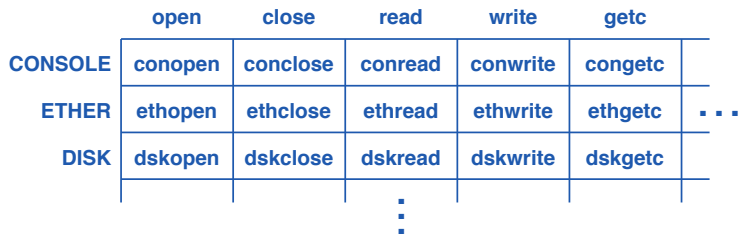

Figure 14.4 Conceptual organization of the device switch table with one row per device and one column per abstract I/O operation.

As an example, suppose a process invokes the *write* operation on the *CONSOLE* device. The operating system goes to the row of the table that corresponds to the *CONSOLE* device, finds the column that corresponds to the *write* operation, and calls the function named in the entry, *conwrite*.

In essence, each row of the device switch table defines how the I/O operations apply to a single device, which means I/O semantics can change dramatically among devices. For example, when it is applied to a *DISK* device, a *read* might transfer a block of 512 bytes of data. However, when it is applied to a *CONSOLE* device, *read* might transfer a line of characters that the user has entered.

The most significant aspect of the device switch table arises from the ability to define a uniform abstraction across multiple physical devices. For example, suppose a computer contains a disk that uses 1 Kbyte sectors and a disk that uses 4 Kbyte sectors. Drivers for the two disks can present an identical interface to applications, while hiding the differences in the underlying hardware. That is, a driver can always transfer 4 Kbytes to a user, and convert each transfer into four 1 Kbyte disk transfers.

```
#define LF_DISK_DEV     RAM0
```

Each entry in *devtab* corresponds to a single device. The entry specifies the address of functions that constitute the driver for the device, the device CSR address, and other information used by the driver. Fields *dvinit*, *dvopen*, *dvclose*, *dvread*, *dvwrite*, *dvseek*, *dvgetc*, *dvputc*, and *dvcntl* hold the addresses of driver routines that correspond to high-level operations. Field *dvminor* contains an integer index into the control block array for the device. A minor device number accommodates multiple identical hardware devices by allowing a driver to maintain a separate control block entry for each physical device. Field *dvcsr* contains the hardware CSR address for the device. The control block for a device holds additional information for the particular instance of the device and the driver; the contents depend on the device, but may include such things as input or output buffers, device status information (e.g., whether a wireless networking device is currently in contact with another wireless device), and accounting information (e.g., the total amount of data sent or received since the system booted).

14.10 The Implementation Of High–level I/O Operations

Because it isolates high-level I/O operations from underlying details, the device switch table allows high-level functions to be created before any device drivers have been written. One of the chief benefits of such a strategy arises because a programmer can build pieces of the I/O system without requiring specific hardware devices to be present.

The example system contains a function for each high-level I/O operation. Thus, the system contains functions *open*, *close*, *read*, *write*, *getc*, *putc*, and so on. However, a function such as *read* does not perform I/O. Instead, each high-level I/O function operates *indirectly*: the function uses the device switch table to find and invoke the appropriate low-level device driver routine to perform the requested function. The point is:

> *Instead of performing I/O, high-level functions such as* read *and* write *use a level of indirection to invoke a low-level driver function for the specified device.*

An examination of the code will clarify the concept. Consider the *read* function found in file *read.c*:

```
/* read.c - read */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  read  -  Read one or more bytes from a device
 *------------------------------------------------------------------------
 */
syscall read(
          did32         descrp,         /* Descriptor for device        */
          char          *buffer,        /* Address of buffer            */
          uint32        count           /* Length of buffer             */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvread) (devptr, buffer, count);
        restore(mask);
        return retval;
}
```

The arguments to *read* consist of a *device descriptor*, the address of a buffer, and an integer that gives the maximum number of bytes to read. *Read* uses the device descriptor, *descrp*, as an index into *devtab*, and assigns pointer *devptr* the address of the device switch table entry. The *return* statement contains code that performs the task of invoking the underlying device driver function and returning the result to the function that called *read*. The code:

```
(*devptr->dvread) (devptr, buffer, count)
```

performs the indirect function call. That is, the code invokes the driver function given by field *dvread* in the device switch table entry, passing the function three arguments: the address of the *devtab* entry, *devptr*, the buffer address, *buffer*, and a count of characters to read, *count*.

```
/* init.c - init */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  init  -  Initialize a device and its driver
 *------------------------------------------------------------------------
 */
syscall init(
          did32         descrp          /* Descriptor for device        */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvinit) (devptr);
        restore(mask);
        return retval;
}
```

14.12 Open, Close, And Reference Counting

Functions *open* and *close* operate similar to other I/O functions by using the device switch table to call the appropriate driver function. One motivation for using *open* and *close* arises from their ability to establish ownership of a device or prepare a device for use. For example, if a device requires exclusive access, *open* can block a subsequent user until the device becomes free. As another example, consider a system that saves power by keeping a disk device idle when the device is not in use. Although a designer could arrange to use the *control* function to start or stop a disk, *open* and *close* are more convenient. Thus, a disk can be powered on when a process calls *open*, and powered off when a process calls *close*.

Although a small embedded system might choose to power down a disk whenever a process calls *close* on the device, larger systems need a more sophisticated mechanism because multiple processes can use a device simultaneously. Thus, most drivers employ a technique known as *reference counting*. That is, a driver maintains an integer variable

that counts the number of processes using the device. During initialization, the reference count is set to zero. Whenever a process calls *open*, the driver increments the reference count, and whenever a process calls *close*, the driver decrements the reference count. When the reference count reaches zero, the driver powers down the device.

14.15 Perspective

Device-independent I/O is now an integral part of mainstream computing, and the advantages seem obvious. However, it took decades for the computing community to reach consensus on device-independent I/O and to devise a set of primitives. Some of the contention arose because programming languages each define a set of I/O abstractions. For example, FORTRAN used device numbers and required a mechanism that could bind each number to an I/O device or file. Operating system designers wanted to accommodate all languages because a large volume of code has been written in each. So, questions arise. Have we chosen the best set of device-independent I/O functions, or have we merely become so accustomed to using them that we fail to look for alternatives? Are the functions we are using ideal for systems that focus on graphical devices?

14.16 Summary

An operating system hides the details of peripheral devices, and provides a set of abstract, device-independent functions that can be used to perform I/O. The example system uses nine abstract functions: *open*, *close*, *control*, *getc*, *putc*, *read*, *write*, *seek*, and an initialization function, *init*. In our design, each of the I/O primitives operates *synchronously*, delaying a calling process until the request has been satisfied (e.g., function *read* delays the calling process until data has arrived).

The example system defines an abstract device name (such as *CONSOLE*) for each device, and assigns the device a unique integer device descriptor. The system uses a device switch table to bind a descriptor to a specific device at runtime. Conceptually, the device switch table contains one row for each device and one column for each

abstract I/O operation; additional columns point to a control block for the device, and a minor device number is used to distinguish among multiple copies of a physical device. A high-level I/O operation, such as *read* or *write*, uses the device switch table to invoke the device driver function that performs the requested operation on the specified device. Individual drivers interpret the calls in a way meaningful to a particular device; if an operation makes no sense when applied to a particular device, the device switch table is configured to invoke function *ioerr*, which returns an error code.

EXERCISES

14.1 Identify the set of abstract I/O operations available in Linux.

14.2 Find a system that uses *asynchronous* I/O, and identify the mechanism by which a running program is notified when the operation completes. Which approach, synchronous or asynchronous, makes it easier to program? Explain.

14.3 The chapter discusses two separate bindings: the binding from a device name (e.g., *CONSOLE*) to a descriptor (e.g., 0) and the binding from a device descriptor to a specific hardware device. Explain how Linux performs the two bindings.

14.4 Consider the implementation of device names in the example code. Is it possible to write a program that allows a user to enter a device name (e.g., *CONSOLE*), and then open the device? Why or why not?

14.5 Assume that in the course of debugging you begin to suspect that a process is making incorrect calls to high-level I/O functions (e.g., calling *seek* on a device for which the operation makes no sense). How can you make a quick change to the code to intercept such errors and display the process ID of the offending process? (Make the change without recompiling the source code.)

14.6 Are the abstract I/O operations presented in the chapter sufficient for all I/O operations? Explain. Hint: consider socket functions found in Unix.

14.7 Xinu defines the device subsystem as the fundamental I/O abstraction and merges files into the device system. Unix systems define the file system as the fundamental abstraction and merge devices into the file system. Compare the two approaches and list the advantages of each.

15

An Example Device Driver

It's hard to find a good driver these days, one with character and style.

— Unknown

15.1 Introduction

Chapters in this section of the text explore the general structure of an I/O system, including interrupt processing and real-time clock management. The previous chapter presents the organization of the I/O subsystem, a set of abstract I/O operations, and an efficient implementation using a device switch table.

This chapter continues the exploration of I/O. The chapter explains how a driver can define an I/O service at a high level of abstraction that is independent of the underlying hardware. The chapter also elaborates on the conceptual division of a device driver into upper and lower halves by explaining how the two halves share data structures, such as buffers, and how they communicate. Finally, the chapter shows the details of a particular example: a driver for an asynchronous character-oriented serial device.

15.2 Serial Communication Using UART Hardware

The console line on most embedded systems uses a *Universal Asynchronous Transmitter and Receiver (UART)* chip that implements serial communication according to the RS-232 standard. UART hardware is primitive — it only provides the ability to send and receive individual bytes. It does not interpret the meaning of bytes or provide functions such as the use of backspace to erase previous input.

15.3 The Tty Abstraction

Xinu uses the name *tty* to refer to the abstraction of an interface used with a character-oriented text window that displays characters the user enters on a serial interface, such as a keyboard.† In broad terms, a tty device supports two-way communication: a process can send characters to the output side and/or receive characters from the input side. Although the underlying serial hardware mechanism operates the input and output independently, the tty abstraction allows the two to be treated as a single mechanism in which the input and output appear to be connected. For example, our tty driver supports *character echo*, which means that the input side of the driver can output a copy of each incoming character. Echo is especially important when a user is typing on a keyboard and expects to see characters displayed on a screen.

The tty abstraction illustrates an important feature of many device drivers: multiple *modes*. At any time, the driver operates in one mode, and the mode can be changed at runtime. Figure 15.1 summarizes the three modes our driver supports.

Mode	Meaning
raw	The driver delivers each incoming character as it arrives without echoing the character, buffering a line of text, performing translation, or controlling the output flow
cooked	The driver buffers input, echoes characters in a readable form, honors backspace and line kill, allows type-ahead, handles flow control, and delivers an entire line of text
cbreak	The driver handles character translation, echoing, and flow control, but instead of buffering an entire line of text, the driver delivers each incoming character as it arrives

Figure 15.1 Three modes supported by the tty abstraction.

Raw mode is intended to give applications access to input characters with no preprocessing. In raw mode, the tty driver merely delivers input without interpreting or changing characters. The driver does not echo characters nor does it handle flow control. Raw mode is useful when handling non-interactive communication, such as downloading a binary file over a serial line or using a serial device to control a sensor.

Cooked mode handles interactive keyboard input. Each time it receives a character, the driver echoes the character (i.e., transmits a copy of the character to the output), which allows a user to see characters as they are typed. The driver has a parameter to control character echoing, which means an application can turn echo off and back on (e.g., to prompt for a password). Cooked mode supports *line buffering*, which means that the driver collects all characters of a line before delivering them to a reading proc-

†The name *tty* is taken from early Unix systems that used an ASCII Teletype device that consisted of a keyboard and an associated printer mechanism.

ess. Because the tty driver performs character echo and other functions at interrupt time, a user can type ahead, even if no application is reading characters (e.g., a user can type the next command while the current command is running). A chief advantage of cooked mode arises from the ability to edit a line, either by backspacing or by typing a special character to erase the entire line.

Cooked mode also provides *flow control* and *input mapping*. When enabled, flow control allows a user to type *control-s* to stop output temporarily, and later to type *control-q* to restart output. Input mapping handles hardware (or applications) that use the two-character sequence of *carriage return* (*cr*) and *linefeed* (*lf*) to terminate a line of text instead of a single *lf*. Cooked mode contains a *crlf*† parameter that controls how the driver handles line termination. When a user presses the key labeled *ENTER* or *RETURN*, the driver consults the parameter to decide whether to pass a *linefeed* (also called a *NEWLINE*) character to the application or to map the *linefeed* into a pair of characters, *carriage return* followed by *linefeed*.

Cbreak mode provides a compromise between cooked and raw modes. In cbreak mode, each character is delivered to the application instantly, without waiting to accumulate a line of text. Thus, the driver does not buffer input, nor does the driver support backspace or line kill functions. However, the driver does handle both character echo and flow control.

15.4 Organization Of A Tty Device Driver

Like most device drivers, the example tty driver is partitioned into an *upper half* that contains functions called by application processes (indirectly through the device switch table), and a *lower half* that contains functions invoked when the device interrupts. The two halves share a data structure that contains information about the device, the current mode of the driver, and buffers for incoming and outgoing data. In general, upper-half functions move data to or from the shared structure and have minimal interaction with the device hardware. For example, an upper-half function places outgoing data in the shared structure where a lower-half function can access and send the data to the device. Similarly, the lower half places incoming data in the shared structure where an upper-half function can extract it.

The motivation for driver partitioning can be difficult to appreciate at first. We will see, however, that dividing a driver into two halves is fundamental because the division allows a system designer to decouple normal processing from hardware interrupt processing and understand exactly how each function is invoked. The point is:

> *When creating a device driver, a programmer must be careful to preserve the division between upper-half and lower-half functions because upper-half functions are called by application processes and lower-half functions are invoked by interrupts.*

†Pronounced *curl-if*.

15.5 Request Queues And Buffers

In most drivers, the shared data structure contains two key items:

- Request queue
- Buffered I/O

Request queue. In principle, the most important item found in the data structure shared by upper-half and lower-half functions is a queue into which the upper half places requests. Conceptually, the *request queue* connects high-level operations that applications specify and low-level actions that must be performed on the device. Each driver has its own set of requests, and the contents of elements on a request queue depend on the underlying device as well as the operations to be performed. For example, requests issued to a disk device specify the direction of transfer (read or write), a location on the disk, and a buffer for the data to be transferred. Requests issued to a network device might specify a set of packet buffers, and specify whether to transmit a packet from the buffer or receive an incoming packet. Our example driver has two queues: one contains a set of outgoing characters and the other holds incoming characters. In essence, a character in an outgoing queue is a request to transmit, and space in the input queue serves as a request to receive.

Buffered I/O. Most drivers buffer incoming and outgoing data. Output buffering allows an application to deposit an outgoing item in a buffer, and then continue processing. The item remains in the buffer until the hardware is ready to accept the item. Input buffering allows a driver to accept data from a device before an application is ready to receive it. An incoming item remains in the buffer from the time the device deposits the item until a process requests it.

Buffers are important for several reasons. First, a driver can accept incoming data before a user process consumes the data. Input buffering is especially important for asynchronous devices like a network interface that can receive packets at any time or a keyboard on which a user can press a key at any time. Second, buffering permits an application to read or write arbitrary amounts of data, even if the underlying hardware transfers entire blocks. By placing a block in a buffer, an operating system can satisfy subsequent requests from the buffer without an I/O transfer. Third, output buffering permits the driver to perform I/O concurrently with processing because a process can write data into a buffer, and then continue executing while the driver writes the data to the device.

Our tty driver uses circular input and output buffers, with an extra circular buffer for echoed characters (echoed characters are kept in a buffer separate from normal output because echoed characters have higher priority). We think of each buffer as a conceptual queue, with characters being inserted at the tail and removed from the head. Figure 15.2 illustrates the concept of a circular output buffer, and shows the implementation with an array of bytes in memory.

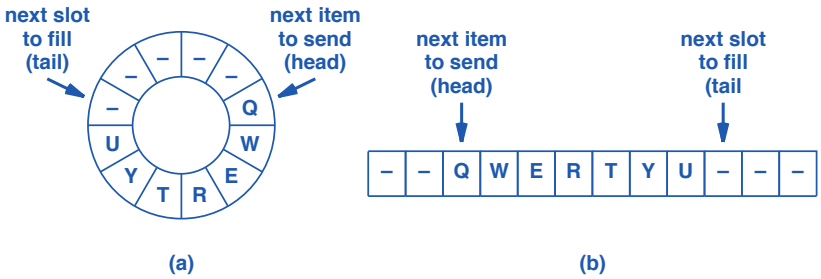

Figure 15.2 (a) A circular output buffer acting as a queue, and (b) the implementation with an array of bytes.

Output functions deposit characters to be sent in the output buffer and return to their caller. When it places characters in the output buffer, an upper-half function must also start output interrupts on the device. Whenever the device generates an output interrupt, the lower half extracts up to sixteen characters from the output buffer, and deposits the characters in the device's output FIFO.† Once all characters in the output FIFO have been transmitted, the device will interrupt again. Thus, output continues until the output buffer becomes empty at which time the driver stops output and the device becomes idle.

Input works the other way around. Whenever it receives characters, the device interrupts and the interrupt dispatcher calls a lower-half function (i.e., *ttyhandler*). The interrupt handler extracts the characters from the device's input FIFO and deposits them in the circular input buffer. When a process calls an upper-half function to read input, the upper-half function extracts characters from the input buffer.

Conceptually, the two halves of a driver only communicate through shared buffers. Upper-half functions place outgoing data in a buffer and extract incoming data from a buffer. The lower half extracts outgoing data from the buffer and sends it to the device, and places incoming data in the buffer. To summarize:

Upper-half functions transfer data between processes and buffers; the lower half transfers data between buffers and the device hardware.

15.6 Synchronization Of Upper Half And Lower Half

In practice, the two halves of the driver usually need to do more than manipulate a shared data structure. For example, an upper-half function may need to start an output transfer if a device is idle. More important, the two halves need to coordinate operations on the request queue and the buffers. For example, if all slots in the output buffer are full when a process tries to write data, the process must be blocked. Later, when characters have been sent to the device and buffer space becomes available, the blocked

†To improve efficiency, most UART hardware has a small on-board character buffer that can hold up to 16 outgoing characters at a time.

process must be allowed to proceed. Similarly, if the input buffer is empty when a process attempts to read from a device, the process must be blocked. Later, when input has been received and placed in the buffer, the process that is waiting for input must be allowed to proceed.

At first glance, synchronization between the upper half and lower half of a driver appears to consist of two instances of *producer–consumer coordination* that can be solved easily with semaphores. On output, the upper-half functions produce data that the lower-half functions consume, and on input, the lower half produces input data that the upper-half functions consume. Input poses no problem for the producer–consumer paradigm; a semaphore can be created that handles coordination. When a process calls an upper-half input function, the process *waits* on the input semaphore until the lower half produces an input data item and *signals* the semaphore.

Output poses an added twist. To understand the problem, recall our restriction on interrupt processing: because it can be executed by the null process, an interrupt function cannot call a function that moves the executing process to any state other than *ready* or *current*. In particular, lower-half routines cannot call *wait*. Consequently, a driver cannot be designed in which a semaphore allows upper-half functions to produce data and lower-half functions to consume data.

How can upper-half and lower-half functions coordinate to control output? Surprisingly, a semaphore solves the problem easily. The trick is to turn around the call to *wait* by changing the purpose of the output semaphore. Instead of having a lower-half routine *wait* for the upper half to produce data, we arrange for the upper half to *wait* for space in the buffer. Thus, we do not view the lower half as a consumer. Instead, a lower-half output function acts as a producer to generate space (i.e., slots) in the buffer, and signals the output semaphore for each slot. To summarize:

> *Semaphores can be used to coordinate the upper half and lower half of a device driver. To avoid having lower-half functions block, output is handled by arranging for upper-half functions to wait for buffer space.*

A character is always inserted at the tail and taken from the head, independent of whether a buffer is used for input or output.

Initially, the head and tail each point to location zero, but there is never any confusion about whether an input or output buffer is completely empty or completely full because each buffer has a semaphore that gives the count of characters in the buffer. Semaphore *tyisem* controls the input buffer, and a nonnegative count *n* means the buffer contains *n* characters. Semaphore *tyosem* controls the output buffer, and a nonnegative count *n* means the buffer contains *n* unfilled slots. The echo buffer is an exception. Our design assumes echo is used for a human typing, which means that only a few characters will ever occupy the echo queue. Therefore, we assume that no overflow will occur, which means that no semaphore is needed to control the queue.