CHAPTER 9

INPUT/OUTPUT

"Here comes the 64-bit local bus."

9.0 INTRODUCTION

Of course you're aware that no matter how powerful the CPU is, a computer system's usefulness ultimately depends on its input and output facilities. Without I/O there is no possibility of keyboard input, of screen output, of printout, or even of disk storage and retrieval. Although you might be inclined to think of I/O in terms of user input and output, there would be no computer network or Internet access either. To the CPU and its programs, all these devices require specialized input and output processing facilities and routines.

In fact, for most business programs and for nearly every multimedia application, I/O is the predominant factor. E-commerce applications offer an even bigger challenge: Web services generally require massive amounts of fast I/O to handle and process I/O requests as they occur. The speed at which most of these programs operate is determined by the ability of their I/O operations to stay ahead of their processing. With PCs rapidly increasing in CPU processing capability, but still somewhat limited in I/O processing, it has been greater I/O capability that has maintained, until recently, the advantage of mainframe computers over PCs for business transaction processing.

We handled input and output in the Little Man Computer by providing input and output baskets for that purpose. Each input instruction transferred one three-digit data number from the input basket to the calculator; similarly, each output instruction transferred one data number from the calculator to the output basket. If we wanted to input three numbers, for example, an input instruction had to be executed three times. This could be done with three separate input instructions or in a loop, but either way, each individual piece of data required the execution of a separate input instruction.

It is possible to transfer data between input and output devices and the CPU of a real computer in a similar manner. In the real computer, the in basket and out basket are commonly replaced by a bus interface that allows a direct transfer between a register within the CPU and a register within an I/O module that controls the particular device. Both input and output are handled similarly. The technique is known as programmed I/O.

There are a number of complicating factors in handling input/output processes (which we will normally simply call I/O) in a real computer. Although the method of transferring data one word at a time does really exist, and may be adequate and appropriate for some slow-operating I/O devices, the volume of data commonly transferred in I/O devices, such as disks and tapes, makes this method too slow and cumbersome to be practical as the only I/O transfer method in a modern high-speed machine. We need to consider some method of transferring data in blocks rather than executing an instruction for each individual piece of data.

The problem is further complicated by the fact that in a real computer, there may be many input and output devices all trying to do I/O, sometimes at the same time. There needs to be a way of distinguishing and separating the I/O from these different devices. Additionally, devices operate at different speeds from each other and from the CPU.

An inkjet printer may output characters at a rate of 150 per second, whereas a disk may transfer data at a rate of tens or hundreds of thousands, or even millions, of bytes per second. Synchronization of these different operations must be achieved to prevent data loss.

Finally, it should be noted that I/O operations take up a lot of computer time. Even if a block of data can be transferred between the CPU and a disk with a single instruction, much time is potentially wasted waiting for the completion of the task. A CPU could execute millions of instructions in the time it takes a printer to print a single character. In a large modern computer, the number of I/O operations may be very large. It would be convenient and useful to be able to use the CPU for other tasks while these I/O transfers are taking place.

In the computer, several different techniques are combined to resolve the problem of synchronizing and handling I/O between a variety of different I/O devices operating with different quantities of data at different speeds. In this chapter, we first consider the I/O requirements of some commonly used devices. This discussion, which appears in Section 9.1, leads to a set of requirements that the I/O-CPU interface should meet to optimize system performance. Next, in Section 9.2 we briefly review programmed I/O, the method used in the Little Man Computer, and consider its limitations. Section 9.3 addresses the important issue of interrupts, the method used to communicate events that need special attention to the CPU. Interrupts are the primary means for the user to interact with the computer, as well as the means used for communication between the CPU and the various I/O devices connected to the system. In Section 9.4 we look at Direct Memory Access, or DMA, a more efficient alternative technique used to perform I/O in the computer. DMA provides the ability to utilize the CPU more fully while I/O operations are taking place. Finally, Section 9.5 considers the I/O modules that provide the capability both to control the I/O devices and to interact with the CPU and memory.

9.1 CHARACTERISTICS OF TYPICAL I/O DEVICES

Before discussing the techniques that are used in the real computer for performing I/O, it will help to consider some characteristics of the devices that will typically be connected to the computer. In this chapter we are not interested in the inner workings of these devices nor with the interconnection of the various computer components and I/O devices that make up the whole computer system—these discussions we'll save for Chapters 10 and 11, respectively. For now, we are only interested in those characteristics of these devices that will affect the I/O capabilities of the computer, in particular the speed and quantity of data transfer required to use the computer efficiently and fully. This survey is intended to be intuitive: what must be true about the I/O, based on what you already know about the particular devices from your own practical experience. Although this discussion may seem like a digression, it is intended to establish a set of basic principles and requirements that will help you to better understand the reasons behind the methods that are used to perform I/O in computers.

Consider, for example, the keyboard as an input device. The keyboard is basically a character-based device. You are probably already aware that typing on the keyboard of your PC results in Unicode or ASCII input to the computer, one character at a time. Even mainframe terminals, many of which can send text to the computer a page at a time, only transmit a page occasionally, so the data rate for keyboards is obviously very slow compared to the speed at which the CPU processes the data.

Input from the keyboard is very slow because it is dependent on the speed of typing, as well as on the thought process of the user. There are usually long thinking pauses between bursts of input, but even during those bursts, the actual input requirements to the computer are very slow compared to the capability of the computer to execute input instructions. Thus, we must assume that if the computer is simply performing a single task, it will spend most of its time waiting for input from the keyboard.

It is also useful to note that there are two different types of keyboard input. There is input that is expected by the application program in response to a ''read'' statement of some kind requesting input data for the program. Then there are other times when the user wishes to interrupt what the computer is doing. On many computers, a character such as Control-''C'' or Control-''D'' or Control-''Q'' can be typed to stop the program that is running. Control-''S'' is used on some machines to stop the display from scrolling. Typing Control-Alt-Delete on a PC will stop normal processing and open an administrative window that can be used to kill a program or shut down the computer. These are examples of unpredicted input, since the executing program is not necessarily awaiting specific input at those times. Using the input method that we already described would not work: the unexpected input would not be noticed, possibly for a long time until the next input instruction was executed for some later expected input.

Finally, on a multiuser system, there may be many keyboards connected to a single computer. The computer must be able to distinguish between them, must not lose input data even if several keyboards send a character simultaneously, and must be able to respond quickly to each keyboard. The physical distances from the computer to these keyboards may be long.

Another input device that will generate unexpected input is the mouse. When you move the mouse, you expect the cursor to move on the screen. Clicking on a mouse button may serve as expected input to a program, or it may be unexpected and change the way in which the program is executing. In fact, unexpected input is fundamental to programs written in modern event-driven languages such as Visual Basic and Java. When the user selects an item on a drop-down menu or clicks on a toolbar icon, she expects a timely response. Again, data rates are slow.

Printers and display screens must operate over a wide range of data rates. Although most monitors and printers are capable of handling pure ASCII or Unicode text, most modern output is produced graphically or as a mixture of font descriptors, text, bitmap graphics, and object graphics, a page or a screen at a time, using a page description language. The choice of page description language and mixture of elements is determined by the capabilities of the printer or graphics card. Clearly, output to a printer consisting only of an occasional page or two of text will certainly not require a high data rate regardless of the output method used.

The output of high resolution bitmap graphics and video images to a monitor is quite a different situation. If the graphics must be sent to the graphics card as bitmap images, even in compressed form, with data for each pixel to be produced, it may take a huge amount of data to produce a single picture, and high-speed data transfer will be essential. A single, color image on a high-resolution screen may require several megabytes of data, and it is desirable to produce the image on the screen as fast as possible. If the image represents video, extremely high data transfer rates are required. This suggests that screen image updates may require bursts of several megabytes per second, even when data

compression methods are used to reduce the transfer rate. It may also suggest to you why it is nearly impossible to transmit high quality images quickly over voice-grade phone lines using modems.

Contrast the I/O requirements of keyboards, screens, and printers with those of disks and DVDs. Since the disk is used to store programs and data, it would be very rare that a program would require a single word of data or program from the disk. Disks are used to load entire programs or store files of data. Thus, disk data is always transferred in blocks, never as individual bytes or words. Disks may operate at transfer rates of tens or, even, hundreds of megabytes per second. As storage devices, disks must be capable of both input and output, although not simultaneously. On a large system there may be several disks attempting to transfer blocks of data to or from the CPU simultaneously. A DVD attempting to present a full screen video at movie rates without dropouts must provide data steadily at input rates approaching 10 megabytes per second, with some transient rates and high definition video rates even higher. In addition, video and audio devices require a steady stream of data over long periods of time. Contrast this requirement with the occasional bursts of data that are characteristic of most I/O devices.

For both disk and image I/O, therefore, the computer must be capable of transferring massive amounts of data very quickly between the CPU and the disk(s) or image devices. Clearly, executing a single instruction for each byte of data is unacceptable for disk and image I/O, and a different approach must be used. Furthermore, you can see the importance of providing a method to allow utilization of the CPU for other tasks while these large I/O operations are taking place.

With the rapid proliferation of networks in recent years, network interfaces have also become an important source of I/O. From the perspective of a computer, the network is just another I/O device. In many cases, the network is used as a substitute for a disk, with the data and programs stored at a remote computer and served to the local station. For the computer that is acting as a server, there may be a massive demand for I/O services. User interfaces such as X Windows, which allow the transfer of graphical information from a computer to a display screen located elsewhere on the network, place heavy demands on I/O capability. With simple object graphics, or locally stored bitmap images, and with a minimal requirement for large file transfers, a small computer with a modem may operate sufficiently at I/O transfer rates of 3,000 bytes per second, but computers with more intensive requirements may require I/O transfer rates of 50 megabytes per second, or more.

A table of typical data rates for various I/O devices appears in Figure 9.1. The values given are rough approximations, since the actual rates are dependent on the particular hardware systems, software, and application. As computer technology advances, the high end data rates continue to increase at a rapid pace. For example, local area networks operating at 1 gigabit (or, equivalently, 125 megabytes) per second are increasingly common.

It should be pointed out that disks, printers, screens, and most other I/O devices operate almost completely under CPU program control. Printers and screens, of course, are strictly output devices, and the output produced can be determined only by the program being executed. Although disks act as both input and output devices, the situation is similar. It is the executing program that must always determine what file is to be read on input, or where to store output. Therefore, it is always a program executing in the CPU that initiates I/O data transfer, even if the CPU is allowed to perform other tasks while waiting for the particular I/O operation to be completed.

FIGURE 9.1

Examples of I/O Devices Categorized by a Typical Data Rate

Device	Input/Output	Data rate	Type
Keyboard	Input	100 bps	char
Mouse	Input	3800 bps	char
Voice input/output	Input/Output	264 Kbps	block burst
Sound input	Input	3 Mbps	block burst or steady
Scanner	Input	3.2 Mbps	block burst
Laser printer	Output	3.2 Mbps	block burst
Sound output	Output	8 Mbps	block burst or steady
Flash drive	Storage	480-800 Mbps read; 80 Mbps write	block burst
USB	Input or output	1.6-480 Mbps	block burst
Network/Wireless LAN	Input or output	11-100 Mbps	block burst
Network/LAN	Input or output	100-1000 Mbps	block burst
Graphics display	Output	800-8000 Mbps	block burst or steady
Optical disk	Storage	4-400 Mbps	block burst or steady
Magnetic tape	Storage	32-90 Mbps	block burst or steady
Magnetic disk	Storage	240-3000 Mbps	block burst

Adapted from Patterson, David A. and John L. Hennessy, (2005), *Computer Organization and Design, 3rd Edition*, Morgan Kaufmann Publishers, Inc, San Fransisco, CA

Some input devices must be capable of generating input to the CPU independent of program control. The keyboard and mouse were mentioned earlier in this context, and voice input would also fall into this category. Some devices, such as CD-ROMs and USB devices, can self-initiate by signaling their presence to a program within the operating system software. Local area networks can also generate this kind of input, since a program on a different CPU might request, for example, a file stored on your disk. In a slightly different category, but with similar requirements, are input devices for which input is under program control, but for which the time delay until arrival of the data is unpredictable, and possibly long. (You might consider regular keyboard input in this category, especially when writing a paper using your word processor.) This would be true if the data is being telemetered from some sort of measurement device. For example, the computer might be used to monitor the water level at a reservoir, and the input is water-level data that is telemetered by a measurement device once per hour. Provision must be made to accept unpredictable input and process it in some reasonable way, preferably without tying up the CPU excessively.

Additionally, there will be situations where an I/O device being addressed is busy or not ready. The most obvious examples are a printer that is out of paper or a DVD drive with no disk in it or a hard disk that is processing another request. It would be desirable for the device to be able to provide status information to the CPU, so that appropriate action can be taken.

The discussion in this section establishes several requirements that will have to be met for a computer system to handle I/O in a sufficient and effective manner:

1. There must be a means for individually addressing different peripheral devices.
2. There must be a way in which peripheral devices can initiate communication with the CPU. This facility will be required to allow the CPU to respond to unexpected inputs from peripherals such as keyboards, mice, and networks, and so that peripherals such as printers and floppy disk drives can convey emergency status information to the executing program.
3. Programmed I/O is suitable only for slow devices and individual word transfers. For faster devices with block transfers, there must be a more efficient means of transferring the data between I/O and memory. Memory is a suitable medium for direct block transfers, since the data has to be in memory for a program to access it. Preferably this could be done without involving the CPU, since this would free the CPU to work on other tasks.
4. The buses that interconnect high-speed I/O devices with the computer must be capable of the high data transfer rates characteristic of modern systems. We will return to this issue in Chapter 11.
5. Finally, there must be a means for handling devices with extremely different control requirements. It would be desirable if I/O for each of these devices could be handled in a simple and similar way by programs in the CPU.

The last requirement suggests that it is not practical to connect the I/O devices directly to the CPU without some sort of interface module unique to each device. To clarify this requirement, note the following conditions established from the previous discussion:

1. The formats required by different devices will be different. Some devices require a single piece of data, and then must wait before another piece of data can be accepted. Others expect a block of data. Some devices expect 8 bits of data at a time; others require 16, 32, or 64. Some devices expect the data to be provided sequentially, on a single data line. Other devices expect a parallel interface. These inconsistencies mean that the system would require substantially different interface hardware and software for each device.
2. The incompatibilities in speed between the various devices and the CPU will make synchronization difficult, especially if there are multiple devices attempting to do I/O at the same time. It may be necessary to buffer the data (i.e., hold it and release part of it at particular times) to use it. A **buffer** works something like a water reservoir or tower. Water enters the reservoir or tower as it becomes available. It is stored and released as it can be used. A computer buffer uses registers or memory in the same way.
3. Although the I/O requirements for most devices occur in bursts, some multimedia, video and audio in particular, provide a steady stream of data that must be transferred on a regular basis to prevent dropouts that can upset a user. I/O devices and the interconnections that support multimedia services must be capable of guaranteeing steady performance. This often includes network interfaces and high-speed communication devices as well as such devices as video

cameras, since networks are frequently used to supply audio and video. (Think of downloading streaming video from the Web.)

4. Devices such as disk drives have electromechanical control requirements that must be met, and it would tie up too much time to use the CPU to provide that control. For example, the head motors in a disk drive have to be moved to the correct disk track to retrieve data and something must continually maintain the current head position on the track once the track is reached. There must be a motor controller to move the print heads in an inkjet printer across the paper to the correct position to print a character. And so on. Of course, the requirements for each device are different.

The different requirements for each I/O device plus the necessity for providing devices with addressing, synchronization, status, and external control capabilities suggest that it is necessary to provide each device with its own special interface. Thus, in general, I/O devices will be connected to the CPU through an I/O module of some sort. The I/O module will contain the specialized hardware circuits necessary to meet all the I/O requirements that we established, including block transfer capability with appropriate buffering and a standardized, simple interface to the CPU. At the other interface, the I/O module will have the capability to control the specific device or devices for which it is designed.

The simplest arrangement is shown in Figure 9.2. I/O modules may be very simple and control a single device, or they may be complex, with substantial built-in intelligence, and may control many devices. A slightly more complex arrangement is shown in Figure 9.3. The additional I/O modules require addressing to distinguish them from each other. The lower module will actually recognize addresses for either of the I/O devices connected to it. I/O modules that control a single type of device are often called **device controllers**. For example, an I/O module that controls disks would be a *disk controller*. We look at the I/O modules more carefully in Section 9.5.

FIGURE 9.2

Simple I/O Configuration

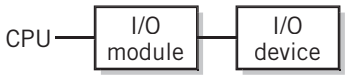
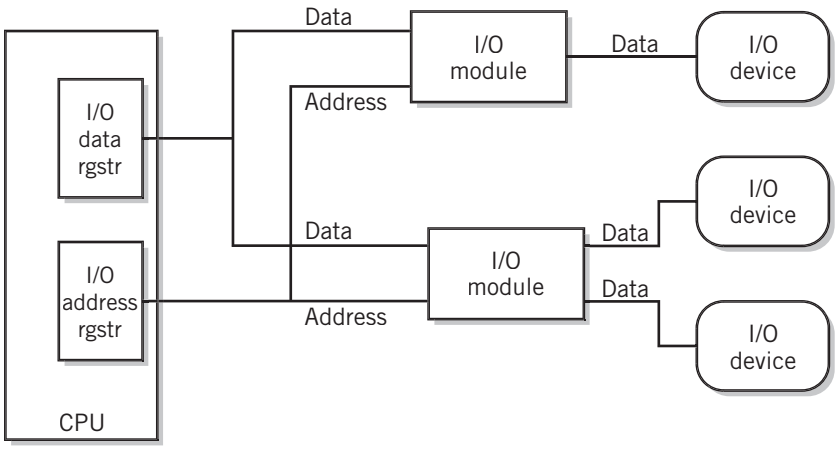


FIGURE 9.3

A Slightly More Complex I/O Module Arrangement



9.2 PROGRAMMED I/O

In the simplest method for performing I/O, an I/O module is connected to a pair of I/O registers in the CPU via a bus. The I/O data register serves the same role in the real CPU as the input and output baskets served in the Little Man Computer. Alternatively, one might view the I/O baskets as buffers, holding multiple inputs or outputs, with the I/O data register as the interface between the CPU and the buffer. The I/O operation is similar to that of the Little Man Computer. Input from the peripheral device is transferred from the I/O module or buffer for that peripheral device one word at a time to the I/O data register and from there to an accumulator register under program control, just as occurred in the Little Man Computer. Similarly, individual words of output data pass from an accumulator register to the I/O data register where they can be read by the appropriate I/O module, again under program control. Each instruction produces a single input or output. This method is known as **programmed I/O**.

In practice, it is most likely that there will be multiple devices connected to the CPU. Since each device must be recognized individually, address information must be sent with the I/O instruction. The address field of the I/O instruction can be used for this purpose. An I/O address register in the CPU holds the address for transfer to the bus. Each I/O module will have an identification address that will allow it to identify I/O instructions addressed to it and to ignore other I/O not intended for it.

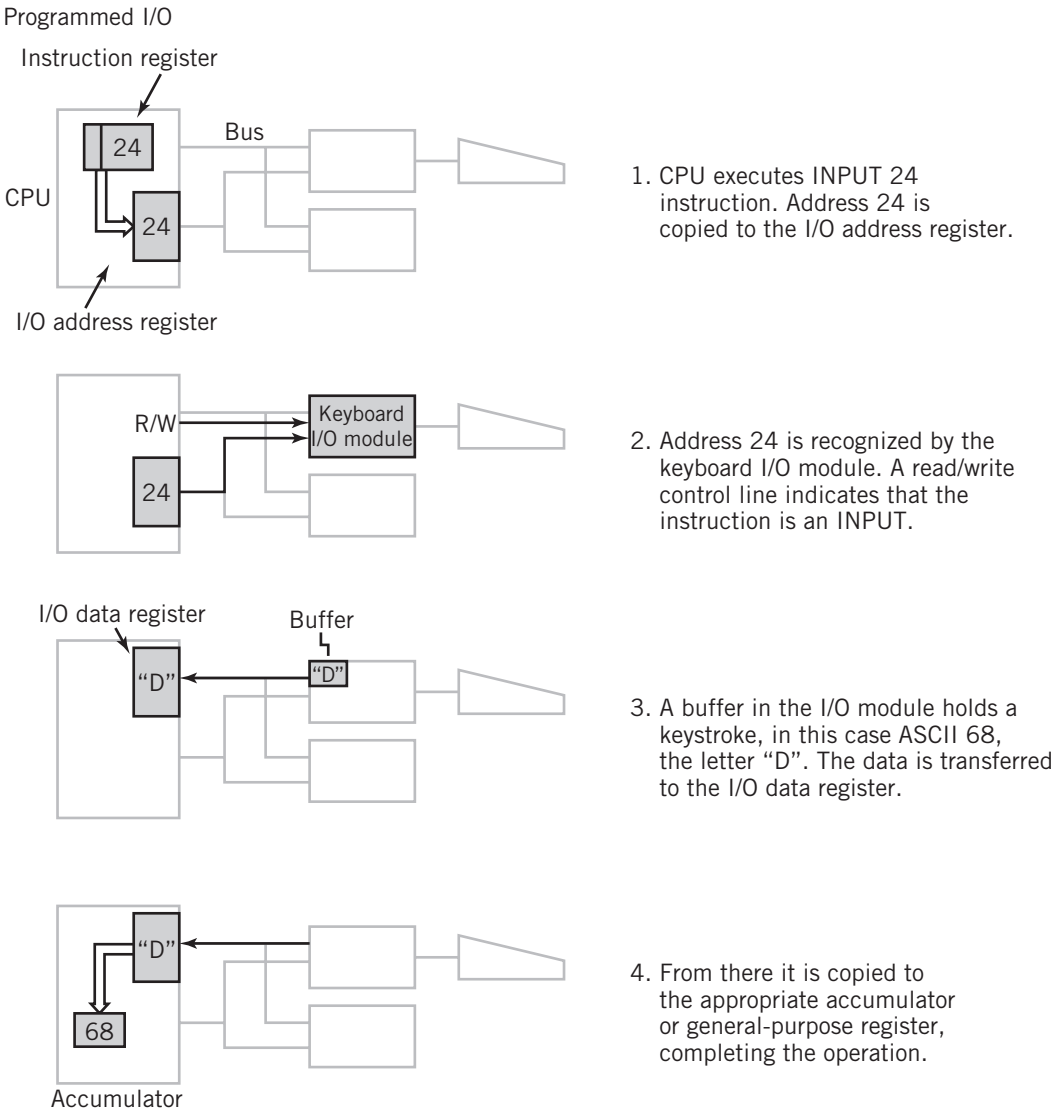
As has been noted, it is common for an I/O module to have several addresses, each of which represents a different control command or status request, or which addresses a different device when a particular module supports multiple devices. For example, the address field in the Little Man input and output instructions could be used to address up to a combination of one hundred devices, status requests, or control commands. Figure 9.4 illustrates the concept of programmed I/O. Indeed, the LMC uses the address field to select the I-basket (901) or O-basket (902) as the I/O device within the 900 instruction.

The I/O data and address registers work similarly to the memory address register (MAR) and memory data register (MDR). In fact, in some systems, they may even be connected to the same bus. The CPU places a control signal on the bus to indicate whether the transfer is I/O or memory.

Programmed I/O is obviously slow, since a full instruction fetch-execute cycle must be performed for each and every I/O data word to be transferred. Programmed I/O is used today primarily with keyboards, with occasional application to other simple character based data transfers, such as the transmission of commands through a network I/O module or modem. These operations are slow compared with the computer, with small quantities of data that can be handled one character at a time. One limitation, which we shall address later in the chapter, is that with programmed I/O, input from the keyboard is accepted only under program control. An alternative means must be found to accept unexpected input from the keyboard.

There is one important application for programmed I/O: alternative methods of I/O use the I/O module to control I/O operations from outside the CPU, independent of the CPU, using memory as the intermediate site for the data transfer. Programmed I/O is used by programs in the CPU to send the necessary commands to the I/O modules to set up parameters for the transfer and to initiate I/O operations. We shall return to this topic in Section 9.4.

FIGURE 9.4



9.3 INTERRUPTS

As you know from our previous discussion, there are many circumstances under which it is important to interrupt the normal flow of a program in the computer to react to special events. An unexpected user command from the keyboard or other external input, an abnormal situation, such as a power failure, that requires immediate attention from the computer, an attempt to execute an illegal instruction, a request for service from a network

controller, or the completion of an I/O task initiated by the program: all of these suggest that it is necessary to include some means to allow the computer to take special actions when required. Interrupt capabilities are also used to make it possible to time share the CPU between several different programs or program segments at once.

Modern computers provide interrupt capability by providing one or more special control lines to the central processor known as **interrupt lines**. For example, the standard I/O for a modern PC may contain as many as thirty-two interrupt lines, labeled IRQ0 through IRQ31. (IRQ stands for Interrupt ReQuest.) The messages sent to the computer on these lines are known as **interrupts**. The presence of a message on an interrupt line will cause the computer to suspend the program being executed and jump to a special interrupt processing program.

Consider, as an example, the following situation:

EXAMPLE

In a large, multiuser system there may be hundreds of keyboards being used with the computer at any given time. Since any of these keyboards could generate input to the computer at any time, it is necessary that the computer be aware of any key that is struck from any keyboard in use. This process must take place quickly, before another key is struck on the same keyboard, to prevent data loss from occurring when the second input is generated.

Theoretically, though impractically, it would be possible for the computer to perform this task by checking each keyboard for input in rotation, at frequent intervals. This technique is known as **polling**. The interval would have to be shorter than the time during which a fast typist could hit another key. Since there may be hundreds of keyboards in use, this technique may result in a polling rate of thousands of samples per second. Most of these samples will not result in new data; therefore, the computer time spent in polling is largely wasted.

This is a situation for which the concept of the interrupt is well suited. The goal is achieved more productively by allowing the keyboard to notify the CPU by using an interrupt when it has input. When a key is struck on any keyboard, it causes the interrupt line to be activated, so that the CPU knows that an I/O device connected to the interrupt line requires action. Interrupts satisfy the requirement for external input controls, and also provide the desirable feature of freeing the CPU from waiting for events to occur.

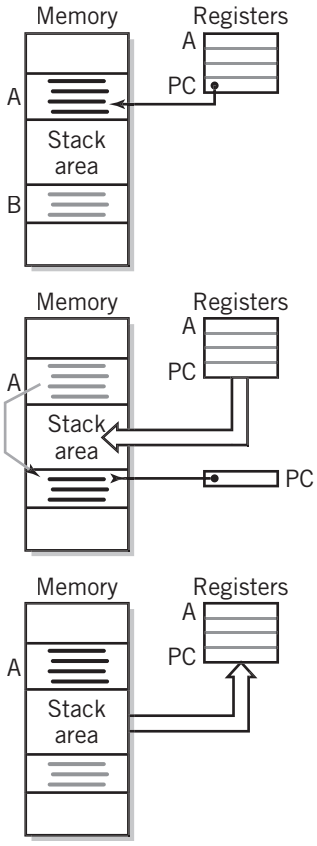
Servicing Interrupts

Since the computer is capable only of executing programs, interrupt actions take the form of special programs, executed whenever triggered by an interrupt signal. Interrupt procedures follow the form shown in Figure 9.5.

Specifically, the interrupt causes the temporary suspension of the program in progress. All the pertinent information about the program being suspended, including the location of the last instruction executed, and the values of data in various registers, is saved in a known part of memory, either in a special area associated with the program, known as the **process control block** (**PCB**), or in a part of memory known as the stack area. This information is known as the program's **context**, and will make it possible to restart the program exactly where it left off, without loss of any data or program state. Many computers have a single instruction that saves all the critical information at once. The

FIGURE 9.5

Servicing an Interrupt



1. Before interrupt arrives, program A is executing. The program counter points to the current instruction.
2. When the interrupt is received by the CPU, the current instruction is completed, all the registers are saved in the stack area (or in a special area known as a process control block). The PC is loaded with the starting location of program B, the interrupt handler program. This causes a jump to program B, which becomes the executing program.
3. When the interrupt routine is complete, the registers are restored, including the program counter, and the original program resumes exactly where it left off.

memory belonging to the original program is kept intact. The computer then branches to a special interrupt handler program elsewhere in memory; the **interrupt handler** program is also known as an **interrupt routine**. The interrupt handler program determines the appropriate course of action. This process is known as **servicing the interrupt**. Since many interrupts exist to support I/O devices, most of the interrupt handling programs are also known as **device drivers**.

When the interrupt routine completes its task, it normally would return control to the interrupted program, much like a subroutine. Original register values would be restored, and the original program would resume execution *exactly* where it left off, and in its identical state, since all the registers were restored to their original values. There are some circumstances when this is not the case, however, since actions taken by the interrupt routine may make a difference in what the original program is supposed to do. For example, a printer interrupt indicating that the printer is out of paper would require a different action by the original program (perhaps a message to the screen telling the user to load more paper); it would not be useful for the program to send more characters!

Intuitively, the servicing of interrupts works just the way that you would expect. Suppose that you were giving a speech in one of your classes, and someone in the class interrupts you with a question. What do you do? Normally, you would hold your current thought and answer the question. When you finish answering the question, you return to your speech just where you left off, pick up the thought, and continue as though no interrupt had occurred. This would be your normal interrupt servicing routine. Suppose, however, that the interrupt is the bell ending class or the instructor telling you that you have run out of time. In this case, your response is different. You would *not* return to your speech. Instead, you might do a quick wrap-up followed by an exit.

In other words, you would react in a way quite similar to the way in which the interrupt servicing routines work.

The Uses of Interrupts

The way in which an interrupt is used depends on the nature of the device. You've already seen that externally controlled inputs are best handled by generating interrupts whenever action is required. In other cases, interrupts occur when some action is *completed*. This section introduces several different ways in which interrupts are used.

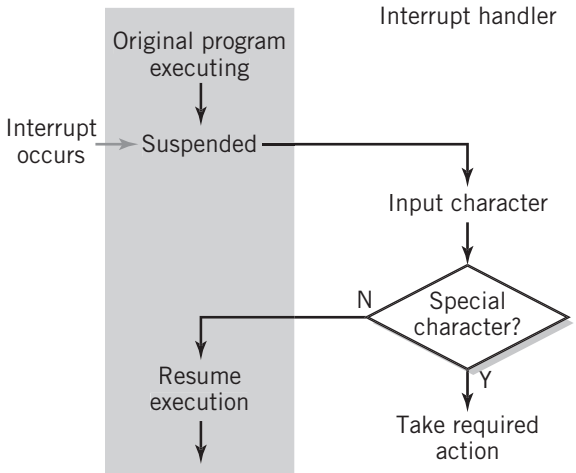
THE INTERRUPT AS AN EXTERNAL EVENT NOTIFIER As previously discussed, interrupts are useful as notifiers to the CPU of **external events** that require action. This frees the CPU from the necessity of performing polling to determine that input data is waiting.

EXAMPLE

Keyboard input can be processed using a combination of programmed I/O and interrupts. Suppose a key is struck on the keyboard. This causes an interrupt to occur. The current program is suspended, and control is transferred to the keyboard interrupt handler program.

FIGURE 9.6

Using a Keyboard Handler Interrupt



The keyboard interrupt handler first inputs the character, using programmed I/O, and determines what character has been received. It would next determine if the input is one that requires special action. If so, it would perform the required action, for example, suspending the program or freezing the data on the screen. Otherwise, it would pass the input data to the program expecting input from that keyboard. Normally, the input character would be stored in a known memory location, ready for the program to use when it is reactivated.

When the action is complete, that is, when the interrupt has been serviced, the computer normally restores the register values and returns control to the suspended program, unless the interrupt request specifies a different course of action. This would be the case, for example, if the user typed a command to suspend the program being run.

Figure 9.6 shows the steps in processing a keyboard input interrupt.

EXAMPLE

A real-time system is a computer system used primarily to measure external events that happen in ''real time''; that is, the event, when it occurs, requires processing quickly because the data is of critical time-sensitive value.

As an example, consider a computer system that monitors the coolant temperature from the core of a power plant nuclear reactor. The temperature is transmitted once a minute by a temperature measurement transducer to the computer.

In this particular case, the transducer input is expected, and, when it occurs, requires immediate evaluation. It is reasonable to assume, however, that the computer system is to be used for other purposes, and it is not desirable to tie up the CPU in an input loop waiting for the transducer data to arrive.

This is a perfect application for interrupts. The transducer input is assigned to an interrupt. The interrupt service routine in this case is used to process the transducer input data. When the interrupt occurs, the interrupt routine evaluates the input. If everything is normal, the routine returns control to whatever the computer was doing. In an emergency, the interrupt routine would transfer control instead to the program that handles emergency situations.

THE INTERRUPT AS A COMPLETION SIGNAL The keyboard and transducer examples demonstrate the usefulness of the interrupt as a means for the user to control the computer from an input device, in this case the keyboard or transducer. Let us next consider the interrupt technique as a means of controlling the flow of data to an output device. Here, the interrupt serves to notify the computer of the completion of a particular course of action.

EXAMPLE

As noted previously, the printer is a slow output device. The computer is capable of outputting data to the printer much faster than the printer can handle it. The interrupt can be used to control the flow of data to the printer in an efficient way.

The computer sends one block of data at a time to the printer. The size of the block depends on the type of printer and the amount of memory installed in the printer. When the printer is ready to accept more data, it sends an interrupt to the computer. This interrupt indicates that the printer has completed printing the material previously received and is ready for more.

In this case, the interrupt capability prevents the loss of output, since it allows the printer to control the flow of data to a rate that the printer can accept. Without the interrupt capability, it would be necessary to output data at a very slow rate to assure that the computer did not exceed the ability of the printer to accept output. The use of an interrupt also allows the CPU to perform other tasks while it waits for the printer to complete its printing.

By the way, you might notice that the printer could use a second, different interrupt as a way of telling the computer to stop sending data temporarily when the printer's buffer fills up.

This application is diagrammed in Figure 9.7. Another application of the interrupt as a completion signal is discussed in Section 9.4, as an integral part of the direct memory access technique.

THE INTERRUPT AS A MEANS OF ALLOCATING CPU TIME A third major application for interrupts is to use the interrupt as a method of allocating CPU time to different programs or threads that are sharing the CPU. (Threads are small pieces of a program that can be executed independently, such as the spell checker in a word processing program.)

Since the CPU can only execute one sequence of instructions at a time, the ability to time share multiple programs or threads implies that the computer system must share the CPU by allocating small segments of time to each program or thread, in rapid rotation among them. Each program sequence is allowed to execute some instructions. After a certain period of time, that sequence is interrupted and relinquishes control to a dispatcher program within the operating system that allocates the next block of time to another sequence. This is illustrated in Figure 9.8.

FIGURE 9.7

Using a Print Handler Interrupt

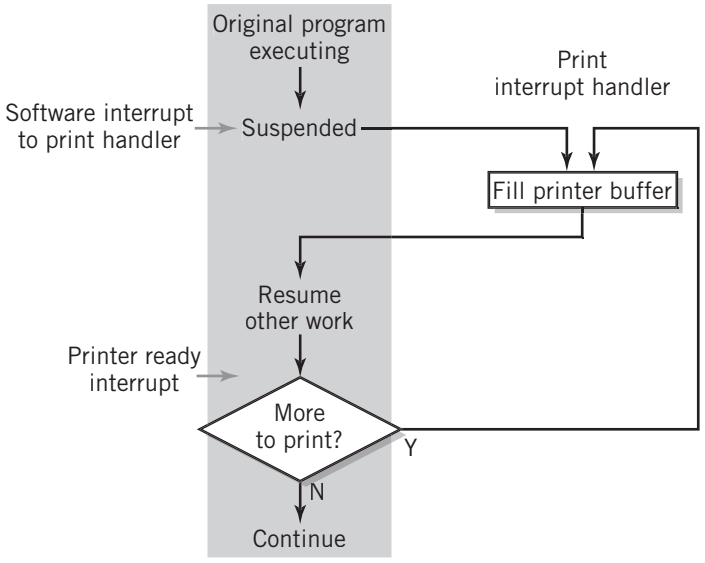
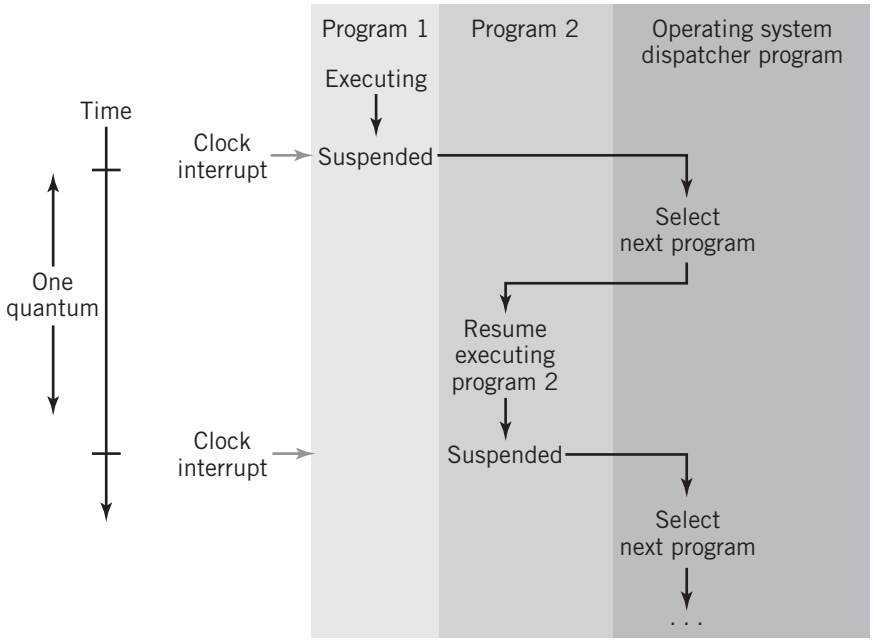


FIGURE 9.8

Using an Interrupt for Time Sharing



The system cannot count on an instruction sequence relinquishing control voluntarily, since a program caught in an infinite loop would not be able to do so. Instead, the computer system provides an internal clock that sends an interrupt periodically to the CPU. The time between interrupt pulses is known as a **quantum**, and represents the time that each program or thread will have allotted to it. When the clock interrupt occurs, the interrupt routine returns control to the operating system, which then determines which program or thread will receive CPU time next. The interrupt is a simple but effective method for allowing the operating system to share CPU resources among several programs at once.

Time sharing is discussed in more depth in Chapters 15 and 18.

THE INTERRUPT AS AN ABNORMAL EVENT INDICATOR The fourth major use for interrupts is to handle **abnormal events** that affect operation of the computer system itself. Under certain conditions, we would like the computer to respond with a specific course of action, quickly and effectively. This usage is similar to that of other external input events, but in this case, the events are directed at problems or special conditions within the computer system itself.

One obvious example of an external event requiring special computer action is power failure. Most computers provide enough internal power storage to save the work that is being performed and to shut down gracefully, provided that the computer has quick notification of the power failure. A power line monitor that connects to the interrupt facility provides this capability. The interrupt routine will save the status of programs that

are in memory, close open files, and perform other housekeeping operations that will allow the computer to restart without any loss of data. It will then halt the computer.

Another important application of the abnormal event interrupt is when a program attempts to execute an illegal instruction such as a *divide by 0* or a nonexistent op code, or when a hardware error is detected, such as a memory parity error. When the error occurs it is not possible to complete the executing program. Yet it is important that the system attempt to recover from the error and that the appropriate personnel be notified. It is not acceptable simply to halt the computer. Particularly in modern multitasking computer systems this would be undesirable since it would also stop other executing programs that might not be affected by the error and would affect other users if the system is a multiuser system. Instead, an interrupt routine can notify the user of the error and return control of the CPU to the operating system program. You should notice that these interrupts are actually generated from inside the CPU, whereas the other interrupts that we have discussed so far are generated externally. Internal interrupts are sometimes called **traps** or **exceptions**.

EXAMPLE

Most modern computers have a set of instructions known as **privileged instructions**. These instructions are intended for use by an operating system program. The HALT instruction generally is a privileged instruction. Privileged instructions are designed to provide system security by preventing application programs from altering memory outside their own region, from stopping the computer, or from directly addressing an I/O device that is shared by multiple programs or users. (Suppose, for example, that two programs sharing the computer each sent text out to a printer. The resulting printout would be garbage, a mixture of the outputs from each program.) An attempt by a user's program to execute a privileged instruction would result in an illegal instruction interrupt.

You might assume from the examples above that abnormal event interrupts always result from critical errors or catastrophic failures within the computer system, but this not necessarily the case. *Virtual storage* is a memory management technology that makes it appear that a computer system has more memory than is physically installed in the computer. (There is a detailed discussion of virtual storage in Chapter 18.) One particularly important interrupt event occurs as an integral part of the design and operation of virtual storage. Other internal and external events also make use of the interrupt facility. The table in Figure 9.9 shows a list of the built-in interrupts for the IBM System z family of computers.

SOFTWARE INTERRUPTS In addition to the actual hardware interrupts already discussed, modern CPU instruction sets include an instruction that simulates an interrupt. In the Intel x86 architecture, for example, this instruction has the mnemonic INT, for INTerrupt. The IBM System z uses the mnemonic SVC for SUPERVISOR CALL. The interrupt instruction works in the same way as a hardware interrupt, saving appropriate registers and transferring control to an interrupt handling procedure. The address space of the INT instruction can be used to provide a parameter that specifies which interrupt is to be executed. The **software interrupt** is very similar to a subroutine jump to a known, fixed location.

FIGURE 9.9

Table of Interrupts for zSeries Family

Priority	Interrupt class	Type of interrupts
Highest	Machine check	Nonrecoverable hardware errors
	Supervisor call	Software interrupt request by program
	Program check	Hardware-detectible software errors: illegal instruction, protected instruction, divide by 0, overflow, underflow, address translation error
	Machine check	Recoverable hardware errors
	External	Operator intervention, interval timer expiration, set timer expiration
	I/O	I/O completion signal or other I/O-related event
Lowest	Restart	Restart key, or restart signal from another CPU when multiple CPUs are used

Software interrupts make the interrupt routines available for use by other programs. Programs can access these routines simply by executing the INT instruction with the appropriate parameter.

An important application for software interrupts is to centralize I/O operations. One way to assure that multiple programs do not unintentionally alter another program's files or intermingle printer output is to provide a single path for I/O to each device. Generally, the I/O paths are interrupt routines that are a part of the operating system software. Software interrupts are used by each program to request I/O from the operating system software. As an example, a software interrupt was used in Figure 9.7 to initiate printing.

Multiple Interrupts and Prioritization

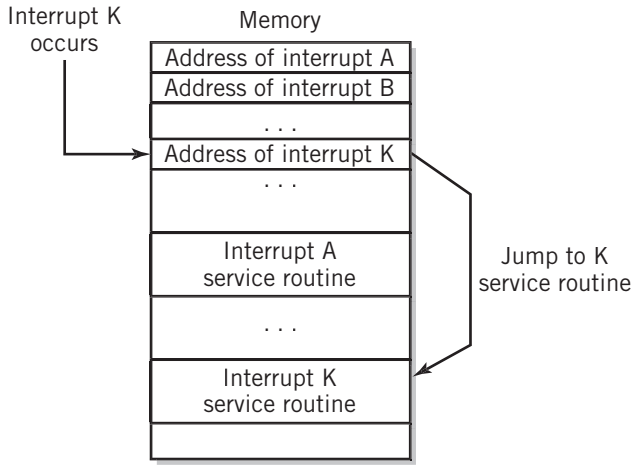
As you have now seen, there may be many different input and output devices and event indicators connected to interrupt lines. This means that there may be many different events vying for attention. Inevitably, multiple interrupts will occur from time to time.

There are two questions that must be answered when an interrupt occurs. First, are there other interrupts already awaiting service, and, if so, how does the computer determine the order in which the interrupts get serviced? And, second, how does the computer identify the interrupting device?

Two different processing methods are commonly used for determining which device initiated the interrupt. Some computers use a method known as **vectored interrupt**, in which the address of the interrupting device is included as part of the interrupt. Another method provides a general interrupt that is shared by all devices. The computer identifies the interrupting device by **polling** each device. These two methods are illustrated in Figures 9.10 and 9.11, respectively. The vectored interrupt method is obviously faster, but requires additional hardware to implement. Some systems use different interrupt lines for each interrupt; others use a method called "daisy chaining," which places the interrupts onto a single interrupt line to the CPU in such a way that highest priorities are recognized first.

FIGURE 9.10

Vectored Interrupt Processing



Multiple interrupts can be handled by assigning **priorities** to each interrupt. In general, multiple interrupts will be handled top priority first. A higher-priority interrupt will be allowed to interrupt an interrupt of lower priority, but a lower-priority interrupt will have to wait until a higher-priority interrupt is completed.

This leads to a hierarchy of interrupts, in which higher-priority interrupts can interrupt other interrupts of lower priority, back and forth, eventually returning control to the original program that was running. Although this sounds complicated, this situation is actually quite common, and is fairly easy to implement. Figure 9.12 shows a simple example of this situation. In this figure, interrupt routine C is the highest priority, followed by B and A.

Most computer systems allow the system manager to establish priorities for the various interrupts. Priorities are established in a logical way. The highest priorities are reserved for time-sensitive situations, such as power failure or external events that are being time measured. Keyboard events are also usually considered high-priority events, since data loss can occur if the keyboard input is not read quickly. Task completion interrupts usually take lower priorities, since the delay will not affect the integrity of the data under normal conditions.

Depending on the system, priorities may be established with software or with hardware. In some systems, the priority of I/O device interrupts is established by the way their I/O module cards are physically placed on the backplane. The daisy chain interrupt line can

FIGURE 9.11

Polled Interrupt Processing

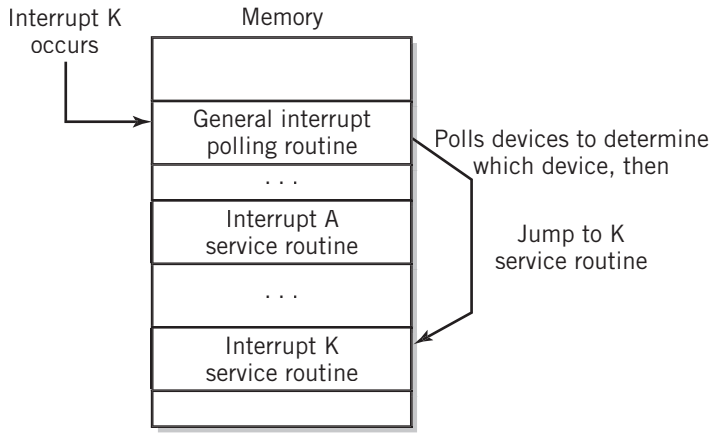
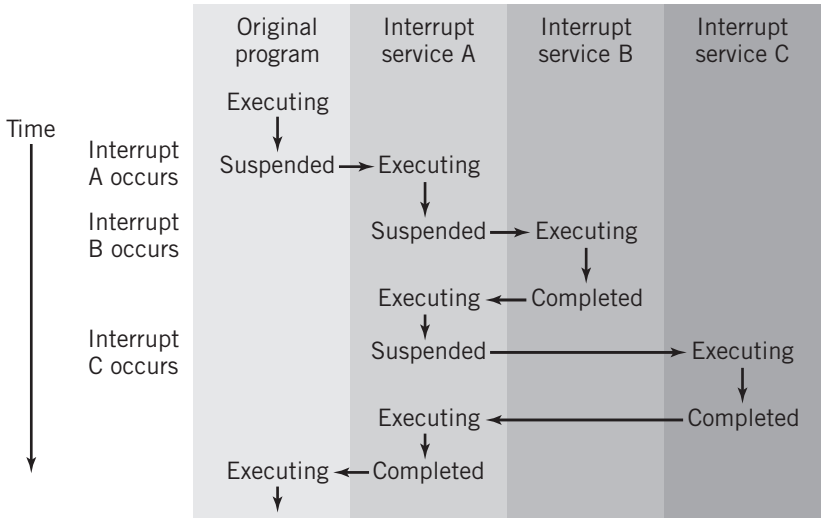


FIGURE 9.12

Multiple Interrupts



be used for this purpose: the highest-priority devices are placed closest to the CPU and block the signals of lower-priority devices that are farther down the line. In other systems, priorities are established by assigning a priority number to each interrupt.

Most interrupts can be temporarily disabled by program instructions when a program is performing a critical task that would be negatively affected if an interrupt were to occur. This is particularly true of time-sensitive tasks. In many systems, interrupts are **maskable**; that is, they can be selectively disabled. Certain interrupts, such as power failure, that are never disabled are sometimes referred to as *nonmaskable interrupts*. Most modern computer systems save interrupts that occur when interrupts are disabled, so that when the interrupts are reenabled, the pending interrupts will be processed.

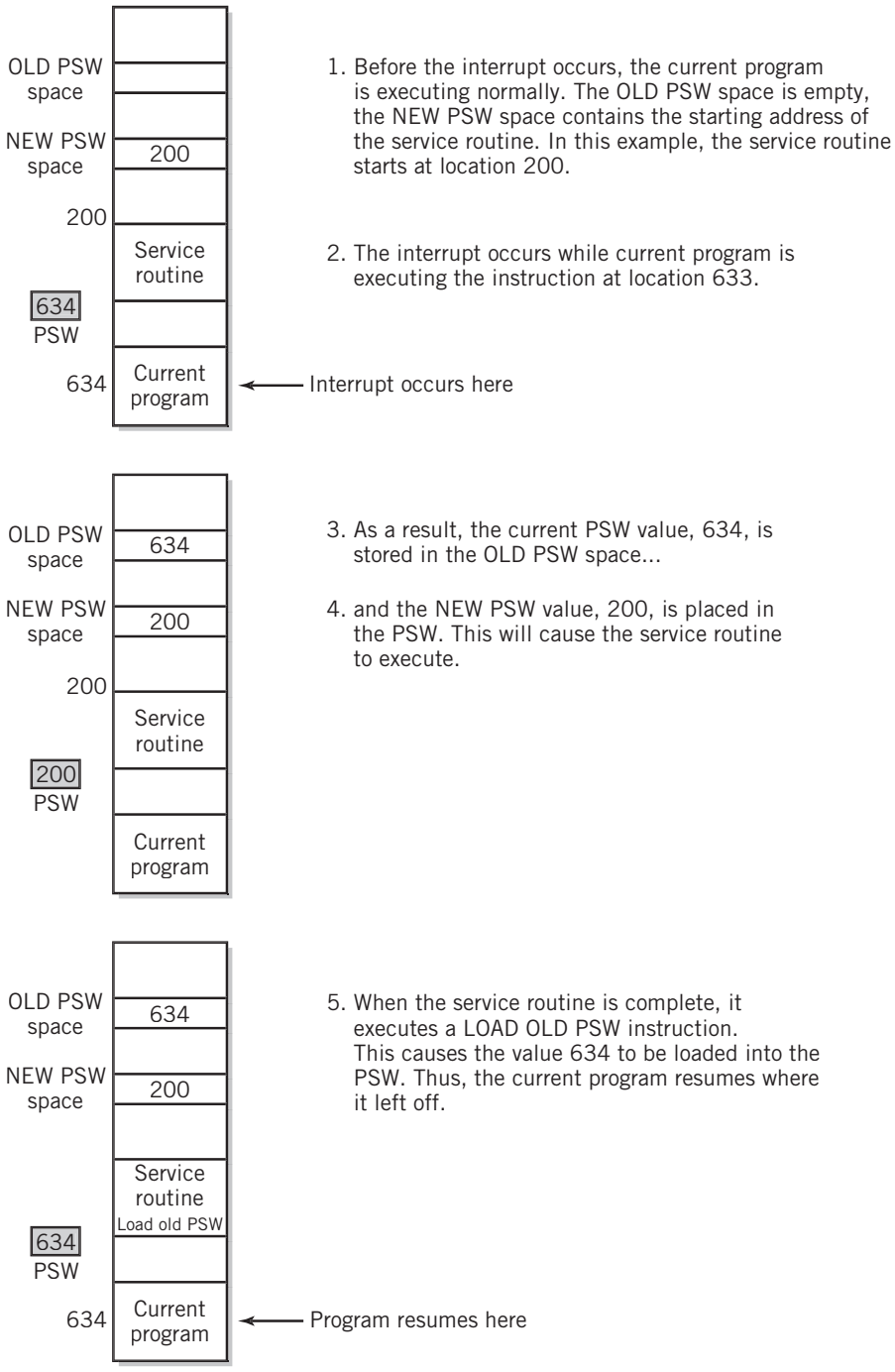
EXAMPLE

In the IBM System z architecture, interrupts are divided into six classes, with the priorities shown in Figure 9.9. All the different interrupts within each class are handled by the interrupt service routine for that class. Each interrupt class has two vectored addresses permanently associated with it. The first of these is a space reserved for the Program Status Word of the current program, known in IBM jargon as the OLD PSW. The Program Status Word is a 64-bit word that includes the program counter and other significant information about the program. The second vectored address contains a pointer to the interrupt routine. This address is known as the NEW PSW. The method used to switch from the original program to a service routine and back is illustrated in Figure 9.13.

When an interrupt occurs, the PSW for the current program is stored automatically in the OLD PSW space, and the NEW PSW is loaded. The effect is a jump to the location in memory pointed to by the NEW PSW, which is the location of the interrupt service routine for that class of interrupts. Incidentally, note that this procedure does not save other registers. Each interrupt service routine saves and restores the registers that it uses.

FIGURE 9.13

Processing an Interrupt in the IBM zSeries



The Z-architecture CPU has an instruction LOAD OLD PSW. When the interrupt service routine is completed, it simply uses this instruction to return to the original program. Interrupts of lower priority are masked, but higher-priority interrupts are allowed to interrupt the lower priority service routine while it is executing. Most important, interrupts belonging to the same class must be masked. Since there is only a single address space available for storing the OLD PSW for each class, a second interrupt of the same class would destroy the return address to the original program. Worse yet, the second interrupt would store the current address being executed in the OLD PSW space. Since that address is itself within the service routine, this would result in an infinite loop. To see this, look again at Figure 9.13. Pick a location inside the service routine, say, 205, and cause an interrupt to occur at that point. Now, follow through the diagram, and notice the results.

The preceding example demonstrates one way of providing return access from interrupts. An alternative method is used in x86 series computers. The x86 interrupt structure is also vectored, but the context is stored on an interrupt stack. Using a stack in this way is essentially analogous to the way in which subroutine jumps and returns work. These were discussed in detail in Chapter 7. Stack storage for interrupts makes interrupts reentrant, although such a condition would seem to be extremely rare.

Interrupts are normally checked at the completion of each instruction. That is, interrupts are normally checked *after* one instruction is finished and *before* another begins. This assures that conditions won't change *in the middle* of an instruction that would affect the instruction's execution. Certain long System z instructions can be interrupted in the middle of their fetch-execution cycle, however. These instructions use the general-purpose registers for their intermediate values, so it is important that the general-purpose registers be stored during an interrupt for later retrieval; otherwise, some instructions could not be restarted properly. The System z computer does not automatically store registers when an interrupt occurs. It is therefore important that the interrupt programs be written carefully so that the interrupted instruction doesn't crash when the routine is restarted. In the x86 computer, the registers are also generally stored on a stack, which makes retrieval simple even if the interrupt routine itself is interrupted. Virtual storage also requires the ability to interrupt in the middle of an instruction.

9.4 DIRECT MEMORY ACCESS

For most applications, it is impractical to transfer data to the CPU from a peripheral device using programmed I/O, even with interrupts. Indeed, the data from disks and tapes are transferred only in blocks, and it does not make sense to execute a separate instruction for each piece of data in the block. It is also more reasonable to transfer blocks of data directly between a device's I/O module and memory, since most processing will also take place in blocks. This suggests bypassing the CPU registers, if possible, and then processing the block of data as a group, from memory.

As a simple example, consider a program that sorts a block of numbers. To operate efficiently, the entire block of numbers must be stored in memory for the sort operation to take place, since instructions in the CPU can operate only on data in memory. Thus, it makes sense to move the entire block from disk to memory at once.

For this purpose, computer systems provide a more efficient form of I/O that transfers block data directly between the I/O module and computer memory, under control of the I/O module. The transfer is initiated by a program in the CPU, using programmed I/O, but the CPU can then be bypassed for the remainder of the transfer. The I/O module will notify the CPU with an interrupt when the transfer is complete. Once this has occurred, the data is in memory, ready for the program to use. This technique of I/O–memory data transfer is known as **direct memory access**, or more commonly, simply as **DMA**.

In Little Man terms, direct memory access could be viewed as providing data for the Little Man by loading data directly into the mailboxes through a rear door, bypassing the Little Man I/O instruction procedures. To reemphasize the fact that this operation only takes place under program control, we would have to provide a means for the Little Man to initiate such a transfer and a means to notify the Little Man when the data transfer is complete.

For direct memory access to take place, three primary conditions must be met:

1. There must be a method to connect together the I/O interface and memory. In some systems both are already connected to the same bus, so this requirement is easily met. In other cases, the design must contain provisions for interconnecting the two. The issue of system configuration is discussed in Chapter 11.
2. The I/O module associated with the particular device must be capable of reading and writing to memory. It does so by simulating the CPU's interface with memory. Specifically, the I/O module must be able to load a memory address register and to read and write to a memory data register, whether its own or one outside the I/O module.
3. There must be a means to avoid conflict between the CPU and the I/O module. It is not possible for the CPU and a module that is controlling disk I/O to load different addresses into the MAR at the same instant, for example, nor is it possible for two different I/O modules to transfer data between I/O and memory on the same bus at the same instant. This requirement simply means that memory can only be used by one device at a time, although, as we mentioned in Chapter 8, Section 8.3, some systems interleave memory in such a way that the CPU and I/O modules can access different parts of memory simultaneously. Special control circuits must be included to indicate which part of the system, CPU or particular I/O module, is in control of the memory and bus at any given instant.

DMA is particularly well suited for high-speed disk transfers, but there are several other advantages as well. Since the CPU is not actively involved during the transfer, the CPU can be used to perform other tasks during the time when I/O transfers are taking place. This is particularly useful for large multiuser systems. Of course, DMA is not limited to just disk-to-memory transfers. It can be used with other high-speed devices. And the transfers may be made in either direction. DMA is an effective means to transfer video data from memory to the video I/O system for rapid display, for example.

The procedure used by the CPU to initiate a DMA transfer is straightforward. Four pieces of data must be provided to the I/O controller for the particular I/O device to initiate the transfer. The four pieces of data that the I/O module must have to control a DMA transfer are

1. The location of the data on the I/O device (for example, the location of the block on the disk)
2. The starting location of the block of data in memory
3. The size of the block to be transferred
4. The direction of transfer, read (I/O → memory) or write (memory → I/O)

Normally, the I/O module would have four different registers, each with its own I/O address available for this purpose. In most modern systems, normal programmed I/O output instructions are used to initiate a DMA transfer. On some systems, a fifth programmed I/O instruction actually initiates the transfer, whereas other systems start the DMA transfer when the fourth piece of data arrives at the I/O module.

IBM mainframes work a bit differently, although the principle is the same. A single programmed I/O START CHANNEL instruction initiates the process. A separate **channel program** is stored in memory. The I/O module uses this channel program to perform its DMA control. The four pieces of data are a part of the channel program and are used by the I/O module to initiate the DMA transfer. The concept of I/O channels is considered in more detail in Chapter 11.

Once the DMA transfer has been initiated, the CPU is free to perform other processing. Note, however, that the data being transferred should not be modified during this period, since doing so can result in transfer errors, as well as processing errors.

If, for example, a program should alter the number in a memory location being transferred to disk, the number transferred is ambiguous, dependent on whether the alteration occurred before or after the transfer of that particular location. Similarly, the use of a number being transferred into memory depends on whether the transfer for that particular location has already occurred.

This would be equivalent to having the Little Man read a piece of data from the area of memory being loaded from the rear of the mailboxes. The number on that piece of data would depend on whether a new value loaded in from the rear came before or after the Little Man's attempt to read it. Clearly, this is not an acceptable situation.

It is thus important that the CPU know when the transfer is complete, assuring that the data in memory is stable. The interrupt technique is used for this purpose. The program waiting for the data transfer is suspended or performs other, unrelated processing during the time of transfer. The controller sends a completion signal interrupt to the CPU when the transfer is complete. The interrupt service routine notifies the program that it may continue with the processing of the affected data.

Finally, note that it takes several programmed output instructions to initiate a DMA transfer. This suggests, correctly, that it is not useful to perform a DMA transfer for very small amounts of data. For small transfers, it is obviously more efficient to use programmed I/O. It is also worth pointing out that if a computer is capable only of performing a single task, then the time freed up by DMA cannot be used productively, and there is little advantage in using DMA.

It is worth interrupting the discussion at this point (yes, the pun was intentional) to remind you that in reality an application program would not be performing I/O directly, since doing so might conflict with other programs that are also performing I/O at the same time. Instead, the application program would request I/O services from the operating system software by calling a procedure within the operating system that performs the I/O operations

described here. The I/O instructions and interrupt procedures are, of course, privileged: only the operating system software is allowed access to these instructions and procedures.

EXAMPLE

Consider the steps required to write a block of data to a disk from memory. The executing program has already created the block of data somewhere in memory.

First, the I/O service program uses programmed I/O to send four pieces of data to the disk-controlling I/O module: the location of the block in memory; the location where the data is to be stored on disk; the size of the block (this step might be unnecessary if a fixed disk size is always used on the particular system); and the direction of transfer, in this case a write to disk.

Next, the service program sends a ''ready'' message to the I/O module, again using programmed I/O. At this point, the DMA transfer process takes place, outside the control of the CPU, the I/O service, or the program that requested I/O service. Depending on the design of the operating system programs, the current application program may resume execution of other tasks, or it may be suspended until the DMA transfer is complete.

When the transfer is complete, the I/O module sends an interrupt to the CPU. The interrupt handler either returns control to the program that initiated the request or notifies the operating system that the program can be resumed, depending on the design of the system.

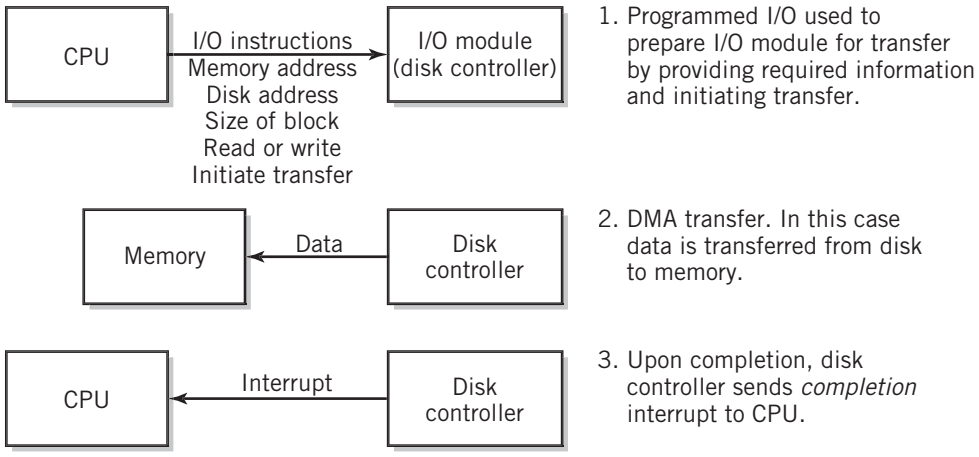
This example shows how the programmed I/O, DMA, and interrupt methodologies work together in the most important and common way of doing I/O. The technique is diagrammed in Figure 9.14.

9.5 I/O MODULES

In the example shown in Figure 9.14, a major role is played by the disk controller. The disk controller is an example of an I/O module. The I/O module serves as an interface between

FIGURE 9.14

DMA Initiation and Control



the CPU and the specific device, in this case a disk drive, accepting commands from the CPU on one side and controlling the device on the other. In this example, the I/O module provides the following functions:

- The I/O module recognizes messages addressed to it and accepts commands from the CPU establishing what the disk drive is to do. In this case, the I/O module recognizes that a block of data is to be written from memory to disk using DMA.
- The I/O module provides a buffer where the data from memory can be held until it can be transferred to the disk.
- The I/O module provides the necessary registers and controls to perform a direct memory transfer. This requires that the I/O module have access to a memory address register and a memory data register separate from those of the CPU, either within the I/O module or as a separate DMA controller.
- The I/O module controls the disk drive, moving the head to the physical location on the disk where data is to be written.
- The I/O module copies data from its buffer to the disk.
- The I/O module has interrupt capability, which it uses to notify the CPU when the transfer is complete. It can also interrupt the CPU to notify it of errors or problems that arise during the transfer.

It is desirable to offload tasks specific to I/O operations from the CPU to a separate module or modules which are designed specifically for I/O data transfer and device control. In some cases, the I/O module even provides a processor of its own to offload I/O related processing from the system CPU.

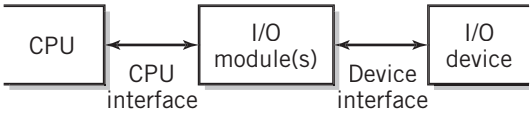
The use of separate I/O modules offers several benefits:

- The module can be designed to provide the specialized control required by a particular device.
- The module frees the CPU to perform other tasks while the much slower I/O operations are taking place.
- The presence of I/O modules allows control of several different I/O devices to occur simultaneously.
- A processor-based module can provide specialized services that would otherwise overload the system CPU with time-consuming CPU-intensive work. For example, a high-end graphics display I/O controller can decode compressed and encrypted MPEG video or adjust images for lighting and shading effects.

As seen in Figure 9.15, I/O modules perform two different functions. At the CPU interface, the module performs CPU interfacing tasks: accepting I/O commands from the CPU, transferring data between the module and the CPU or memory, and sending interrupts and status information to the CPU. At the device interface, the I/O module supplies control of the device—moving the head to the correct track in a disk drive and rewinding tape, for example. Most I/O modules provide buffering of the data to synchronize the different speeds of the CPU and the various I/O devices. Some modules must also have the capability of receiving requests from a device independently from the

FIGURE 9.15

I/O Module Interfaces



computer, and must be able to pass the request in the form of an interrupt to the computer. This is true for any device that can cause an unexpected interrupt, including devices that can be installed or removed during computer operation, (sometimes known as *hot-swappable* or *plug-and-play*), devices with removable media, and network modules.

An I/O module or control unit module used to control a peripheral device is known as a **device controller**, or sometimes as a **device card**. Often, the device is named–the module in the preceding example would be referred to specifically as a *disk controller*, a network module would be called a *network interface controller* or *network interface card*. And, of course, you're probably already familiar with *graphics cards*. A device controller accepts I/O requests and interacts directly with the device to satisfy those requests. The device controllers are individually designed to provide the specialized built-in circuitry necessary for control of a particular type of device. This ability is important, because there is such a variety of requirements for different peripheral devices. A tape drive must be turned on and off and switched between fast forward, play, and rewind. A disk head must be moved to the correct track. A display screen requires a steady transfer of data representing each point on the screen and special circuitry to maintain the correct position on the screen for the display of each point. (Operation of a display controller is discussed in Chapter 10.)

It would be difficult to program the CPU to provide the correct types of signals to operate these and other types of I/O devices, and the CPU time required to control these devices would significantly reduce the usefulness of the system. With a device controller, simple CPU I/O instructions can be used to control quite complex operations. Multiple devices of the same kind can often be controlled with a single controller.

In a small system, most of the I/O modules are device controllers that serve as direct interfaces between a general system bus and each of the system's peripheral devices. There may also be I/O modules that act as an additional interface between the system bus and other modules that then connect to the device. In a typical PC, for example, the disk controller is normally mounted inside the PC case and connects directly to a system bus. The printer, on the other hand, is controlled indirectly. One I/O module connects to the system bus and terminates in a parallel bus or USB port; the actual print controller is inside the printer, at the other end of the bus.

In general I/O modules simplify the task of interfacing peripheral devices to a CPU. I/O modules offload a considerable amount of work from the CPU. They make it possible to control I/O to a peripheral with a few simple I/O commands from the CPU. They support DMA, so that the CPU may be free to perform other tasks. And, as we have already noted, device controllers provide the specialized circuitry required to interface different types of peripherals to the computer.

Much of the power in modern computers comes from the ability to separate out CPU operations from other, more individualistic, I/O peripheral functions, and allowing the processing of each to progress in parallel. In fact, the more powerful the computer, the more essential is the separation of I/O to the satisfactory operation of the system as a whole.

SUMMARY AND REVIEW

This chapter describes the two methods used for input/output, programmed I/O and DMA, and introduces the various components and configurations that make both methods

possible. After a brief description of the I/O requirements of the most common peripherals, the text describes the process of programmed I/O, and describes the advantages and disadvantages of this technique. In general, the use of programmed I/O is limited to slow devices such as keyboards and mouses that are not block oriented.

Next, we introduced the concept of an interrupt, a means of causing the CPU to take special action. We described various ways in which an interrupt could be used, including as notification of an external event that requires attention, as a completion signal for I/O, as a means of allocating CPU time, as an abnormal event indicator, and as a way for software to cause the CPU to take special action. We explained the method used by the interrupt to attract the CPU's attention and the ways in which the interrupt is serviced by the CPU. We considered the situation in which multiple interrupts occur and discussed the prioritization of interrupts.

As an alternative to programmed I/O, direct memory access allows the transfer of blocks of data directly between an I/O device and memory. We discussed the hardware requirements that make DMA possible and showed how DMA is used. We explained how DMA works in conjunction with interrupts.

We concluded with a discussion of the I/O modules that serve to control the I/O devices and act as an interface between the peripheral devices, the CPU, and memory. The I/O modules receive messages from the CPU, control the device, initiate and control DMA when required, and produce interrupts. The I/O modules in a channel architecture also serve to direct I/O requests to the proper channel and provide independent, intelligent control of the I/O operation.

FOR FURTHER READING

Detailed discussions of I/O concepts and techniques, including the concepts of interrupts and DMA, can be found in the engineering textbooks previously mentioned, particularly those by Stallings [STAL05] and Tanenbaum [TAN05]. An outstanding treatment of I/O in the IBM mainframe architecture can be found in Prasad [PRAS94] and in Cormier and others [CORM83]. PC I/O is discussed in a number of excellent books, among them Messmer [MESS01] and Sargent [SARG95]. Somewhat less organized, but still valuable, is the treatment found in Henle [HENL92].

KEY CONCEPTS AND TERMS

abnormal event
buffer
channel program
context
device card
device controller
device driver
direct memory access (DMA)
exception
external event
interrupt
interrupt handler
interrupt lines
interrupt routine
interrupt service
maskable
polling
priority
privileged instruction
process control block (PCB)
programmed I/O
quantum
software interrupt
trap
vectored interrupt

READING REVIEW QUESTIONS

9.1 In terms of the nature of the data, how does a keyboard differ from a hard disk as an input device?

9.2 Name at least two devices that can generate unexpected input.

9.3 Explain the purpose of a buffer.

9.4 Explain the reasons why programmed I/O does not work very well when the I/O device is a hard disk or a graphics display.

9.5 When an interrupt occurs, what happens to the program that is currently executing at the time?

9.6 What is a *context*? What does it contain? What is it used for?

9.7 The book lists four primary uses for interrupts. State and explain at least three of them.

9.8 What kind of interrupt occurs when a user's program tries to execute a privileged instruction?

9.9 What does DMA stand for? What capability does DMA add to a computer?

9.10 What are the three primary conditions that are required for DMA to take place?

9.11 What data must an I/O controller have before a DMA transfer takes place? How is this data sent to the controller?

9.12 What is the purpose of a completion interrupt at the conclusion of a DMA transfer?

9.13 A graphics card is an example of an I/O controller. I/O controllers have (at least) *two* interfaces. What are the two interfaces of a graphics card connected to?

9.14 Name at least three benefits that are provided by I/O modules.

EXERCISES

9.1 Why would DMA be useless if the computer did not have interrupt capability?

9.2 What is the advantage of using a disk controller to control the hard disk? How else could you do the job that the disk controller does?

9.3 DMA is rarely used with dumb computer terminals. Why?

9.4 Consider the interrupt that occurs at the completion of a disk transfer.

- **a.** "Who" is interrupting "whom"?
- **b.** Why is the interrupt used in this case? What would be necessary if there were no interrupt capability on this computer?
- **c.** Describe the steps that take place after the interrupt occurs.

9.5 Suppose you wish to send a block of data to a tape drive for storage using DMA. What information must be sent to the tape controller before the DMA transfer can take place?

9.6 What is an interrupt vector?

9.7 What is polling used for? What are the disadvantages of polling? What is a better way to perform the same job?

9.8 To use a computer for multimedia (moving video and sound), it is important to maximize the efficiency of the I/O. Assume that the blocks of a movie are stored

consecutively on a CD-ROM. Describe the steps used to retrieve the blocks for use by the movie display software. Discuss ways in which you could optimize the performance of the I/O transfer.

9.9 Consider the interface between a computer and a printer. For a typical printout it is clearly impractical to send output data to the printer one byte or one word at a time (especially over a network!). Instead data to be printed is stored in a buffer at a known location in memory and transferred in blocks to memory in the printer. A controller in the printer then handles the actual printing from the printer's memory.

The printer's memory is not always sufficient to hold the entire printout data at one time. Printer problems, such as an ''out of paper'' condition, can also cause delays. Devise and describe, in as much detail as you can, an interrupt/DMA scheme that will assure that all documents will be successfully printed.

9.10 The UNIX operating system differentiates between block-oriented and character-oriented devices. Give an example of each, explain the differences between them, and explain how the I/O process differs for each.

9.11 Describe a circumstance where an interrupt occurs at the beginning of an event. Describe a circumstance where an interrupt occurs at the completion of an event. What is the difference between the types of events?

9.12 In general, what purpose does an interrupt serve? Stated another way, suppose there were no interrupts provided in a computer. What capabilities would be lost?

9.13 What is the difference between polling and polled interrupt processing?

9.14 Describe the steps that occur when a system receives multiple interrupts.