
CHAPTER

4

INPUT/OUTPUT ORGANIZATION

CHAPTER OBJECTIVES

In this chapter you will learn about:

- How program-controlled I/O is performed using polling
- The idea of interrupts and the hardware and software needed to support them
- Direct memory access as an I/O mechanism for high-speed devices
- Data transfer over synchronous and asynchronous buses
- The design of I/O interface circuits
- Commercial bus standards, in particular the PCI, SCSI, and USB buses

One of the basic features of a computer is its ability to exchange data with other devices. This communication capability enables a human operator, for example, to use a keyboard and a display screen to process text and graphics. We make extensive use of computers to communicate with other computers over the Internet and access information around the globe. In other applications, computers are less visible but equally important. They are an integral part of home appliances, manufacturing equipment, transportation systems, banking and point-of-sale terminals. In such applications, input to a computer may come from a sensor switch, a digital camera, a microphone, or a fire alarm. Output may be a sound signal to be sent to a speaker or a digitally coded command to change the speed of a motor, open a valve, or cause a robot to move in a specified manner. In short, a general-purpose computer should have the ability to exchange information with a wide range of devices in varying environments.

In this chapter, we will consider in detail various ways in which I/O operations are performed. First, we will consider the problem from the point of view of the programmer. Then, we will discuss some of the hardware details associated with buses and I/O interfaces and introduce some commonly used bus standards.

4.1 ACCESSING I/O DEVICES

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement, as shown in Figure 4.1. The bus enables all the devices connected to it to exchange information. Typically, it consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the requested data are transferred over the data lines. As mentioned in Section 2.7, when I/O devices and the memory share the same address space, the arrangement is called *memory-mapped I/O*.

With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address

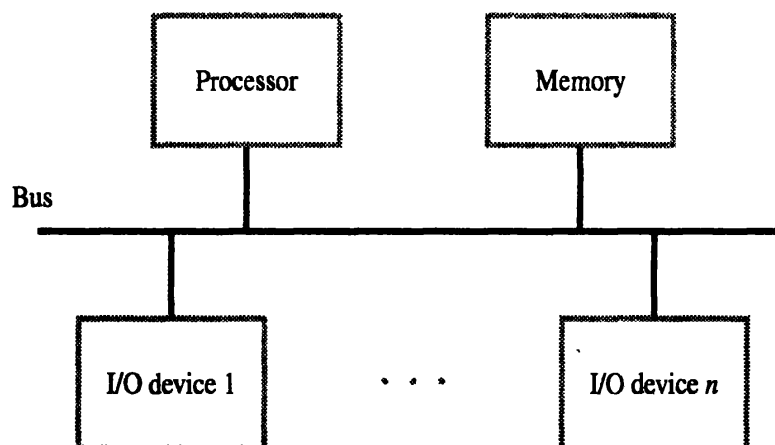


Figure 4.1 A single-bus structure.

of the input buffer associated with the keyboard, the instruction

Move DATAIN,R0

reads the data from DATAIN and stores them into processor register R0. Similarly, the instruction

Move R0,DATAOUT

sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer.

Most computer systems use memory-mapped I/O. Some processors have special In and Out instructions to perform I/O transfers. For example, processors in the Intel family described in Chapter 3 have special I/O instructions and a separate 16-bit address space for I/O devices. When building a computer system based on these processors, the designer has the option of connecting I/O devices to use the special I/O address space or simply incorporating them as part of the memory address space. The latter approach is by far the most common as it leads to simpler software. One advantage of a separate I/O address space is that I/O devices deal with fewer address lines. Note that a separate I/O address space does not necessarily mean that the I/O address lines are physically separate from the memory address lines. A special signal on the bus indicates that the requested read or write transfer is an I/O operation. When this signal is asserted, the memory unit ignores the requested transfer. The I/O devices examine the low-order bits of the address bus to determine whether they should respond.

Figure 4.2 illustrates the hardware required to connect an I/O device to the bus. The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and

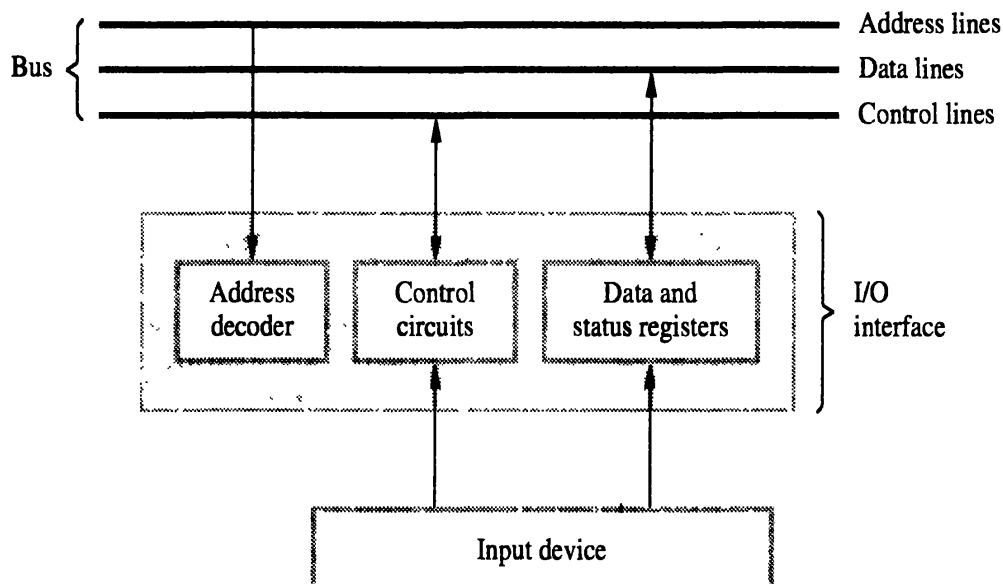


Figure 4.2 I/O interface for an input device.

assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's *interface circuit*.

I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. Also, we must make sure that an input character is read only once.

The basic ideas used for performing input and output operations were introduced in Section 2.7. For an input device such as a keyboard, a status flag, SIN, is included in the interface circuit as part of the status register. This flag is set to 1 when a character is entered at the keyboard and cleared to 0 once this character is read by the processor. Hence, by checking the SIN flag, the software can ensure that it is always reading valid data. This is often accomplished in a program loop that repeatedly reads the status register and checks the state of SIN. When SIN becomes equal to 1, the program reads the input data register. A similar procedure can be used to control output operations using an output status flag, SOUT.

Example 4.1

To review the basic concepts, let us consider a simple example of I/O operations involving a keyboard and a display device in a computer system. The four registers shown in Figure 4.3 are used in the data transfer operations. Register STATUS contains two control flags, SIN and SOUT, which provide status information for the keyboard and the display unit, respectively. The two flags KIRQ and DIRQ in this register are used in conjunction with interrupts. They, and the KEN and DEN bits in register CONTROL, will be discussed in Section 4.2. Data from the keyboard are made available

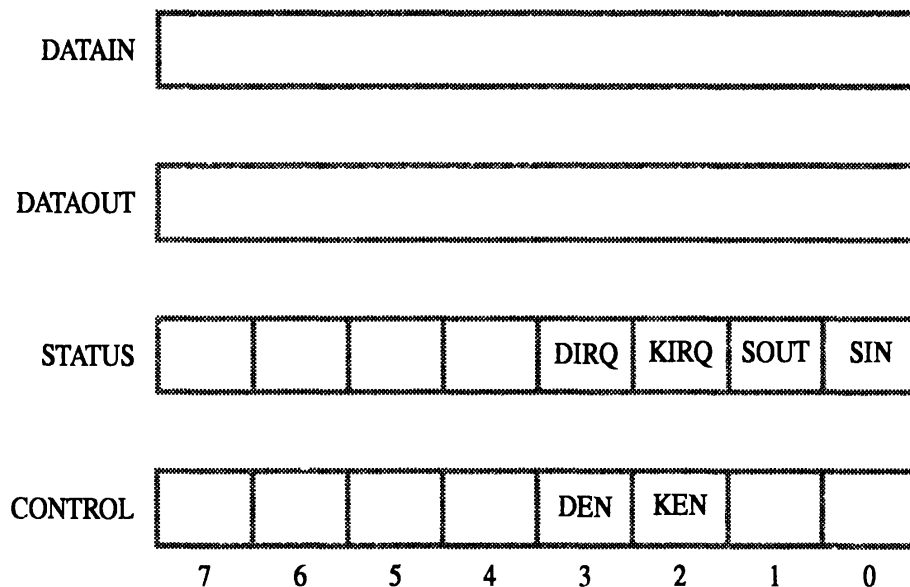


Figure 4.3 Registers in keyboard and display interfaces.

	Move	#LINE,R0	Initialize memory pointer.
WAITK	TestBit	#0,STATUS	Test SIN.
	Branch=0	WAITK	Wait for character to be entered.
	Move	DATAIN,R1	Read character.
WAITD	TestBit	#1,STATUS	Test SOUT.
	Branch=0	WAITD	Wait for display to become ready.
	Move	R1,DATAOUT	Send character to display.
	Move	R1,(R0)+	Store character and advance pointer.
	Compare	#\$0D,R1	Check if Carriage Return.
	Branch≠0	WAITK	If not, get another character.
	Move	#\$0A,DATAOUT	Otherwise, send Line Feed.
	Call	PROCESS	Call a subroutine to process the the input line.

Figure 4.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

in the DATAIN register, and data sent to the display are stored in the DATAOUT register.

The program in Figure 4.4 is similar to that in Figure 2.20. This program reads a line of characters from the keyboard and stores it in a memory buffer starting at location LINE. Then, it calls a subroutine PROCESS to process the input line. As each character is read, it is *echoed back* to the display. Register R0 is used as a pointer to the memory buffer area. The contents of R0 are updated using the Autoincrement addressing mode so that successive characters are stored in successive memory locations.

Each character is checked to see if it is the Carriage Return (CR) character, which has the ASCII code 0D (hex). If it is, a Line Feed character (ASCII code 0A) is sent to move the cursor one line down on the display and subroutine PROCESS is called. Otherwise, the program loops back to wait for another character from the keyboard.

This example illustrates *program-controlled I/O*, in which the processor repeatedly checks a status flag to achieve the required synchronization between the processor and an input or output device. We say that the processor *polls* the device. There are two other commonly used mechanisms for implementing I/O operations: interrupts and direct memory access. In the case of interrupts, synchronization is achieved by having the I/O device send a special signal over the bus whenever it is ready for a data transfer operation. Direct memory access is a technique used for high-speed I/O devices. It involves having the device interface transfer data directly to or from the memory, without continuous involvement by the processor. We will discuss these mechanisms in the next three sections. Then, we will examine the hardware involved, which includes the processor bus and the I/O device interface.

4.2 INTERRUPTS

In the example of Figure 4.4, the program enters a wait loop in which it repeatedly tests the device status. During this period, the processor is not performing any useful computation. There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the processor when it becomes ready. It can do so by sending a hardware signal called an *interrupt* to the processor. At least one of the bus control lines, called an *interrupt-request* line, is usually dedicated for this purpose. Since the processor is no longer required to continuously check the status of external devices, it can use the waiting period to perform other useful functions. Indeed, by using interrupts, such waiting periods can ideally be eliminated.

Example 4.2

Consider a task that requires some computations to be performed and the results to be printed on a line printer. This is followed by more computations and output, and so on. Let the program consist of two routines, COMPUTE and PRINT. Assume that COMPUTE produces a set of n lines of output, to be printed by the PRINT routine.

The required task may be performed by repeatedly executing first the COMPUTE routine and then the PRINT routine. The printer accepts only one line of text at a time. Hence, the PRINT routine must send one line of text, wait for it to be printed, then send the next line, and so on, until all the results have been printed. The disadvantage of this simple approach is that the processor spends a considerable amount of time waiting for the printer to become ready. If it is possible to overlap printing and computation, that is, to execute the COMPUTE routine while printing is in progress, a faster overall speed of execution will result. This may be achieved as follows. First, the COMPUTE routine is executed to produce the first n lines of output. Then, the PRINT routine is executed to send the first line of text to the printer. At this point, instead of waiting for the line to be printed, the PRINT routine may be temporarily suspended and execution of the COMPUTE routine continued. Whenever the printer becomes ready, it alerts the processor by sending an interrupt-request signal. In response, the processor interrupts execution of the COMPUTE routine and transfers control to the PRINT routine. The PRINT routine sends the second line to the printer and is again suspended. Then the interrupted COMPUTE routine resumes execution at the point of interruption. This process continues until all n lines have been printed and the PRINT routine ends.

The PRINT routine will be restarted whenever the next set of n lines is available for printing. If COMPUTE takes longer to generate n lines than the time required to print them, the processor will be performing useful computations all the time.

This example illustrates the concept of interrupts. The routine executed in response to an interrupt request is called the *interrupt-service routine*, which is the PRINT routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction i in Figure 4.5. The