chapter

3

BASIC INPUT/OUTPUT

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Transferring data between a processor and input/output (I/O) devices
- The programmer's view of I/O transfers
- How program-controlled I/O is performed using polling
- How interrupts are used in I/O transfers

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 sent to a speaker, or a digitally coded command that changes the speed of a motor, opens a valve, or causes a robot to move in a specified manner. In short, computers should have the ability to exchange digital and analog information with a wide range of devices in many different environments.

In this chapter we will consider the input/output (I/O) capability of computers as seen from the programmer's point of view. We will present only basic I/O operations, which are provided in all computers. This knowledge will enable the reader to perform interesting and useful exercises on equipment found in a typical teaching laboratory environment. More complex I/O schemes, as well as the hardware needed to implement the I/O capability, are discussed in Chapter 7.

3.1 ACCESSING I/O DEVICES

The components of a computer system communicate with each other through an interconnection network, as shown in Figure 3.1. The interconnection network consists of circuits needed to transfer information between the processor, the memory unit, and a number of I/O devices.

In Chapter 2, we described the concept of an address space and how the processor may access individual memory locations within such an address space. Load and Store instructions use addressing modes to generate effective addresses that identify the desired locations. This idea of using addresses to access various locations in the memory can be

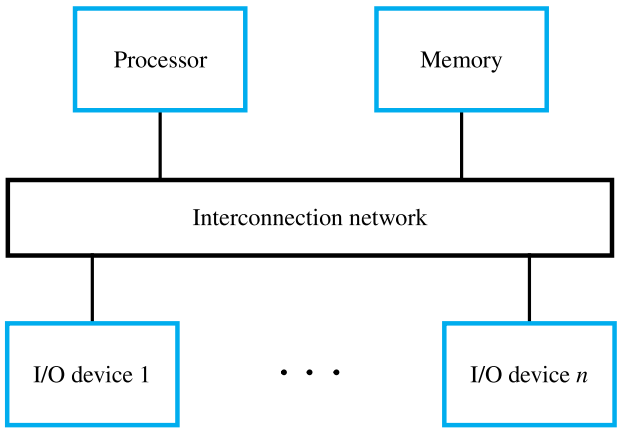

Figure 3.1 A computer system.

extended to deal with the I/O devices as well. For this purpose, each I/O device must appear to the processor as consisting of some addressable locations, just like the memory. Some addresses in the address space of the processor are assigned to these I/O locations, rather than to the main memory. These locations are usually implemented as bit storage circuits (flip-flops) organized in the form of registers. It is customary to refer to them as *I/O registers*. Since the I/O devices and the memory share the same address space, this arrangement is called *memory-mapped I/O*. It is used in most computers.

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 of a register in an input device, the instruction

```
Load     R2, DATAIN
```

reads the data from the DATAIN register and loads them into processor register R2. Similarly, the instruction

```
Store    R2, DATAOUT
```

sends the contents of register R2 to location DATAOUT, which is a register in an output device.

3.1.1 I/O DEVICE INTERFACE

An I/O device is connected to the interconnection network by using a circuit, called the *device interface*, which provides the means for data transfer and for the exchange of status and control information needed to facilitate the data transfers and govern the operation of the device. The interface includes some registers that can be accessed by the processor. One register may serve as a buffer for data transfers, another may hold information about the current status of the device, and yet another may store the information that controls the operational behavior of the device. These *data*, *status*, and *control* registers are accessed by program instructions as if they were memory locations. Typical transfers of information are between I/O registers and the registers in the processor. Figure 3.2 illustrates how the keyboard and display devices are connected to the processor from the software point of view.

3.1.2 PROGRAM-CONTROLLED I/O

Let us begin the discussion of input/output issues by looking at two essential I/O devices for human-computer interaction—keyboard and display. Consider a task that reads characters typed on a keyboard, stores these data in the memory, and displays the same characters on a display screen. A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action. This method is known as *program-controlled I/O*.

In addition to transferring each character from the keyboard into the memory, and then to the display, it is necessary to ensure that this happens at the right time. An input character must be read in response to a key being pressed. For output, a character must be sent to

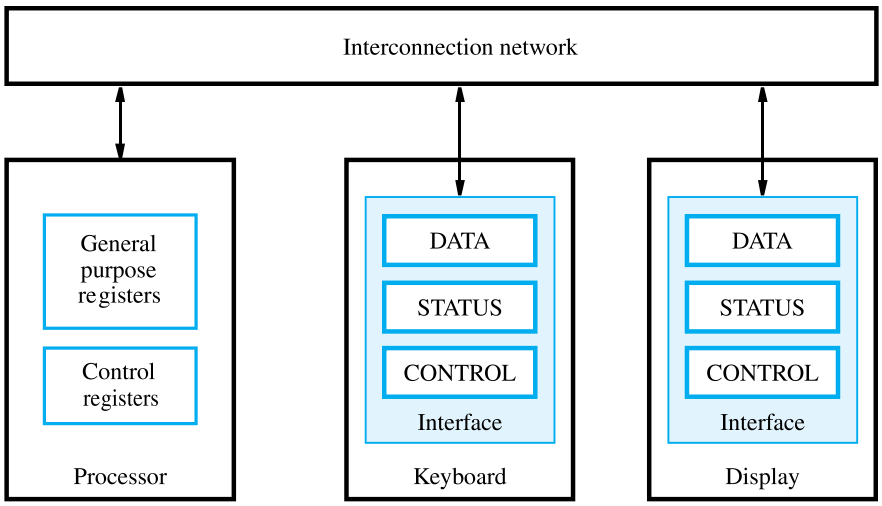

Figure 3.2 The connection for processor, keyboard, and display.

the display only when the display device is able to accept it. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted to and displayed on the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can execute billions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

One solution to this problem involves a signaling protocol. On output, the processor sends the first character and then waits for a signal from the display that the next character can be sent. It then sends the second character, and so on. An input character is obtained from the keyboard in a similar way. The processor waits for a signal indicating that a key has been pressed and that a binary code that represents the corresponding character is available in an I/O register associated with the keyboard. Then the processor proceeds to read that code.

The keyboard includes a circuit that responds to a key being pressed by producing the code for the corresponding character that can be used by the computer. We will assume that ASCII code (presented in Table 1.1) is used, in which each character code occupies one byte. Let KBD_DATA be the address label of an 8-bit register that holds the generated character. Also, let a signal indicating that a key has been pressed be provided by setting to 1 a flip-flop called KIN, which is a part of an eight-bit status register, KBD_STATUS. The processor can read the *status flag* KIN to determine when a character code has been placed in KBD_DATA. When the processor reads the status flag to determine its state, we say that the processor *polls* the I/O device.

The display includes an 8-bit register, which we will call DISP_DATA, used to receive characters from the processor. It also must be able to indicate that it is ready to receive the

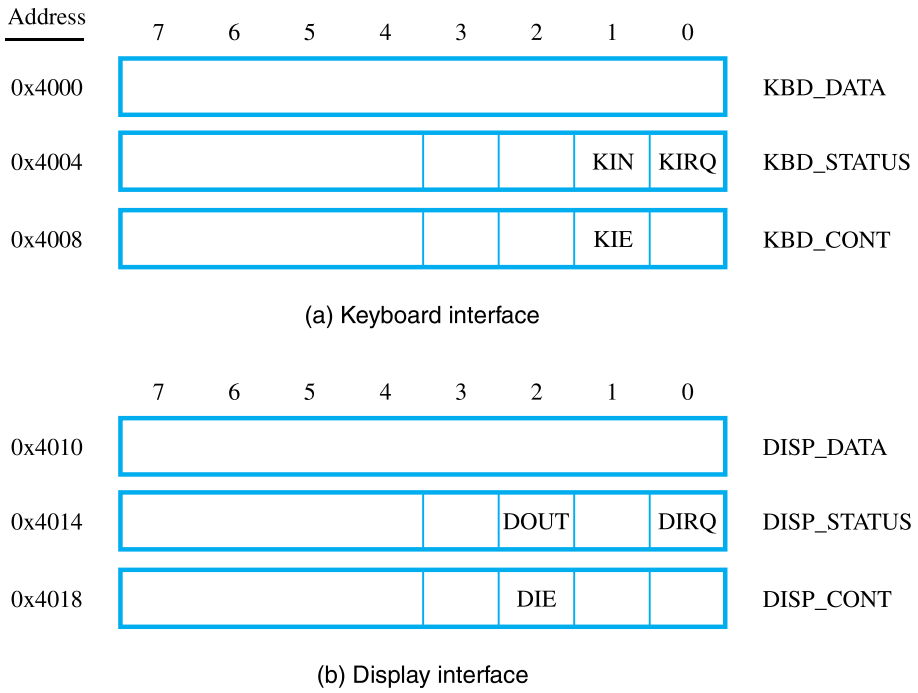

Figure 3.3 Registers in the keyboard and display interfaces.

next character; this can be done by using a status flag called DOUT, which is one bit in a status register, DISP_STATUS.

Figure 3.3 illustrates how these registers may be organized. The interface for each device also includes a control register, which we will discuss in Section 3.2. We have identified only a few bits in the registers, those that are pertinent to the discussion in this chapter. Other bits can be used for other purposes, or perhaps simply ignored.

If the registers in I/O interfaces are to be accessed as if they are memory locations, each register must be assigned a specific address that will be recognized by the interface circuit. In Figure 3.3, we assigned hexadecimal numbers 4000 and 4010 as base addresses for the keyboard and display, respectively. These are the addresses of the data registers. The addresses of the status registers are four bytes higher, and the control registers are eight bytes higher. This makes all addresses word-aligned in a 32-bit word computer, which is usually done in practice. Assigning the addresses to registers in this manner makes the I/O registers accessible in a program executed by the processor. This is the programmer's view of the device.

A program is needed to perform the task of reading the characters produced by the keyboard, storing these characters in the memory, and sending them to the display. To perform I/O transfers, the processor must execute machine instructions that check the state of the status flags and transfer data between the processor and the I/O devices.

Let us consider the details of the input process. When a key is pressed, the keyboard circuit places the ASCII-encoded character into the KBD_DATA register. At the same time, the circuit sets the KIN flag to 1. Meanwhile, the processor is executing the I/O program which continuously checks the state of the KIN flag. When it detects that KIN is set to 1, it transfers the contents of KBD_DATA into a processor register. Once the contents of KBD_DATA are read, KIN must be cleared to 0, which is usually done automatically by the interface circuit. If a second character is entered at the keyboard, KIN is again set to 1 and the process repeats. The desired action can be achieved by performing the operations:

```
READWAIT    Read the KIN flag
            Branch to READWAIT if KIN = 0
            Transfer data from KBD_DATA to R5
```

which reads the character into processor register R5.

An analogous process takes place when characters are transferred from the processor to the display. When DOUT is equal to 1, the display is ready to receive a character. Under program control, the processor monitors DOUT, and when DOUT is equal to 1, the processor transfers an ASCII-encoded character to DISP_DATA. The transfer of a character to DISP_DATA clears DOUT to 0. When the display device is ready to receive a second character, DOUT is again set to 1. This can be achieved by performing the operations:

```
WRITEWAIT   Read the DOUT flag
            Branch to WRITEWAIT if DOUT = 0
            Transfer data from R5 to DISP_DATA
```

The wait loop is executed repeatedly until the status flag DOUT is set to 1 by the display when it is free to receive a character. Then, the character from R5 is transferred to DISP_DATA to be displayed, which also clears DOUT to 0.

We assume that the initial state of KIN is 0 and the initial state of DOUT is 1. This initialization is normally performed by the device control circuits when power is turned on.

In computers that use memory-mapped I/O, in which some addresses are used to refer to registers in I/O interfaces, data can be transferred between these registers and the processor using instructions such as Load, Store, and Move. For example, the contents of the keyboard character buffer KBD_DATA can be transferred to register R5 in the processor by the instruction

```
LoadByte    R5, KBD_DATA
```

Similarly, the contents of register R5 can be transferred to DISP_DATA by the instruction

```
StoreByte   R5, DISP_DATA
```

The LoadByte and StoreByte operation codes signify that the operand size is a byte, to distinguish them from the Load and Store operation codes that we have used for word operands.

The Read operation described above may be implemented by the RISC-style instructions:

```
READWAIT:   LoadByte          R4, KBD_STATUS
            And               R4, R4, #2
            Branch_if_[R4]=0  READWAIT
            LoadByte          R5, KBD_DATA
```

The And instruction is used to test the KIN flag, which is bit b_1 of the status information in R4 that was read from the KBD_STATUS register. As long as $b_1 = 0$, the result of the AND operation leaves the value in R4 equal to zero, and the READWAIT loop continues to be executed.

Similarly, the Write operation may be implemented as:

```
WRITEWAIT:  LoadByte          R4, DISP_STATUS
            And               R4, R4, #4
            Branch_if_[R4]=0  WRITEWAIT
            StoreByte         R5, DISP_DATA
```

Observe that the And instruction in this case uses the immediate value 4 to test the display's status bit, b_2.

3.1.3 AN EXAMPLE OF A RISC-STYLE I/O PROGRAM

We can now put together a complete program for a typical I/O task, as shown in Figure 3.4. The program uses the program-controlled I/O approach described above to read, store, and display a line of characters typed at the keyboard. As the characters are read in, one by one, they are stored in the memory and then *echoed* back to the display. The program finishes when the carriage return character, CR, is encountered. The address of the first byte location of the memory where the line is to be stored is LOC. Register R2 is used to point to this part of the memory, and it is initially loaded with the address LOC by the first instruction in the program. R2 is incremented for each character read and displayed.

3.1.4 AN EXAMPLE OF A CISC-STYLE I/O PROGRAM

Let us now perform the same task using CISC-style instructions. In CISC instruction sets it is possible to perform some arithmetic and logic operations directly on operands in the memory. So, it is possible to have the instruction

```
TestBit     destination, #k
```

which tests bit b_k of the destination operand and sets the condition flag Z (Zero) to 1 if $b_k = 0$ and to 0 otherwise. Since the operand can be in a memory location, we can use the instruction

```
TestBit     KBD_STATUS, #1
```

```
        Move                R2, #LOC        Initialize pointer register R2 to point to the
                                              address of the first location in main memory
                                              where the characters are to be stored.
        MoveByte            R3, #CR         Load ASCII code for Carriage Return into R3.
READ:   LoadByte            R4, KBD_STATUS  Wait for a character to be entered.
        And                 R4, R4, #2      Check the KIN flag.
        Branch_if_[R4]=0    READ
        LoadByte            R5, KBD_DATA    Read the character from KBD_DATA
                                              (this clears KIN to 0).
        StoreByte           R5, (R2)        Write the character into the main memory and
        Add                 R2, R2, #1        increment the pointer to main memory.
ECHO:   LoadByte            R4, DISP_STATUS Wait for the display to become ready.
        And                 R4, R4, #4      Check the DOUT flag.
        Branch_if_[R4]=0    ECHO
        StoreByte           R5, DISP_DATA   Move the character just read to the display
                                              buffer register (this clears DOUT to 0).
        Branch_if_[R5]≠[R3] READ            Check if the character just read is the
                                              Carriage Return. If it is not, then
                                              branch back and read another character.
```

Figure 3.4 A RISC-style program that reads a line of characters and displays it.

to test the state of the KIN flag in the keyboard interface. A Branch instruction that checks the state of the Z flag can then be used to cause a branch to the beginning of the wait loop.

Figure 3.5 gives a CISC-style program that reads and displays a line of characters. Observe that the first MoveByte instruction transfers each character directly from KBD_DATA to the memory location pointed to by R2. A Compare instruction

```
Compare    destination, source
```

performs the comparison by subtracting the contents of the source from the contents of the destination, and then sets the condition flags based on the result. It does not change the contents of either the source or the destination. Note that the CompareByte instruction in Figure 3.5 uses the autoincrement addressing mode, which automatically increments the value of the pointer R2 after the comparison has been made. In the RISC-style program in Figure 3.4 the pointer has to be incremented using a separate Add instruction.

We have discussed the memory-mapped I/O scheme, which is used in most computers. There is an alternative that can be found in some processors where there exist special In and Out instructions to perform I/O transfers. In this case, there exists a separate I/O address space used only by these instructions. When building a computer system that uses 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.