CHAPTER

# 9

# EMBEDDED SYSTEMS

## CHAPTER OBJECTIVES

In this chapter you will learn about:

- Embedded applications
- Microcontrollers for embedded systems
- Dealing with I/O device constraints
- Using the C language to control I/O devices
- System-on-a-chip design

Computer systems are used in a myriad of applications; therefore, they come in a variety of organizations, sizes, and capabilities. The important factors that must be considered for any given application include performance, reliability, and cost. Any computer that handles large amounts of graphical information, involving pictorial images and animation, must have good real-time performance. In the case of personal computers and workstations, it is important to achieve the best performance for a cost that meets the demands of the marketplace. Exceptionally high performance inevitably implies a much costlier machine. Such machines are needed in environments where massive computations have to be performed in a reasonable time. Some examples of computationally intensive applications are: simulation of a complex system, determination of a good wiring pattern on a printed circuit board, and many computer-aided design (CAD) tasks. These applications may take many hours to run on a personal computer. It is usually necessary to run them on a *compute server*, which has much higher performance achieved at a correspondingly greater cost. Chapter 8 dealt with many issues involved in building high-performance computers.

There are many applications that do not need a high performance processor. Microprocessor control is now commonly used in cameras, cell phones, display phones, point-of-sale terminals, kitchen appliances, cars, and many toys. High performance is not crucial in these applications. Low cost and high reliability are the essential requirements. Small size and low power consumption are often of key importance. All of this can be achieved by placing on a single chip not only the processor circuitry, but also some input/output interfaces, timer circuits, and other design features to make it easy to implement a complete computer control system using very few chips. Microprocessor chips that include I/O interfaces and some memory are generally referred to as *microcontrollers*. A physical system that employs computer control for a specific purpose, rather than for general-purpose computation, is referred to as an *embedded system*. Such systems are the subject of this chapter.

## 9.1 EXAMPLES OF EMBEDDED SYSTEMS

In this section we present three examples of embedded systems to illustrate the processing and control capability needed in a typical embedded application.

### 9.1.1 MICROWAVE OVEN

Many household appliances use computer control to govern their operation. A typical example is a microwave oven. This appliance is based on a magnetron power unit that generates microwaves used to heat food in a confined space. When turned on, the magnetron generates its maximum power output. Lower power levels are achieved by turning the magnetron on and off for controlled time intervals. Thus, by controlling the power level and the total heating time, it is possible to realize a variety of user-selectable cooking options.

The specification for a microwave oven may include the following cooking options:

- Manual selection of the power level and cooking time
- A manually selected sequence of different cooking steps
- Automatic selection where the user specifies the type of food (for example, meat, vegetables, or popcorn) and the weight of the food. An appropriate power level and time are then calculated by the controller
- Automatic defrosting of meat by specifying the weight

The oven includes an output display that can show:

- Time of day clock
- Decrementing clock timer while cooking
- Information messages to the user

An audio alert signal, in the form of a beep tone, is used to indicate the end of a cooking operation. An exhaust fan and oven light are provided. Finally, a door interlock must turn the magnetron off if the door of the oven is open. All of these functions can be controlled by a microprocessor.

The input/output capability needed to communicate with the user has to include:

- Input keys that comprise the number pad 0 to 9 and function keys such as Reset, Start, Stop, Power Level, Auto Defrost, Auto Cooking, Clock Set, and Fan Control. To reduce the total number of keys, some keys may have multiple functions, for example, pressing the Fan Control key a number of times may be used to select the speed of the fan.
- Visual output in the form of a liquid crystal display (similar to the seven-segment display discussed in Section A.9).
- A small speaker that produces the beep tone.

The controller for a microwave oven can be implemented by a small microprocessor-based computer unit. The computational tasks are quite simple. They include maintaining the time of day clock, determining the actions needed in the various cooking options, and generating the control signals needed to turn on or off devices such as the magnetron and the fan, and generating display information. Therefore, it is possible to use a relatively simple processor to perform these tasks. The program needed to implement the desired actions is quite small. It must be stored in a nonvolatile read-only memory, so that it will not be lost when the power is turned off. It is also necessary to have a small RAM for use during computations and to hold the user-entered data. The most substantial requirement is to have a lot of I/O capability to deal with all of the input keys, displays, and output control signals.

From the designer's point of view, it is important to find a cost-effective solution to realize the desired controller. Parallel I/O ports provide a convenient mechanism for dealing with the external input and output signals. Figure 9.1 shows a possible organization of the microwave oven. A key point is that the amount of hardware used is quite small. A simple processor with small ROM and RAM units is sufficient, and

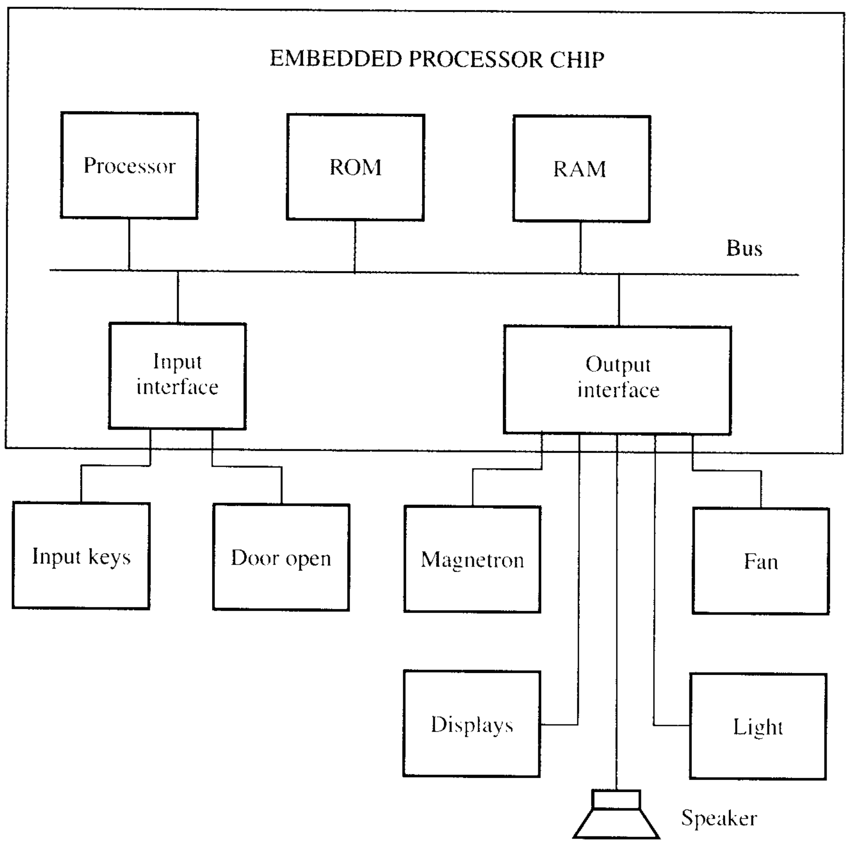

**Figure 9.1** A block diagram of a microwave oven.

simple input and output interfaces are used to connect to the rest of the system. It is possible to realize most of this circuitry on a single VLSI chip.

## 9.1.2 DIGITAL CAMERA

Digital cameras provide an excellent example of a sophisticated embedded system in a small package. Figure 9.2 shows the main parts of a digital camera.

Traditional cameras use film to capture images. In a digital camera, an array of optical sensors is used to capture images. These sensors are based on photodiodes which convert light into electrical charge. The intensity of light determines the amount of charge that is generated. Two different types of sensors are used in commercial products. One type is known as *charge-coupled devices* (CCDs). It is the first type of sensing device used in digital cameras, and it has been refined to give high-quality images. More recently, sensors based on CMOS technology have been developed. They are less expensive, but they do not give images of quite as high quality as CCDs.

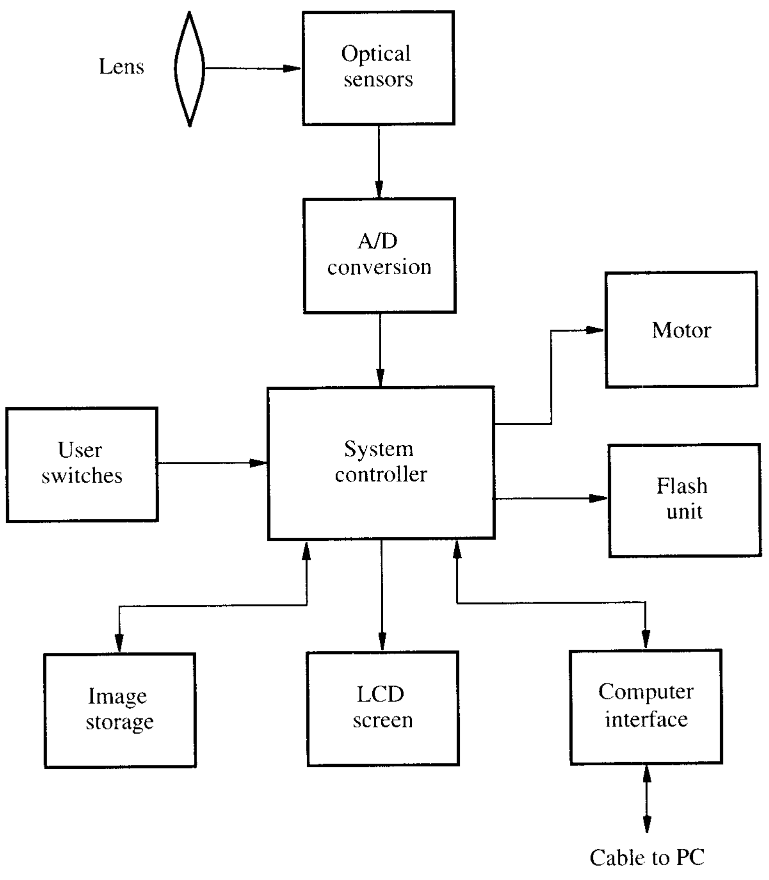

**Figure 9.2** A simplified block diagram of a digital camera.

Each sensing element generates a charge that corresponds to one *pixel*, which is one point of a pictorial image. The number of pixels determines the quality of pictures that can be recorded and displayed. The charge is an analog quantity, which is converted into a digital representation using *analog-to-digital* (A/D) conversion circuits. The A/D conversion produces a digital representation of the image in which the color and intensity of each pixel is represented by a number of bits. The digital form of the image can then be manipulated using standard computer circuitry.

A key functional part is the system controller. This block contains a processor, memory (both RAM and EEPROM), and a variety of interface circuits needed to connect to other parts of the system. The processor governs the operation of the camera. It processes the raw image data obtained from the A/D circuits to generate images represented in standard formats suitable for use in computers, printers, and display devices. The main formats used are TIFF for uncompressed images and JPEG for compressed images.

The processed images are stored in a larger image storage device. Flash memory cards, discussed in Section 5.3.5, are a popular choice for storing images. Other choices include floppy disks and miniature hard disk drives.

A captured and processed image can be displayed on a liquid crystal display (LCD) screen, which is included in the camera. This allows the user to decide whether the image is worth keeping. The number of images that can be taken and saved depends on the size of the image storage unit. It also depends on the chosen quality of the images, namely on the number of pixels per image.

A standard interface provides a simple mechanism for transferring the images to a computer or a printer. This may be a simple serial or parallel interface, or a connector for a standard bus such as PCI or USB. If flash memory cards are used, images can also be transferred by physically transferring the card.

The system controller also generates the signals needed to control the operation of the motor (for focusing purposes) and the flash unit. Some of the inputs come from switches activated by the user.

A digital camera requires a considerably more powerful processor than is needed for the previously discussed microwave-oven application. The processor has to perform quite complex signal processing functions. Yet, it is essential that the processor not consume much power because the camera is a battery-powered device. Typically, the processor consumes less power than the display and flash units of a camera.

## 9.1.3 HOME TELEMETRY

The use of computers in the home is increasing rapidly. They are used as general-purpose computing machines and also in a host of embedded applications. In Subsection 9.1.1, we considered the microwave oven example. Similar examples can be found in other equipment, such as washers, dryers, dishwashers, cooking ranges, furnaces, and air conditioners. Another notable example is the display telephone, in which an embedded processor enables a variety of useful features. In addition to the standard telephone features, a microprocessor controlled phone can be used to provide remote access to other devices in the home that can communicate as computer equipment.

Using the telephone one can remotely perform functions such as:

- Communicate with a computer-controlled home security system
- Set a desirable temperature for a furnace or an air conditioner to maintain
- Set the start time, the cooking time, and temperature for food that has been placed in the oven at some earlier time
- Read the electricity, gas, and water meters, replacing the need for the utility companies that provide these services to send an employee to the home to read the meters

All of this is easily implementable if the device in question is controlled by a microprocessor. It is only necessary to provide a link between the device microprocessor and the microprocessor in the telephone. Such links can be realized in different ways. The simplest is to use bit-serial communication, which is easily accomplished if the controller chips include UART interfaces of the type discussed in Section 4.6.2. Using signaling from a remote location to observe and control the state of equipment is often referred to as *telemetry*.

## 9.2 PROCESSOR CHIPS FOR EMBEDDED APPLICATIONS

A chip that contains a processor, some memory, and I/O interface circuitry useful in embedded applications is often called an *embedded processor*. Since such chips perform important control functions in the applications, and are based on a microprocessor, they are also known as *microcontroller* chips.

An embedded processor chip should be versatile enough to serve a wide variety of applications. Figure 9.3 shows the block diagram of a typical chip. The main part is a *processor core*, which may be the basic version of a commercially available microprocessor. It is prudent to choose a microprocessor architecture that has proven to be popular in practice, because for such processors there exist numerous CAD tools, good books, and a large amount of experience and knowledge that facilitate the design of new products.

It is useful to include some memory on the chip, which may be sufficient to satisfy the memory requirements found in small applications. Some of this memory has to be

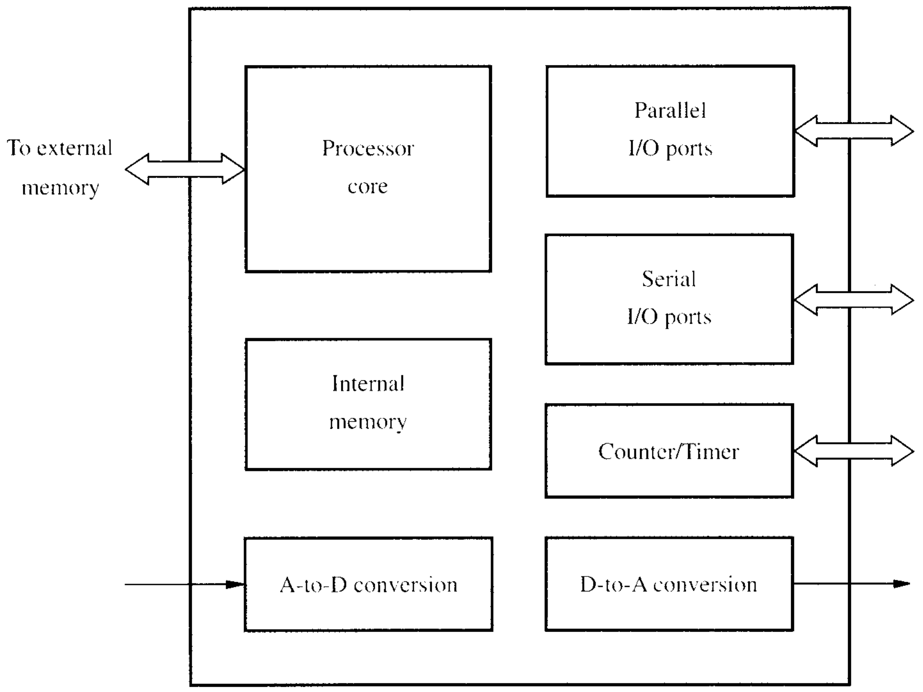

**Figure 9.3** A block diagram of an embedded processor.

of RAM type to hold the data that changes during computations. Some should be of ROM type to hold the software because an embedded system usually does not include a disk drive. To allow cost-effective use in low-volume applications, it is necessary to have a field-programmable type of ROM storage. Popular choices for realization of this storage are EEPROM and Flash memory.

Several I/O ports are usually provided for both parallel and serial interfaces. These interfaces allow easy implementation of standard I/O connections. In many applications, it is necessary to generate control signals at programmable time intervals. This task is achieved easily if a timer circuit is included in the embedded processor chip. Since the timer is a circuit that counts clock pulses, it can also be used for counting purposes, for example to count the number of pulses on a given input line.

An embedded system may include some analog devices. To deal with such devices, it is necessary to be able to convert analog signals into digital representations, and vice versa. This is conveniently accomplished if the embedded controller includes A/D and D/A conversion circuits.

Many embedded processor chips are available commercially. Some of the better known examples are: Motorola's 68HC11, 683xx and MCF5xxx families, Intel's 8051 and MCS-96 family, which use CISC-type processor cores, and ARM microcontrollers which have a RISC-type processor. The nature of the processor core is not important to our discussion in this chapter. We will emphasize the system aspects of embedded applications to illustrate how the concepts presented in the previous chapters fit together in the design of a complete embedded computer system.

## 9.3 A SIMPLE MICROCONTROLLER

In this section we discuss a possible organization of a simple microcontroller to illustrate how some typical features may be used in practice. Figure 9.4 gives its block diagram. There is a processor core and some on-chip memory. Since the on-chip memory may not be sufficient to support all potential applications, processor bus connections are also provided on the pins of the chip so that external memory may be added.

There are two 8-bit parallel interfaces, called A and B, and one serial interface. The microcontroller also contains a 32-bit counter/timer circuit, which can be used to generate internal interrupts at programmed time intervals, to serve as a system stop-watch, to count the pulses on an input line, to generate square-wave output signals of variable duty cycle, and so on.

### 9.3.1 PARALLEL I/O PORTS

The parallel interface provides I/O capability similar to the scheme depicted in Figure 4.34. Individual port lines on each of the A and B ports can be used as either inputs or outputs, as determined by the bit pattern stored in a data direction register. Figure 9.5 illustrates the bidirectional control for one bit in Port A. Port pin $PA_i$ is treated as an input if the data direction flip-flop contains a 0. In this case, an activation of the

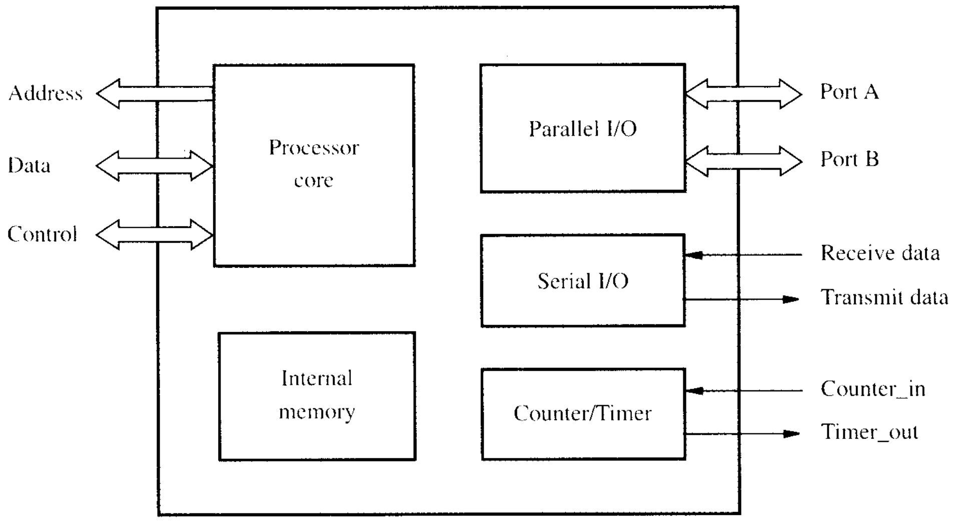

Figure 9.4 An example microcontroller.

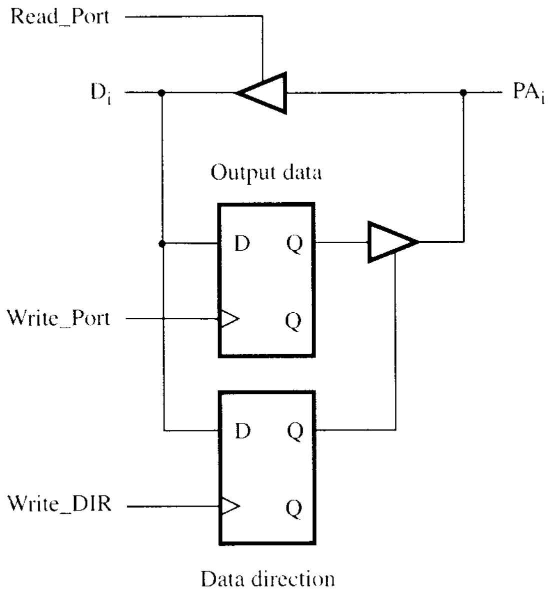

Figure 9.5 Access to one bit in Port A in Figure 9.4.

control signal Read_Port places the logic value on the port pin onto the data line $D_i$ of the processor bus. We have chosen not to include a storage element (corresponding to DATAIN in Figure 4.34) in the input path. Hence, the processor reads the data on the pins directly. The port pin serves as an output if the data direction flip-flop is set to 1. In this case, the logic value loaded into the data output flip-flop, under control of the Write_Port signal, is placed on the pin. Since a data direction bit is provided for each pin, some pins can be programmed as inputs and others can serve as outputs.

The data transfer operations on ports A and B involve the eight 8-bit registers depicted in Figure 9.6. The figure also gives the memory-mapped addresses assigned

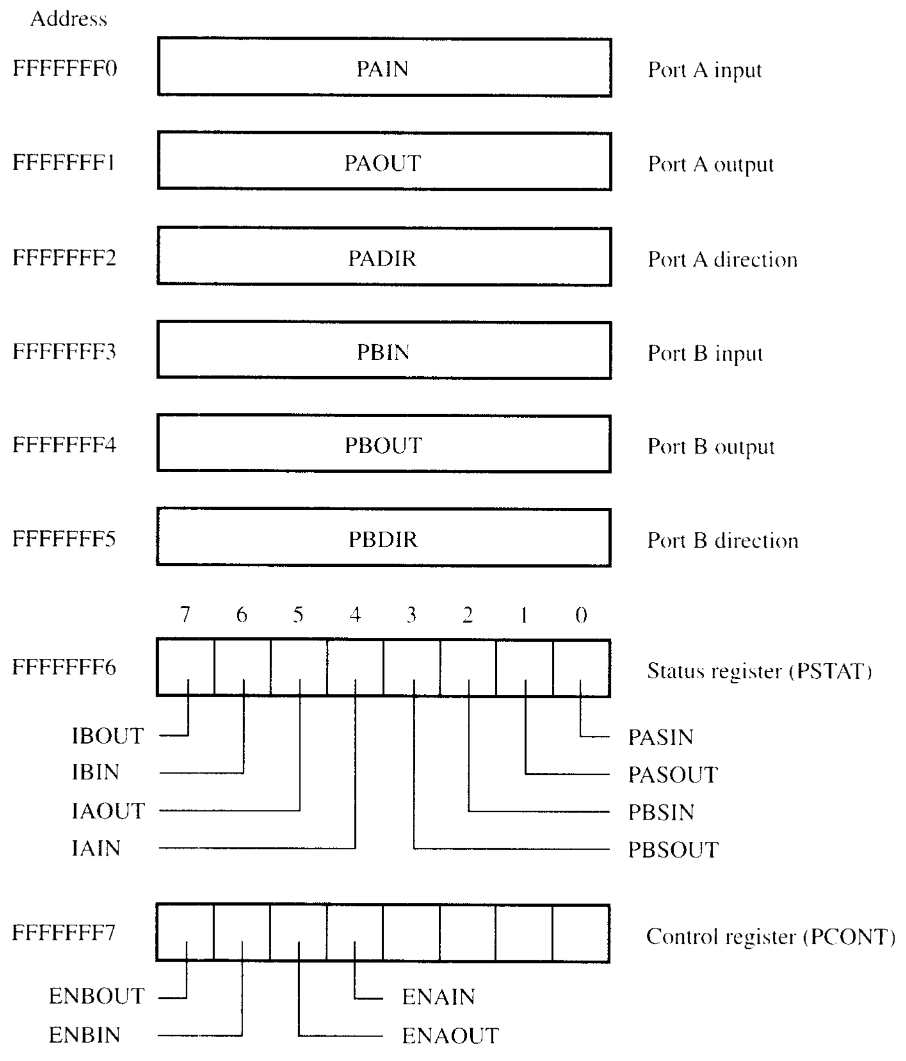

**Figure 9.6** Parallel interface registers.

to these registers. We have arbitrarily chosen the addresses at the high end of a 32-bit address range.

The status register, PSTAT, contains the status flags. The PASIN flag is set to 1 when there is new data on the pins of port A. It is cleared to 0 when the processor accepts the data by reading the PAIN register. The PASOUT flag is set to 1 when the data in register PAOUT are transferred to the connected device, to indicate that the processor may now load new data into PAOUT. (The transfer to the device is signaled on a control line as described below.) The PASOUT flag is cleared to 0 when the processor writes data into PAOUT. The flags PBSIN and PBSOUT perform the same function for port B.

The status register also contains four interrupt flags. An interrupt flag, such as IAIN, is set to 1 when that interrupt is enabled and the corresponding I/O action takes place. The interrupt enable bits are held in the control register, PCONT. An enable bit is set to 1 to enable the corresponding interrupt. For example, if ENAIN = 1 and PASIN = 1, then the interrupt flag IAIN is set to 1 and an interrupt request is raised. Thus,

$$\text{IAIN} = \text{ENAIN} \cdot \text{PASIN}$$

A single interrupt request signal is used. In response to an interrupt request, the processor must examine the interrupt flags to determine the actual source of the request.

Information in the status and control registers is used for controlling the data transfers to and from the devices connected to ports A and B. Port A has two control lines, *CAIN* and *CAOUT*, which can be used to provide a signaling mechanism between the interface and the attached device as follows. When the device places new data on the port's pins, it signifies this action by activating the *CAIN* line for one clock cycle. When the interface circuit sees *CAIN* = 1, it sets the status bit PASIN to 1. Later, this bit is cleared to 0 when the processor reads the input data. This action also causes the interface to send a pulse on the *CAOUT* line to inform the device that it may send new data to the interface. For an output transfer, the processor writes the data into the PAOUT register. This action clears the PASOUT bit to 0 and sends a pulse on the *CAOUT* line to inform the device that new data are available. When the device takes the data, it sends a pulse on the *CAIN* line to signify this action, which in turn sets PASOUT to 1. This signaling mechanism is operational when all data pins of a port have the same orientation, that is when the port serves as either an input or an output port. If some pins are selected as inputs and others as outputs, then neither the control lines nor the status and control registers contain meaningful information.

### 9.3.2 Serial I/O Interface

The serial interface provides the UART (Universal Asynchronous Receiver Transmitter) capability to transfer data based on the principle indicated in Figure 4.37. Double buffering is used in both the transmit and receive paths, as shown in Figure 9.7. The need for such buffering, as explained in the discussion of Figure 4.37, is to smooth out short bursts in I/O transfers.

Figure 9.8 shows the addressable registers of the serial interface. Input data are read from the 8-bit Receive buffer, and output data are loaded into the 8-bit Transmit buffer. The status register, SSTAT, provides information about the current status of the

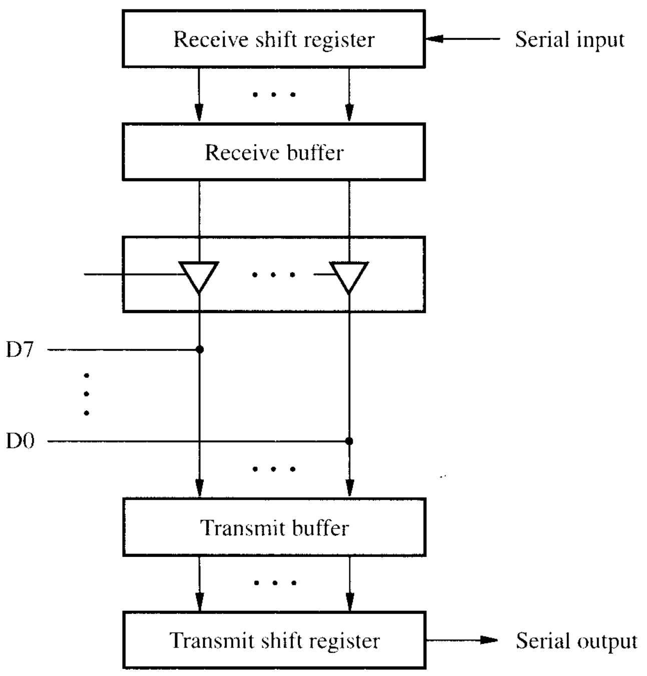

**Figure 9.7** Receive and transmit structure of the serial interface.

receive and transmit units. Bit $SSTAT_0$ is set to 1 when there are valid data in the receive buffer. Bit $SSTAT_1$ is set to 1 when the transmit buffer is empty and can be loaded with new data. These bits serve the same purpose as the status flags SIN and SOUT discussed in Section 4.1. Bit $SSTAT_2$ is set to 1 if an error occurs during the receive process. For example, an error occurs if the data in the receive buffer are overwritten by a subsequently received character before the first character is read by the processor. The status register also contains the interrupt flags. Bit $SSTAT_4$ is set to 1 when the receive buffer is full and the receiver interrupt is enabled. Similarly, $SSTAT_5$ is set to 1 when the transmit buffer is empty and the transmitter interrupt is enabled. The serial interface raises an interrupt if either $SSTAT_4$ or $SSTAT_5$ is equal to 1. It also raises an interrupt if $SSTAT_6 = 1$, which occurs if $SSTAT_2 = 1$ and the error condition interrupt is enabled.

The control register, SCONT, is used to hold the interrupt enable bits. Setting $SCONT_{6-4}$ bits to 1 or 0 enables or disables the corresponding interrupts, respectively. This register also indicates how the transmit clock is generated. If $SCONT_0 = 0$, then the transmit clock is the same as the system (processor) clock. If $SCONT_0 = 1$, then the transmit clock is obtained using a clock-dividing circuit.

The last register in the serial interface is the clock-divisor register, DIV. This 32-bit register is associated with a counter circuit that divides down the system clock signal to generate the serial transmission clock. The counter generates a clock signal whose

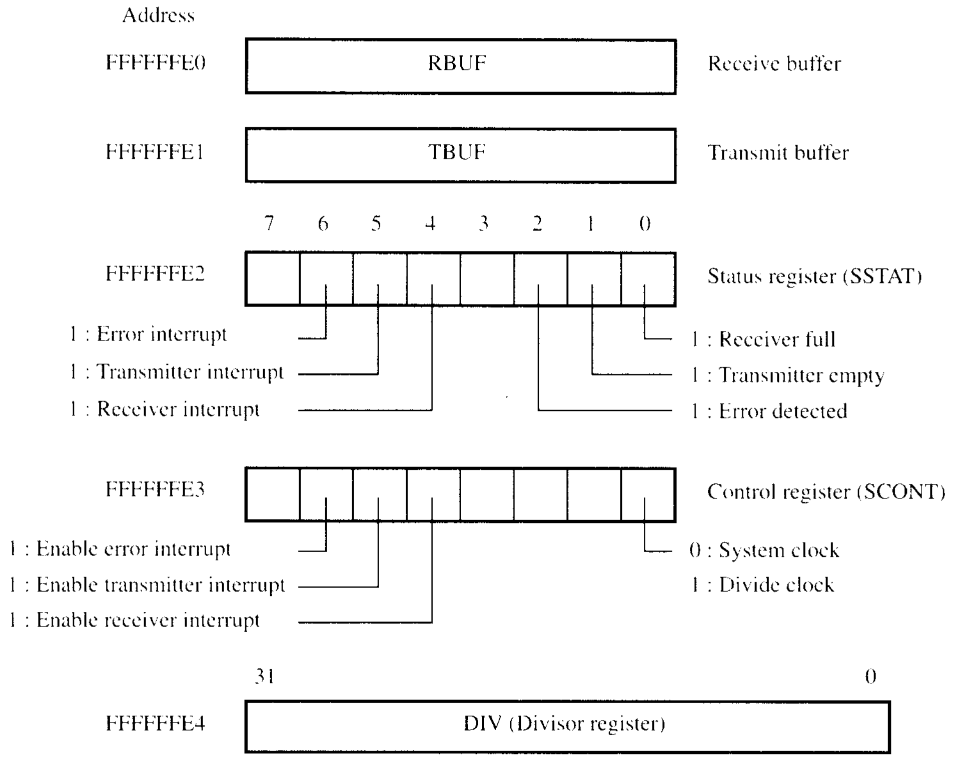

**Figure 9.8** Serial interface registers.

frequency is equal to the frequency of the system clock divided by the contents of this register. The value loaded into this register is transferred into the counter, which then counts down using the system clock. When the count reaches zero, the counter is reloaded using the value in the DIV register.

## 9.3.3 COUNTER/TIMER

A 32-bit down-counter circuit is provided for use as either a counter or a timer. The basic operation of the circuit involves loading a starting value into the counter, and then decrementing the counter contents using either the internal system clock or an external clock signal. The circuit can be programmed to raise an interrupt when the counter contents reach zero. Figure 9.9 shows the registers associated with the counter/timer circuit. The counter/timer register, CNTM, can be loaded with an initial value, which is then transferred into the counter circuit. The current contents of the counter can be read by accessing the memory address FFFFFFD4. The control register, CTCON, is used to specify the operating mode of the counter/timer circuit. It provides a mechanism for starting and stopping the counting process, and for enabling interrupts when the

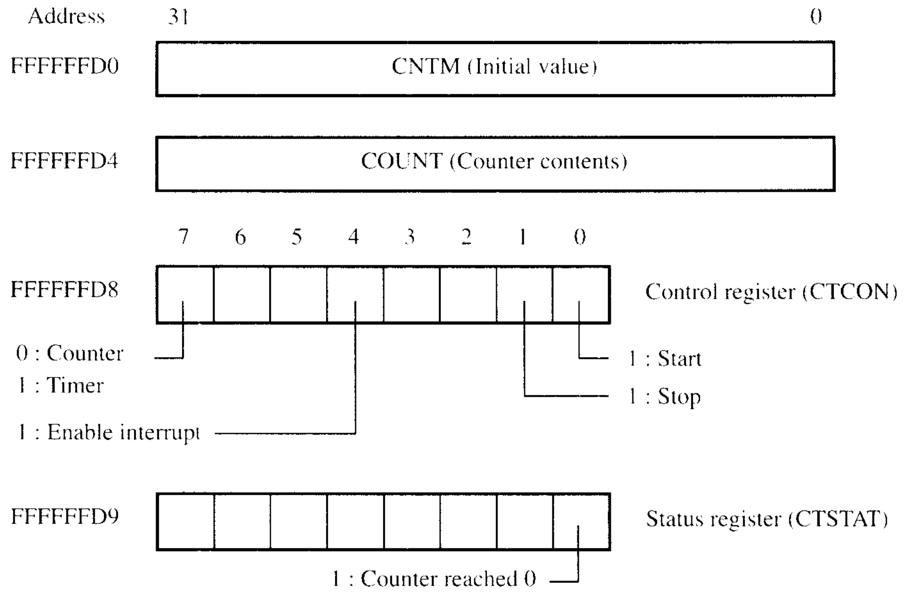

**Figure 9.9** Counter/Timer registers.

counter contents are decremented to 0. The status register, CTSTAT, reflects the state of the circuit.

### Counter Mode

The counter mode is selected by setting $CTCON_7$ to 0. The starting value is loaded into the counter by writing it into register CNTM. The counting process begins when bit $CTCON_0$ is set to 1 by a program instruction. Once the counter starts counting, the $CTCON_0$ bit is automatically cleared to 0. The counter is decremented by pulses on the *Counter_in* line. Upon reaching 0, the counter circuit sets the status flag $CTSTAT_0$ to 1, and will raise an interrupt if the corresponding interrupt enable bit has been set to 1. The next clock pulse causes the counter to reload the starting value, which is held in register CNTM, and counting continues. The counting process is stopped by setting $CTCON_1$ to 1.

### Timer Mode

The timer mode is selected by setting $CTCON_7$ to 1. This mode is suitable for generating a square-wave signal on the output line *Timer_out* in Figure 9.4. The process starts as explained above for the counter mode. As the counter counts down, the value on the output line is held constant. Upon reaching zero, the counter is reloaded automatically with the starting value, and the output signal on the line is inverted. Thus, the period of the output signal is twice the starting counter value times the period of the controlling clock pulse. In the timer mode, the counter is decremented by the system clock.

### 9.3.4 INTERRUPT CONTROL MECHANISM

The microcontroller has two interrupt request lines, *IRQ* and *XRQ*. The *IRQ* line is used for interrupts raised by the I/O interfaces within the microcontroller. The *XRQ* line is used for interrupts raised by external devices. When the processor observes that the *IRQ* line has been activated, it uses the polling method to determine the source(s) of the interrupt request. This is done by examining the flags in the status registers PSTAT, SSTAT and CTSTAT. The *XRQ* interrupts have higher priority than the *IRQ* interrupts.

The processor status register, PSR, has two bits for enabling interrupts. The IRQ interrupts are enabled if $PSR_6 = 1$, and the XRQ interrupts are enabled if $PSR_7 = 1$. When the processor accepts an interrupt, it disables further interrupts at the same priority level, by clearing the corresponding PSR bit before the interrupt service routine is executed. A vectored interrupt scheme is used, with the vectors for IRQ and XRQ interrupts in memory locations \$24 and \$28, respectively. Each vector contains the address of the first instruction of the corresponding interrupt service routine.

Our processor has a Link register, LR, which is used for subroutine linkage as explained in Section 2.9. A subroutine Call instruction causes the updated contents of the program counter, PC, which is the required return address, to be stored in LR prior to branching to the first instruction in the subroutine. The same action takes place when an interrupt request is accepted. But, in addition to saving the return address in LR, the contents of the processor status register, PSR, are saved in a processor register IPSR.

A return from subroutine is performed by a ReturnS instruction, which transfers the contents of LR into PC. A return from interrupt is performed by a ReturnI instruction, which transfers the contents of LR and IPSR into PC and PSR, respectively. Since there is only one LR and IPSR register, nested interrupts can be implemented by saving the contents of these registers on a stack using an instruction in the interrupt service routine. This scheme is similar to the ARM approach discussed in Section 4.3.1.

## 9.4 PROGRAMMING CONSIDERATIONS

Having introduced the microcontroller hardware, we will now consider some software aspects. Programs can be written either in assembly language or in a high-level language. The latter choice is preferable in most applications because the desired code is easier to generate and maintain, and development time is shorter. We will show some examples in both types of languages. The examples in this section are rudimentary and are intended to illustrate the possible approaches. In Section 9.5, we will give a more elaborate example of a complete application. We have chosen the C programming language as the high-level language.

Consider the following task. Our microcontroller is used to transfer 8-bit characters from a bit-serial source to a bit-parallel destination. The source is connected to the serial interface and the destination is connected to parallel port A. The presence of a character in the receive buffer is indicated by the corresponding bit in the status register being set to 1, that is, by $SSTAT_0 = 1$. The parallel port has to be configured for output, which is accomplished by setting all bits in the data direction register to 1 ($PADIR_{7-0}$ = \$FF). We will assume that the output device which receives the characters via Port A is faster than the source that delivers the characters to the serial interface. Hence, it is not necessary to poll Port A to see if it is ready to receive the next character. We will first show how to perform the desired transfer using polling. Then, we will implement the same task using interrupts.

## 9.4.1 POLLING APPROACH

Polling involves testing a status flag repeatedly until a character is received, as discussed in Section 2.7. In our example, it is necessary to poll the bit $SSTAT_0$.

### Assembly Language Program

Figure 9.10 shows how the task may be realized using assembly language. We use the generic format used in Chapter 2. The I/O registers in the microcontroller are referred to by symbolic names associated with the register addresses. The program loop continuously checks the status bit $SSTAT_0$. Whenever $SSTAT_0 = 1$, the character in the receive buffer, RBUF, is moved to Port A. Recall that performing a read operation on RBUF automatically clears the $SSTAT_0$ bit. If $PSTAT_1 = 1$, the character is written into the PAOUT register.

The program in Figure 9.10 uses an infinite loop to transfer a continuous stream of characters. In an actual application it is not likely that one would use an infinite loop in this manner because other tasks would also be involved. We use the infinite loop merely to keep the example simple.

```
RBUF     EQU         $FFFFFFE0        Receive buffer.
SSTAT    EQU         $FFFFFFE2        Status register for serial interface.
PAOUT    EQU         $FFFFFFF1        Port A output data.
PADIR    EQU         $FFFFFFF2        Port A direction register.

* Initialization
         ORIGIN      $1000
         MoveByte    #$FF,PADIR       Configure Port A as output.

* Transfer the characters
LOOP     Testbit     #0,SSTAT         Check if new character is ready.
         Branch=0    LOOP
         MoveByte    RBUF,PAOUT       Transfer a character to Port A.
         Branch      LOOP
```

**Figure 9.10** A generic assembly language program for character transfer using polling.

## C Program

In a C-language program, a memory-mapped I/O location can be represented using a pointer variable, where the value of the pointer is the address of the location. If the contents of this location are to be treated as a character, the pointer should be declared to be of character type. This defines the contents as being one byte in length, which is the size of the I/O registers. The contents can be conveniently handled in hexadecimal form.

Figure 9.11 shows a C program that implements our example task. The *define* statements are used to associate the required addresses with the symbolic names of the pointers. These statements serve the same purpose as the EQU statements in Figure 9.10. They enable the preprocessor of the C compiler to replace the symbolic names in the program with their actual values. As a result, the compiled code will be similar to the code in Figure 9.10.

Note that the RBUF and SSTAT pointers are declared as being volatile. This is necessary because the program only reads the contents of the corresponding locations, but it neither writes any data into them, nor associates a specific value with them. An optimizing compiler may remove program statements that appear to have no impact, which includes statements involving variables whose values never change. Since the contents of RBUF and SSTAT registers change under influences that are external to the program, it is essential to inform the compiler of this fact. The compiler will not remove the statements that contain variables that have been declared as being volatile.

A different approach is illustrated in Figure 9.12. Instead of defining the pointers to I/O registers as constants, as in Figure 9.11, the pointers are declared as variables pointing to locations that hold character-type data. Thus, symbols such as RBUF and

```
/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2

void main()
{
   /* Initialize the parallel port */
   *PADIR = 0xFF;                      /* Configure Port A as output */

   /* Transfer the characters */
   while (1) {                         /* Infinite loop */
      while ((*SSTAT & 0x1) == 0);     /* Wait for a new character */
      *PAOUT = *RBUF;                  /* Move the character to Port A */
   }
}
```

**Figure 9.11** C program for character transfer using polling.

```
/* Define register addresses */
volatile char *RBUF = (char *) 0xFFFFFFE0;
volatile char *SSTAT = (char *) 0xFFFFFFE2;
char *PAOUT = (char *) 0xFFFFFFF1;
char *PADIR = (char *) 0xFFFFFFF2;

void main()
{
    /* Initialize the parallel port */
    *PADIR = 0xFF;                       /* Configure Port A as output */

    /* Transfer the characters */
    while (1) {                          /* Infinite loop */
        while ((*SSTAT & 0x1) == 0);     /* Wait for a new character */
        *PAOUT = *RBUF;                  /* Move the character to Port A */
    }
}
```

**Figure 9.12** An alternative C program for character transfer using polling.

```
        Move        PADIR,R0
        MoveByte    #$FF,(R0)
LOOP    Move        SSTAT,R0
        Testbit     #0,(R0)
        Branch=0    LOOP
        Move        RBUF,R0
        Move        PAOUT,R1
        Move        (R0),(R1)
        Branch      LOOP
```

**Figure 9.13** Possible compiled code for the program segment in Figure 9.12.

PAOUT denote addresses of memory locations that hold the actual addresses of the I/O registers. In this case, the compiled code may appear as shown in Figure 9.13. To access a particular I/O register, its address is loaded into a processor register and then the Register indirect addressing mode is used to access the desired I/O register. Note that only a portion of the compiled code is shown in the figure. The compiler also indicates the memory address values that are associated with the symbols RBUF, SSTAT, PAOUT, and PADIR. The machine code generated from this figure is larger than the code derived from the program in Figure 9.11. In subsequent examples we will define the pointers as shown in Figure 9.11. Using the *define* statements to specify the addresses of I/O locations emphasizes the fact that these addresses are constants that never change in the course of execution of a program.

In the C programs in this chapter, we include the specific values that have to be loaded into registers directly in the statements that perform the desired actions. For example, in Figure 9.11 the statement

*PADIR = 0xFF;

sets all eight bits in register PADIR to 1, making Port A behave as an output port. A more usual approach in writing C programs is to declare names for such constant values and then use the names in the rest of the program. Our choice is motivated by a desire to keep the figures as small as possible and to make it easier to compare the given values with the specification of I/O registers in Figures 9.6 through 9.9.

## 9.4.2 INTERRUPT APPROACH

Instead of polling the bit $SSTAT_0$ to detect a new character, we can configure the I/O interface to raise an interrupt request when $SSTAT_0 = 1$. The corresponding interrupt enable bit in the SCONT register, $SCONT_4$, has to be set to 1. It is also necessary to enable IRQ interrupts in the processor by setting $PSR_6$ to 1. This is accomplished by loading $40 into PSR, which also disables the XRQ interrupts. The address of the interrupt service routine has to be placed in memory location $24.

### Assembly Language Program

Figure 9.14 shows an assembly language program that implements the interrupt scheme for the character transfer example. An interrupt request is raised when $SSTAT_0 =$ 1. In response, the interrupt service routine transfers a character from RBUF to PAOUT. The action of reading the contents of the receive buffer RBUF also causes the $SSTAT_0$ bit to be cleared to 0. Note that there is no reference to the status register SSTAT in the program. Again, as in the previous example, an infinite loop is used to wait for new characters to keep the example simple.

### C Program

To write a C program that uses interrupts we need to address two issues:

- How do we access processor registers?
- How do we write an interrupt service routine?

The interrupt approach requires the interrupt control bits in the processor status register to be set appropriately. This involves writing into PSR. Variables in a high-level language, such as C, are represented in the compiled code by memory locations. Hence, memory-mapped I/O registers can be handled in a straightforward manner, as we have done in Figures 9.11 and 9.12. However, the processor registers, such as the status register PSR, do not have memory addresses associated with them. These registers can be accessed by including suitable assembly language instructions directly in the

```
RBUF      EQU        $FFFFFFE0       Receive buffer.
SCONT     EQU        $FFFFFFE3       Control register for serial interface.
PAOUT     EQU        $FFFFFFF1       Port A output data.
PADIR     EQU        $FFFFFFF2       Port A direction register.

* Initialization
          ORIGIN     $1000
          MoveByte   #$FF,PADIR      Configure Port A as output.
          Move       #INTSERV,$24    Set the interrupt vector.
          Move       #$40,PSR        Processor responds to IRQ interrupts.
          MoveByte   #$10,SCONT      Enable receiver interrupts.

* Transfer loop
LOOP      Branch     LOOP            Infinite wait loop.

* Interrupt service routine
INTSERV   MoveByte   RBUF,PAOUT      Transfer a character to Port A.
          ReturnI                    Return from interrupt.
```

**Figure 9.14** A generic assembly language program for character transfer using interrupts.

C program. For example, the statement

```
__asm__("Move #0x40,%PSR");
```

causes the C compiler to insert the assembly instruction

```
Move #$40,PSR
```

into the compiled code. This loads the pattern $40 into the PSR register.

The second issue is the interrupt service routine. This routine has to be written as a function in the C program. However, the compiler implements a function as a subroutine. Figure 9.15 gives an example. There is a main program that performs some task; we have not shown any statements of this program. There is also a function named *intserv*, which merely transfers one character from the receive buffer RBUF to the output port PAOUT. The compiler-generated code for the function *intserv* comprises the instructions

```
Move      $FFFFFFE0,$FFFFFFF1
ReturnS
```

Consider now what happens if *intserv* is an interrupt service routine. Using interrupts, the return from the interrupt service routine must be done using the return from interrupt instruction, ReturnI. It causes the contents of both the program counter and the processor status register to be restored to their previous values. Therefore, we have to insert the ReturnI instruction in the program by the statement

```
__asm__("ReturnI");
```

```
#define RBUF (volatile char *) 0xFFFFFFE0
#define PAOUT (char *) 0xFFFFFFF1
        .
        .
        .

void main()
{
        .
        .
        .
}

void intserv()
{
   *PAOUT = *RBUF;    /* Move a character to Port A */
}
```

**Figure 9.15** A function call in a C program.

so that the compiled code will be

```
Move      $FFFFFFE0,$FFFFFFF1
ReturnI
ReturnS
```

Of course, the ReturnS instruction will never be executed in this case.

We can now write the desired program that uses interrupts. Figure 9.16 gives a possible program, using the style of Figure 9.11. Note that the pointers to I/O registers are of character type because they point to locations that hold one byte of data. However, the pointer int_addr is of unsigned integer type because it points to a memory location that stores a 4-byte interrupt vector.

## 9.5 I/O DEVICE TIMING CONSTRAINTS

The example in the previous section is made simple by the assumption that the output device connected to Port A is faster than the device that provides the characters via the serial interface. Practical systems involve devices with varying speed requirements. Suppose that we reverse our speed assumption for the previous example, namely, let the output device be slower than the input device. This means that characters cannot be sent directly from RBUF to PAOUT. Instead, it is necessary to store them temporarily in a memory buffer. This buffer may be organized as a first-in-first-out (FIFO) queue.

```
/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFFE0
#define SCONT (char *) 0xFFFFFFE3
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define int_addr (int *) (0x24)

void intserv();

void main()
{
   /* Initialize the parallel port */
   *PADIR = 0xFF;                   /* Configure Port A as output */

   /* Initialize the interrupt mechanism */
   int_addr = &intserv;             /* Set interrupt vector */
   __asm__("Move #0x40,%PSR");      /* Processor responds to IRQ interrupts */
   *SCONT = 0x10;                   /* Enable receiver interrupts */

   /* Transfer the characters */
   while (1);                       /* Infinite loop */
}

/* Interrupt service routine */
void intserv()
{
   *PAOUT = *RBUF;                  /* Transfer the character to Port A */
   __asm__("ReturnI");              /* Return from interrupt */
}
```

**Figure 9.16** C program for character transfer using interrupts.

The problem with a simple FIFO is that the pointers that point to the head and the tail of the queue are progressively incremented, so that the queue moves through the memory as the characters are stored and retrieved. A better solution is to use a circular buffer, that is, a circular queue, which consists of a fixed number of memory locations and wraps around when the end of the buffer is reached. Of course, this buffer can overflow if the fast source generates many more characters in a burst than the output device can accept during the burst time period. To avoid dealing with the overflow problem at this point, we will assume that the source generates the characters in bursts of less than 80 characters and that the output device will accept these characters before another burst arrives. This means that the circular buffer has to be able to hold up to 80 characters.

We can implement the circular buffer as an array of 8-bit entries, using two index values to denote the current position of the head and the tail of the queue. When these indexes are equal, the queue is empty.

### 9.5.1 C PROGRAM FOR TRANSFER VIA A CIRCULAR BUFFER

Figure 9.17 gives a possible C program that uses the circular buffer to transfer characters. The buffer is an array called *mbuffer*, and it has 80 entries. Characters are placed into the buffer using index *fin*, and they are retrieved using index *fout*. On each pass through the loop, the status register SSTAT is tested first to see if there is a new character

```
/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PSTAT (volatile char *) 0xFFFFFFF6
#define BSIZE 80

void main()
{
   unsigned char mbuffer[BSIZE];
   unsigned char fin, fout;
   unsigned char temp;

   /* Initialize Port A and circular buffer */
   *PADIR = 0xFF;                          /* Configure Port A as output */
   fin = 0;
   fout = 0;

   /* Transfer the characters */
   while (1) {                             /* Infinite loop */
      while ((*SSTAT & 0x1) == 0) {        /* Wait for a new character */
         if (fin != fout) {                /* If circular buffer is not empty */
            if (*PSTAT & 0x2) {            /*   and output device is ready */
               *PAOUT = mbuffer[fout];     /*   send a character to Port A */
               if (fout < BSIZE-1)         /* Update the output index */
                  fout++;
               else
                  fout = 0;
            }
         }
      }
      mbuffer[fin] = *RBUF;                /* Read a character from receive buffer */
      if (fin < BSIZE-1)                   /* Update the input index */
         fin++;
      else
         fin = 0;
   }
}
```

**Figure 9.17** C program for transfer through a circular buffer.

in RBUF, because transfers from the faster device must be given higher priority. If there is no character in RBUF, then a transfer to Port A is performed if the circular buffer is not empty and the port is ready. The index values are updated as each transfer is made.

## 9.5.2 Assembly Language Program for Transfer via a Circular Buffer

Figure 9.18 shows an implementation in assembly language. The circular buffer is accessed using the indexed addressing mode. Register R0 points to the first location in

```
RBUF        EQU          $FFFFFFE0          Receive buffer.
SSTAT       EQU          $FFFFFFE2          Status register for serial interface.
PAOUT       EQU          $FFFFFFF1          Port A output data.
PADIR       EQU          $FFFFFFF2          Port A direction register.
PSTAT       EQU          $FFFFFFF6          Status register for parallel interface.
MBUFFER     ReserveByte  80                 Define the circular buffer.

* Initialization
            ORIGIN       $1000
            MoveByte     #$FF,PADIR         Configure Port A as output.
            Move         #MBUFFER,R0        R0 points to the buffer.
            Move         #0,R1              Initialize head pointer.
            Move         #0,R2              Initialize tail pointer.

* Transfer the characters
LOOP        Testbit      #0,SSTAT           Check if new character is ready.
            Branch≠0     READ
            Compare      R1,R2              Check if queue is empty.
            Branch=0     LOOP               Queue is empty.
            Testbit      #1,PSTAT           Check if Port A is ready.
            Branch=0     LOOP
            MoveByte     (R0,R2),PAOUT      Send a character to Port A.
            Add          #1,R2              Increment the tail pointer.
            Compare      #80,R2             Is the pointer past queue limit?
            Branch<0     LOOP
            Move         #0,R2              Wrap around.
            Branch       LOOP
READ        MoveByte     RBUF,(R0,R1)       Place new character into queue.
            Add          #1,R1              Increment the head pointer.
            Compare      #80,R1             Is the pointer past queue limit?
            Branch<0     LOOP
            Move         #0,R1              Wrap around.
            Branch       LOOP
```

**Figure 9.18** A generic assembly language program for transfer through a circular buffer.

the queue, and registers R1 and R2 are the head and tail indexes, respectively. The flow in the program is essentially the same as in the program in Figure 9.17.

## 9.6 Reaction Timer — An Example

Having introduced the basic features of the microcontroller, we will now show how it can be used in a simple application. Instead of using an example of a typical embedded application, which would involve considerable complexity, our choice is a simple and easily understood task.

We want to design a "reaction timer" that can be used to measure the speed of response of a person to a visual stimulus. The idea is to have the microcontroller turn on a light and then measure the reaction time that the subject takes to turn the light off by pressing a switch. The detailed operation should be as follows:

- There are two manual pushbutton switches, *Go* and *Stop*, a light-emitting diode (LED) and a three-digit seven-segment display.
- The system is activated by pressing the *Go* switch.
- Upon activation, the seven-segment display is set to 000 and the LED is turned off.
- After a three-second delay, the LED is turned on and the timing process begins.
- When the *Stop* switch is pressed, the timing process is stopped, the LED is turned off, and the elapsed time is displayed on the seven-segment display.
- The elapsed time is calculated and displayed in hundredths of a second. Since the display has only three digits, it is assumed that the elapsed time will be less than ten seconds.

Figure 9.19 depicts the hardware that can implement the desired reaction timer. The microcontroller provides all of the functionality except for the input switches and the output displays.

We will use the parallel ports, A and B, for all input/output functions. The two most-significant BCD digits of the displayed time are connected to Port A, and the least-significant digit to the upper four bits of Port B. The switches and the LED are connected to the lower four bits of Port B as shown in the figure. The counter/timer circuit is used to measure the elapsed time. It is driven by the system clock, which we assume to have a frequency of 100 MHz.

A program to realize the required task can be based on the following approach:

- The user's intention to begin a test is monitored by means of a wait loop that polls the state of the *Go* switch.
- Upon observing that the *Go* switch has been closed, that is, having detected $PB_1 = 0$, and after a further delay of three seconds, the LED is turned on.
- The counter is set to the initial value \$FFFFFFFF and the counting process is activated; the count is decremented by each clock pulse.
- A wait loop polls the state of the *Stop* switch to detect when the user reacts by pressing the switch.

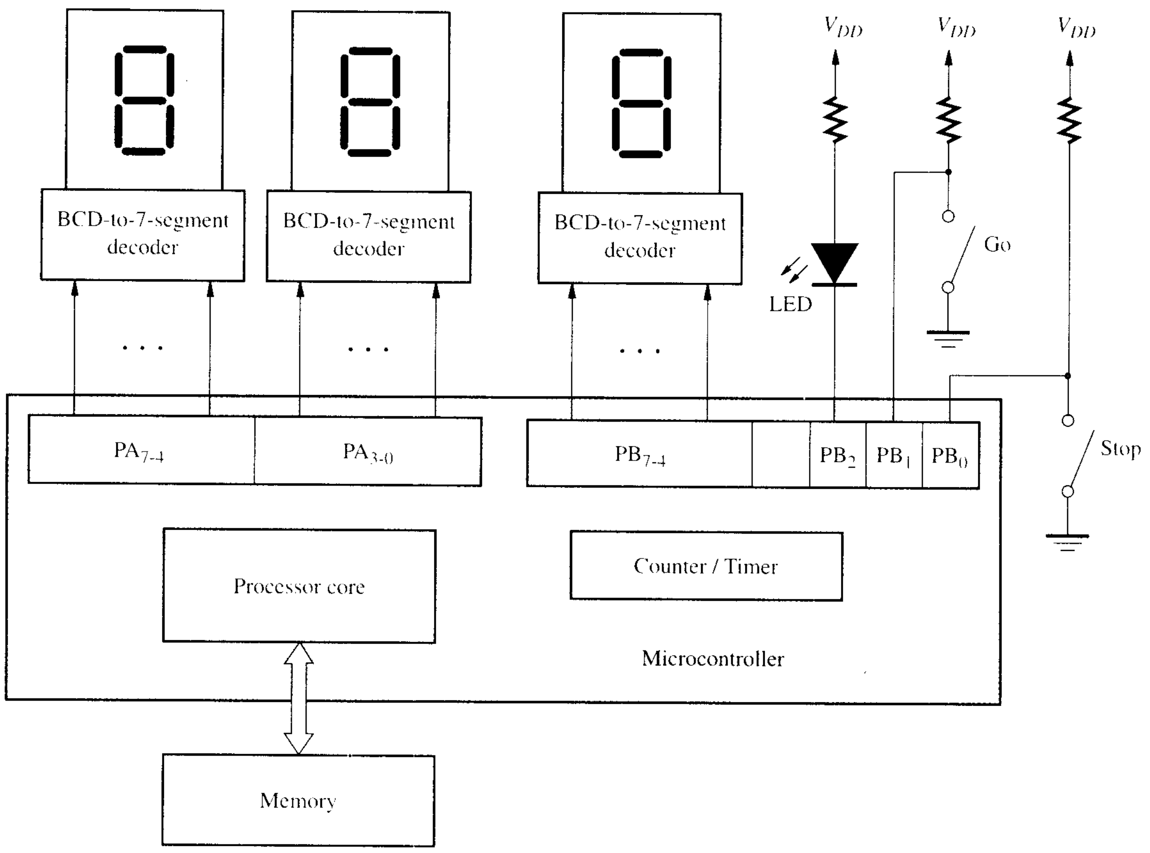

**Figure 9.19** The reaction-timer circuit.

- When the *Stop* switch is pressed, the counter is stopped and the elapsed time is calculated.
- The measured delay is converted into a BCD number and sent to the seven-segment displays.

The addresses of various I/O registers in the microcontroller are as given in Figures 9.6 through 9.9. The program must configure Ports A and B as specified in Figure 9.19. All of Port A and the high-order four bits of Port B are configured as outputs. In the low-order four bits of Port B, $PB_0$ and $PB_1$ are used as inputs, while $PB_2$ is an output. There is no need to use the control signals available on the two ports, because the input device consists of simple switches, and the output device is a display that immediately follows any changes in signals on the port pins that drive it.

We will show how the required task can be implemented using the C programming language. Then, we will show how it may be realized in assembly language.

In both programs there are instructions that perform the following tasks. After the *Go* key is pressed, a delay of three seconds is implemented by using the timer. Since the counter/timer circuit is clocked at 100 MHz, the counter is initialized to the hex value 11E1A300, which corresponds to the decimal value 300,000,000. The process of counting down is started by setting the $CTCONT_0$ bit to 1. When the count reaches zero, the LED is turned on to begin the reaction time test. Next, the counter is set to FFFFFFFF to begin the reaction timing. Upon detecting that the *Stop* key has been pressed, the counting process is stopped by setting $CTCONT_1 = 1$. The total count is computed as

$$\text{Total count} = \text{0xFFFFFFFF} - \text{Present count}$$

Since this is the total number of clock cycles, the actual time in hundredths of seconds is

$$\text{Actual time} = (\text{Total count})/1000000$$

This binary integer can be converted to a decimal number by first dividing it by 100 to generate the most-significant digit. The remainder is then divided by 10 to generate the next digit. The final remainder is the least-significant digit.

### 9.6.1 C Program for the Reaction Timer

Figure 9.20 gives a possible program. Having configured Ports A and B as required, and having turned off the display and LED, the program polls the value on pin $PB_1$. After the *Go* key is pressed and $PB_1$ becomes equal to 1, a three-second delay is introduced. Then, the LED is turned on, and the reaction timing process starts. Another polling operation is used to wait for the *Stop* key to be pressed. When this key is pressed, the LED is turned off, the counter is stopped, and its contents are read. The computation of the elapsed time and the conversion to a decimal number are performed as explained above. The resulting three BCD digits are written to the port data registers according to the arrangement depicted in Figure 9.19.

### 9.6.2 Assembly Language Program for the Reaction Timer

Figure 9.21 shows how the program for the reaction timer can be realized using assembly language. It is based on the general strategy outlined at the beginning of Section 9.6, and it parallels the C program in Figure 9.20. The comments given with each instruction should enable the reader to follow the flow of the program.

The conversion to BCD representation involves using the Divide instruction, discussed in Section 2.10.3. This instruction divides a dividend in register R2 by a divisor in register R1. The result of the operation is a quotient placed in R2 and a remainder placed in R3. Note that after dividing the actual time (in hundredths of seconds) by 100, only the least-significant four bits of the quotient in R2 need to be considered. We have assumed that the elapsed time will be less than ten seconds, so that only three digits have to be displayed. The first digit of the quotient is moved to R4. The remainder is moved from R3 to R2, and the division by ten is performed next.

```
/* Define register addresses */
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBIN (volatile char *) 0xFFFFFFF3
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5
#define CNTM (int *) 0xFFFFFFD0
#define COUNT (volatile int *) 0xFFFFFFD4
#define CTCON (char *) 0xFFFFFFD8

void main()
{
   unsigned int counter_value, total_count;
   unsigned int actual_time, seconds, tenths, hundredths;

   /* Initialize the parallel ports */
   *PADIR = 0xFF;                       /* Configure Port A */
   *PBDIR = 0xF4;                       /* Configure Port B */
   *PAOUT = 0x0;                        /* Turn off display */
   *PBOUT = 0x4;                        /* and LED */

   /* Start the test */
   while (1) {                          /* Infinite loop */
      while ((*PBIN & 0x2) == 0);       /* Wait for Go key to be pressed */

      /* Wait 3 seconds and then turn LED on */
      *CNTM = 0x11E1A300;               /* Set timer value to 300,000,000 */
      *CTCONT = 0x1;                    /* Start the timer */
      while ((*CTSTAT & 0x1) == 0);     /* Wait until timer reaches zero */
      *PBOUT = 0x0;                     /* Turn LED on */

      /* Initialize the counting process */
      counter_value = 0;
      *CNTM = 0xFFFFFFFF;               /* Set the starting counter value */
      *CTCONT = 0x1;                    /* Start counting */

      while ((*PBIN & 0x1) == 0);       /* Wait for the Stop key to be pressed */

      /* The Stop key has been pressed - stop counting */
      *PBOUT = 0x4;                     /* Turn LED off */
      *CTCONT = 0x2;                    /* Stop the counter */
      counter_value = *COUNT;           /* Read the contents of the counter */
```

**Figure 9.20** C program for the reaction timer.

```
        /* Compute the total count */
        total_count = (0xFFFFFFFF - counter_value);

        /* Convert count to time */ ;
        actual_time = total_count / 1000000;        /* Time in hundredths of seconds */
        seconds = actual_time / 100;
        tenths = (actual_time - seconds * 100) / 10;
        hundredths = actual_time - (seconds * 100 + tenths * 10);

        /* Display the elapsed time */
        *PAOUT = ((seconds << 4) | tenths);
        *PBOUT = ((hundredths << 4) | 0x4);     /* Keep the LED turned off */
    }
}
```

**Figure 9.20** *(Continued)*

```
PAOUT    EQU         $FFFFFFF1           Port A output data.
PADIR    EQU         $FFFFFFF2           Port A direction register.
PBIN     EQU         $FFFFFFF3           Port B input pins.
PBOUT    EQU         $FFFFFFF4           Port B output data.
PBDIR    EQU         $FFFFFFF5           Port B direction register.
CNTM     EQU         $FFFFFFD0           Initial counter value.
COUNT    EQU         $FFFFFFD4           Counter contents.
CTCONT   EQU         $FFFFFFD8           Control register.

* Initialization
         ORIGIN      $1000
         MoveByte    #$FF,PADIR          Configure Port A.
         MoveByte    #$F4,PBDIR          Configure Port B.

START    MoveByte    #0,PAOUT            Turn the display off.
         MoveByte    #4,PBOUT            Turn the LED off, by making PB2 = 1.

* Wait for Go key to be pressed
GKEY     Testbit     #1,PBIN             Go key is connected to PB1 pin.
         Branch=0    GKEY

* Delay 3 seconds before the LED is turned on
         Move        #11E1A300,CNTM      Timer value is 300,000,000.
         MoveByte    #1,CTCONT           Start the timer.
```

**Figure 9.21** Assembly language program for the reaction timer using polling.

```
DELAY   Testbit     #0,CTCONT            Wait until timer
        Branch=0    DELAY                  reaches zero.
        MoveByte    #0,PBOUT             Turn the LED on.

* Initialize the counting process
        Move        #$FFFFFFFF,CNTM      Set the starting counter value.
        MoveByte    #1,CTCONT            Start counting.

* Wait for Stop key to be pressed
SKEY    Testbit     #0,PB                Stop key is connected
        Branch=0    SKEY                   to PB0 pin.

* Stop counting and read the last count
        MoveByte    #4,PBOUT             Turn the LED off, by making PB2 = 1.
        MoveByte    #2,CTCONT            Stop the counter.
        Move        COUNT,R0             Read the counter.

* Compute the total count
        Move        #$FFFFFFFF,R2        Determine the
        Subtract    R0,R2                  actual count.

* Convert the count to actual time in hundredths of seconds, and then to BCD.
* Put the 2 most-significant BCD digits in R4 and the least-significant digit in R3.
        Move        #1000000,R1          Determine the count in
        Divide      R1,R2                  hundredths of seconds.
        Move        #100,R1              Divide by 100 to find the digit that
        Divide      R1,R2                  denotes the number of seconds.
        Move        R2,R4                Save the digit in R4.
        Move        R3,R2                Use the remainder as the next dividend.
        Move        #10,R1               Divide by 10 to find the digit that
        Divide      R1,R2                  denotes 1/10th of a second.
        LShiftL     #4,R4                The first 2 BCD digits
        Or          R2,R4                  are placed in R4.

* Display the elapsed time
        MoveByte    R4,PAOUT             First 2 digits to Port A.
        LShiftL     #4,R3                Third digit to Port B
        Or          #4,R3                  and keep the LED
        MoveByte    R3,PBOUT               turned off.
        Branch      START                Ready for next test.
```

**Figure 9.21** *(Continued)*

The BCD digits that denote the seconds and the tenths of a second of the elapsed time are placed into the least-significant byte of R4 to be sent to Port A for display. The third digit, denoting the hundredths of a second, is shifted in R3 into bit positions $R3_{7-4}$. Since the contents of R3 are sent to Port B, it is also necessary to set $R3_3 = 1$ to keep the LED turned off.

### 9.6.3 FINAL COMMENTS

The reaction timer is a fairly complete example of a computer-controlled application. This is in contrast to the examples in the previous chapters, which typically illustrated isolated features of a computer system.

An important aspect of software written for embedded systems is that it has to interact closely with hardware. The term *reactive system* is often used to describe the fact that the points in time at which various routines are executed are determined by events external to the processor, such as the closing of a switch or the arrival of a character at an input port. As discussed in Chapter 4, two mechanisms are available to coordinate this interaction — polling and interrupts. The software designer must decide how this interaction will be achieved.

## 9.7 EMBEDDED PROCESSOR FAMILIES

Having presented an example of an embedded processor in Section 9.3 and a simple application in Section 9.6, we now briefly consider the broad range of commercially available chips. Many embedded applications do not require a powerful processor. Certainly, the microwave oven discussed in Section 9.1.1 does not need a powerful controller because its computational requirements are rather modest. For such applications it is preferable to use a chip that has a simpler processor but which contains sufficient memory resources so that this is the only chip needed to implement all controller functions. The digital camera, discussed in Section 9.1.2, has much more demanding computational requirements, which dictates using a more powerful processor.

Processors may be characterized by how many data bits they handle in parallel when accessing data in the memory. The most powerful microcontrollers are likely to include a 32-bit processor, with a 32-bit-wide data bus. Such is the case with some microcontrollers based on the ARM architecture. It is also possible to have a processor that has an internal 32-bit structure, but a 16-bit-wide data bus to memory. An example is Motorola's 683xx family of microcontrollers, which use a 68000-based processor core. Such devices are classified as 16-bit microcontrollers. The most popular microcontrollers are 8-bit chips. They are much cheaper, yet powerful enough to satisfy the requirements of a large number of embedded applications. There exist even smaller 4-bit chips, which are attractive due to their simplicity and extremely low cost.

### 9.7.1 MICROCONTROLLERS BASED ON THE INTEL 8051

In the early 1980s, Intel Corporation introduced a microcontroller chip called the 8051. This chip has the basic architecture of Intel's 8080 microprocessor family, which used 8-bit chips for general-purpose computing applications. The 8051 chip gained rapid popularity. It has become one of the most widely used chips in practice. The 8051 has four 8-bit I/O ports, a UART, and two 16-bit counter/timer circuits. It also contains 4K bytes of ROM and 128 bytes of RAM storage. The EPROM version of the same chip, in which there are 4K bytes of EPROM rather than ROM, is available under the name 8751.

A number of other chips that are based on the 8051 architecture are available; they provide various enhancements. For example, the 8052 chip has 8K bytes of ROM and 256 bytes of RAM, as well as an extra counter/timer circuit. Its EPROM version is called 8752.

Microcontroller chips may be fabricated using either NMOS or CMOS technology. The CMOS devices have the advantage of consuming less power, hence they are particularly attractive for battery-driven applications. The CMOS versions of the above microcontrollers are known as 80C51 and 80C52.

The 8051 architecture was developed by Intel. Subsequently, a number of other semiconductor manufacturers have produced chips that are either identical to those in the 8051 family or have some enhanced features but are fully compatible. From the point of view of the designer of an embedded application, it is useful to have second sources of the chips used. This ensures a higher degree of availability of the chips and competitive prices.

### 9.7.2 MOTOROLA MICROCONTROLLERS

Intel and Motorola had the dominant positions as manufacturers of microprocessor chips in the 1980s. Their most popular 8-bit microprocessors became the basis for their microcontrollers. There is a wide range of Motorola microcontrollers, based on different processor cores.

#### 68HC11 Microcontroller

Motorola's most popular 8-bit microprocessors were the 6800 and the 6809. A microcontroller chip that implements instructions that are a superset of 6800 instructions is known as the 68HC11. It has five I/O ports that can be used for a variety of purposes. The I/O structure includes two serial interfaces, one asynchronous and one synchronous. The asynchronous interface uses the start-stop protocol discussed in Section 10.3.1. The synchronous interface implements a scheme known as the *serial peripheral interface* (SPI). It is possible to connect up to eight analog inputs to the 68HC11 because the chip has the capability to perform A/D conversion. Finally, there are counter/timer circuits capable of operating in a number of different modes.

The amount of memory included on 68HC11 chips depends on the particular model. This typically ranges from an 8K-byte ROM, a 512-byte EEPROM, and a 256-byte

RAM in the original chip, to a 12K-byte ROM, a 512-byte EEPROM, and a 512-byte RAM in more advanced chips.

### 683xx Microcontrollers

A family of microcontrollers, known as 683xx, is based on the 68000 processor core. These chips include parallel and serial ports, counter/timer capability, and A/D conversion circuits. The amount of on-chip memory depends on the particular chip. For example, the 68376 chip has an 8K-byte EEPROM and a 4K-byte RAM.

### ColdFire Microcontrollers

The 68000 instruction set architecture provides the basis for the MCF5xxx microcontrollers, known as the ColdFire embedded processors. Their distinguishing feature is a pipelined structure that leads to enhanced performance. They implement a full 32-bit bus. The ColdFire processor core is also intended for use in the system-on-a-chip environment, which is discussed in Section 9.9.

### PowerPC Microcontrollers

Motorola's high-end microprocessors, known as PowerPC, are based on a RISC-style architecture. This processor architecture is also available in the microcontroller form, in chips comprising the MPC5xx family.

## 9.7.3 ARM Microcontrollers

The ARM architecture, presented in Chapter 3, is attractive for embedded systems where substantial computing capability is needed and the cost and power consumption have to be relatively low. A key objective has been to make the ARM processor designs suitable for use in the system-on-a-chip environment. The ARM microcontrollers are also available as separate chips.

There has been a progression of ARM processor cores intended for embedded applications, including ARM6, ARM7, ARM9, and ARM10. The basic ARM architecture uses a 32-bit organization and an instruction set in which all instructions are 32 bits long. There exists another version, known as Thumb, which uses 16-bit instructions and 16-bit data transfers. The Thumb version uses a subset of the ARM instructions, which are encoded to fit into a 16-bit format. It also has fewer registers than the ARM architecture. An advantage of Thumb is that a significantly smaller memory is needed to store programs that comprise highly encoded 16-bit instructions. At execution time, each Thumb instruction is expanded into a normal 32-bit ARM instruction. Thus, a Thumb-aware ARM core contains a *Thumb decompressor* in addition to its normal circuits.

The ARM architecture and processor cores have been developed by Advanced RISC Machines Ltd. A number of other companies are also licensed to provide these cores. Some companies, such as Atmel Corp., Sharp Electronics Corp., and Samsung Semiconductor Inc., market microcontroller chips based on ARM cores. For example, Atmel's AT91F40416 microcontroller uses the ARM7-TDMI Thumb-aware core, and it also contains 4K bytes of RAM, 526K bytes of Flash ROM, 32 programmable I/O lines, 2 serial ports, and a counter/timer circuit.

## 9.8 DESIGN ISSUES

The designer of an embedded system has to make many important decisions. The nature of the application or the product that has to be designed presents certain requirements and constraints. In this section, we will consider some of the most important issues that the designer faces.

### Cost

The cost of electronics in many embedded applications has to be low. The cheapest solution is realized if a single microcontroller chip suffices for the implementation of all functions that must be provided. This is possible only if there is sufficient I/O capability and enough on-chip memory to hold the necessary software.

### I/O Capability

Microcontroller chips provide a variety of I/O resources. This ranges from having simple parallel and serial ports to extensive support that includes counter, timer, A/D and D/A conversion circuits.

The number of available I/O lines is important. Without sufficient I/O lines it is necessary to use external circuitry to make up the shortfall. This is illustrated by the reaction-timer example in Figure 9.19, in which external decoder circuits are used to drive the 7-segment displays from the 4-bit BCD signals provided by the microcontroller. If the microcontroller had four parallel ports, rather than two, it would be possible to connect each seven-segment display to one port. The controlling program would then directly generate the output signals needed to drive the individual segments in the display.

### Size

Microcontroller chips come in various sizes. If an application can be handled adequately with an 8-bit microcontroller, then it makes no sense to use a 16-bit chip which is likely to be more expensive, physically larger, and consume more power. The majority of practical applications can be handled with relatively small chips. In recent years, the largest number of chips sold have been of the 8-bit type, followed by the 4-bit and 16-bit types.

The physical size is important in terms of the area that the chip occupies on a printed circuit board. This area has significant cost implications.

### Power Consumption

Power consumption is an important consideration in all computer applications. In high-performance systems the power consumption tends to be very high, requiring some mechanism to dissipate the heat generated. In many embedded applications the consumed power is low enough so that it is not necessary to worry about heat dissipation. However, these applications often involve battery powered products, so the life of the battery, determined by power consumption, is a key factor.

Power consumption is reduced if CMOS technology is used to fabricate the microcontroller chip. In CMOS technology, power consumption is proportional to clock frequency. If low performance suffices for a given application, then a lower clock frequency can be used to reduce power consumption. Another possible trade-off is with the functionality of the microcontroller chip. Reduced functionality means less circuitry on the chip, and thus lower power consumption.

### On-Chip Memory

Inclusion of memory on a microcontroller chip allows single-chip implementations of simple embedded applications. The size and the type of memory have significant ramifications. A relatively small amount of RAM may be sufficient for storing data during computations. A larger read-only memory is needed to store programs. This memory may be of ROM, PROM, EPROM, EEPROM, or Flash type, characterized by increasing costs. For high-volume products, the most economical choices are microcontrollers with ROM. However, this is also the least flexible choice because the contents of the ROM are permanently implemented at the time the chip is manufactured. The PROM and EPROM types can be programmed at the time the embedded product is made. The greatest flexibility is offered by EEPROM and Flash memories, which can be programmed multiple times.

For applications that are computationally more demanding, it is necessary to use an external memory. Some microcontrollers do not have any on-chip memory. They are typically intended for sophisticated applications where a substantial amount of memory, which cannot be realized within the microcontroller chip, is needed.

### Performance

Performance is not a big issue when microcontrollers are used in applications such as home appliances and toys, except in video games such as the Sony playstation. Small and inexpensive chips can be chosen in these cases. But, in applications such as cell phones and hand-held video games it is essential to have much higher performance. High performance demands more powerful chips, which results in higher cost and greater power consumption. Since the application is often battery powered, it is important to minimize power consumption. In Chapter 3, we discussed the ARM architecture. One implementation of this architecture is the StrongARM chip, which has been especially designed as a low-power processor that offers good performance. Thumb versions of the ARM architecture, discussed in Section 9.7.3, are intended for use in embedded applications in which both the cost and performance issues are critical.

### Software

There are many advantages to using high-level languages for computer application programs. They facilitate the process of program development and make it easy to maintain and modify the software in the future. However, there are some instances when it may be prudent to resort to assembly language. A carefully written program in assembly language is likely to generate object code that is 10 to 20 percent more compact (in terms of the amount of storage needed) than the code produced by a compiler. If an embedded application is based on a microcontroller that has on-chip memory, it is a major advantage if the necessary code can fit into the memory provided on the chip, avoiding the need for external memory.

The designer should be careful not to overestimate the capability of the available on-chip RAM. This memory is used for storage of dynamic data, as a temporary buffer, and for implementing a stack. It is easy to write code that looks compact, for example in C language, but which requires more RAM than is available.

### Instruction Set

Another significant issue is the nature of the instruction set of the processor used. CISC-like instructions lead to more compact code than RISC-like instructions. Thus, the choice of the processor has an impact on the size of the code. An interesting example is provided by the Thumb version of the ARM architecture, where a RISC-like instruction set designed for 32-bit processors has been modified into a more heavily encoded set using 16-bit instructions. Programs written for Thumb versions are up to 30 percent more compact than those written for the full ARM architecture. Recall that at execution time, the Thumb instructions are expanded into normal ARM instructions, as explained in Section 9.7.3.

### Development Tools

Designers of digital systems rely heavily on development tools. These tools include software packages for computer aided design (CAD), operating system software, compilers, assemblers, and simulators for the processors. The range and the availability of tools often depends on the choice of the embedded processor. It is also attractive to have third-party support, where alternate sources of tools and documentation exist. Good documentation and helpful advice from the manufacturer (if needed) are extremely valuable.

### Testability and Reliability

Printed circuit boards are often difficult to test, particularly if they are densely populated with chips. The testing process is greatly simplified if the entire system is designed to be easily testable. A microcontroller chip can include circuitry that makes it easier to test printed circuit boards that contain this chip. For example, some microcontrollers include a *test access port* that is compatible with the IEEE 1149.1 standard for a testable architecture, known as the Test Access Port and Boundary-Scan Architecture [1].

Embedded applications demand robustness and reliability. The life cycle of a typical product is expected to be at least five years. This is in contrast with personal computers, which tend to be considered obsolete in a shorter time.

## 9.9 SYSTEM-ON-A-CHIP

In an embedded application, it is desirable to use as few chips as possible. Ideally, a single chip would realize the entire system. In very simple applications, it is likely that some commercially available microcontrollers can realize all of the necessary functions. This is not the case in more complex applications. Some microcontroller chips are targeted for specific applications that would be difficult to implement with general-purpose microcontrollers. For example, a microcontroller intended for video games should include video and sound processing circuitry. The requirements would be quite different for microcontrollers used in a laser printer or in a cell phone.

Developing a complex microcontroller is a challenging task that takes time. Yet, the development time for most consumer products has to be short. A chip that implements an entire system for a specific application can be designed in a relatively short time if the designer makes use of some existing circuit modules that are available in an easy to use form. A microprocessor core is one of the needed modules. These cores can be obtained, through a licensing agreement, from a number of companies. Other modules that may be used to implement memory, A/D and D/A conversion circuits, or DSP (digital signal processing) circuits can also be obtained. The designer then completes the design by using the available modules and designing the rest of the required circuitry.

In Section 9.7.3, we mentioned that ARM cores have been designed to be used as modules in larger systems. Another interesting example is National Semiconductor's CompactRISC core. One of its features is the scalability from 8 to 64 bits. It has a simple 3-stage pipeline and an on-chip memory of 40K-byte ROM and 1.4K-byte RAM. A bus interface unit is added only when external memory is also needed. Thus, the complexity of the core can be adjusted to be commensurate with the functionality required by the application.

Providers of cores and other modules sell designs rather than chips. In effect, they are selling ideas rather than physical components. Their product is an example of an *intellectual property* (IP), which can be used by others to design their own chips.

## 9.9.1 FPGA IMPLEMENTATION

Field programmable gate arrays (FPGAs) provide an attractive medium for implementing systems on single chips. Unlike the microcontroller chips, which provide a designer with a set of predefined functional units, the FPGA devices allow complete freedom in the design process. They make it easy to include certain standard units and then build the rest of the system as desired. To illustrate the salient features of this approach, we will examine the Excalibur system available from Altera Corporation.

The functional capability of FPGAs has increased dramatically. A single large FPGA chip may implement a system that requires hundreds of thousands of logic gates. Such chips are large enough to implement the typical functionality of a microcontroller and other circuitry needed in a desired system.

The key component of any system on a chip is the processor core. The Excalibur system offers two distinct alternatives. One involves a processor that is defined in software. The other involves an FPGA chip that has a processor core implemented in silicon at the time of manufacture.

**Soft Processor Core**

The Excalibur system provides a software module, written in the Verilog hardware description language, which implements a processor architecture called Nios. This allows the designer to instantiate the processor as a readily available library module, which can be done using either a hardware description language, such as Verilog or

VHDL, or as a functional block using a schematic entry process. The designer can choose either a 32- or 16-bit version of the processor, depending on the performance requirements of the system.

A parallel interface module, similar to the one presented in Section 9.3.1, is available as a parameterized library module. The user can specify the parameters to suit the design requirements. The length of registers can be chosen in the range from 1 to 32 bits. The user can choose either the full bidirectional capability of the interface or a more limited version of it. For example, only the output port may be specified, in which case the output data register is provided, but the input path and the data direction register are not implemented. Thus, the FPGA resources are not wasted on unnecessary components.

A serial interface is available in the form of a UART circuit. The designer specifies the desired parameters, such as the number of data bits, the number of stop bits, and whether the parity bit should be used. The transmit/receive clock rate is selected from a predetermined standard range. This rate can later be changed by the application software, if desired, by including a divisor register that can be loaded with a value by which the clock frequency is to be divided. Again, the user is able to choose only the necessary features, and only the corresponding circuitry is implemented.

A timer module provides the counting and timing capability described in Section 9.3.3. Its operation is fully controlled by the application software.

Large FPGA chips contain a considerable amount of memory. The memory blocks can be used to implement the RAM and ROM parts of an embedded system if the memory size requirements are not too large. The designer can specify the size of the desired memory in terms of both the number of words and the number of bits per word. If the on-chip memory resources are insufficient, then an external memory interface can be instantiated, which results in the corresponding memory-bus signals being implemented on the pins of the FPGA chip.

The Excalibur CAD tools make it easy to design a system on an FPGA. They include a "wizard" which prompts the designer to enter the desired parameters and then generates the specified circuits. Thus, the processor and the I/O modules are implemented automatically. They are interconnected by a bus-like structure that implements the Nios bus protocol. We should note that a bus structure on an FPGA chip is not implemented using tri-state drivers, as discussed in Chapter 7. The FPGA is a general-purpose device that contains a large number of logic elements, interconnection wires, and switches. Tri-state drivers are useful for special purposes only; hence, they are not provided in a typical FPGA. The functionality of a tri-state bus can be implemented using multiplexer circuits. Instead of a single bidirectional path, separate multiplexers are used for each direction. While this approach requires many gates, it is a viable solution because the needed logic elements and interconnection resources are just a small fraction of the total FPGA resources.

The processor and the interface subsystem occupy a relatively small part of an FPGA chip. The rest of the chip is available for implementation of the application-specific circuitry. This circuitry can be connected either directly to the processor bus or perhaps more conveniently to the instantiated I/O ports. In the context of a system on a chip, an I/O port in the processor subsystem is not necessarily connected to an I/O pin of the FPGA device. Instead, the designer may use it to connect the application-specific circuitry implemented on the FPGA to the processor system.

The Nios processor has a RISC-like instruction set. It is capable of delivering performance up to 50 MIPS (millions of instructions per second). The designer may choose to implement more than one Nios processor on the same FPGA, thus implementing a multiprocessor system.

**Hard Processor Core**

An alternative to the soft processor core approach is to implement the processor in silicon, thus creating a specialized FPGA. The Excalibur system provides for such FPGAs based on different processors. One example is an FPGA that has an ARM processor core implemented in one part of the device. In addition to the processor circuits, the ARM processor bus, a RAM module, and a UART serial module are also implemented in silicon. This allows implementation of considerably higher-performance systems. The rest of the chip consists of the normal FPGA resources. Using a hard processor core, the system can deliver performance in the range of hundreds of MIPS.

**Designer's View**

The designer of an embedded system inevitably looks for the simplest and most cost-effective approach. A microcontroller chip that has the resources to implement an entire system may be the best choice. The situation is different if additional chips are needed to realize the system. Then, FPGA solutions become attractive because they are likely to need fewer chips to implement the system.

Another consideration is the availability of predesigned modules. A microcontroller chip contains a number of different modules, and any feature that cannot be realized using these modules has to be implemented using additional chips. An FPGA device allows the designer to design any type of digital circuit. Many practical designs involve circuits that perform commonly used tasks. Such circuits should be available as library modules. This is obviously the case with I/O interfaces and timer circuits. It is very convenient if other useful modules are available. For signal processing applications, the library should include typical filter circuits and fast multipliers. If the designed system is to be connected to another computer via a standard bus, such as PCI, the designer's task is much simpler if a PCI interface is available as a predesigned module.

## 9.10 CONCLUDING REMARKS

This chapter has provided an introduction to the design of a complete embedded computer system in the context of simple applications. We have not used a specific microcontroller because the principles presented are general. They deal with the key issues that face the designer of an embedded system.

It is particularly important to appreciate the close interaction of hardware and software. The design choices may involve trade-offs between polled I/O and interrupts, between different instruction sets in terms of functionality and compactness of code, between power consumption and performance, and so on.

The already large and still rapidly expanding world of embedded applications provides tremendous opportunities for creative use of computer technology. A number of books have been published that focus on embedded systems [2–4].

## PROBLEMS

**9.1** The microcontroller in Section 9.3 receives decimal digits (0 to 9) encoded as ASCII characters on its serial port. As each digit arrives, it has to be displayed on a 7-segment display unit connected to parallel Port A. Show the connections needed to accomplish this function. Label the segments of the display unit as indicated in Figure A.33. Write a C-language program to perform the required task. Use polling to detect the arrival of each ASCII character.

**9.2** Write an assembly language program to implement the task of Problem 9.1.

**9.3** Solve Problem 9.1 by using interrupts to detect the arrival of each ASCII character.

**9.4** Write an assembly language program for Problem 9.3.

**9.5** The microcontroller in Section 9.3 receives decimal numbers on its serial port. Each number consists of two digits encoded as ASCII characters. In order to distinguish between successive 2-digit numbers, a delimiter character H is used. Thus, if two successive numbers are 43 and 28, the received sequence will be H43H28. Each number is to be displayed on two 7-segment displays connected to parallel ports A and B. The delimiter character should not be displayed. The displayed number should change only when both digits of the next number have been received. Show the connections needed to accomplish this function. Label the segments of the display unit as indicated in Figure A.33. Write a C-language program to perform the required task. Use polling to detect the arrival of each ASCII character.

**9.6** Write an assembly language program to implement the task of Problem 9.5.

**9.7** Solve Problem 9.5 by using interrupts to detect the arrival of each ASCII character.

**9.8** Write an assembly language program for Problem 9.7.

**9.9** The microcontroller in Section 9.3 receives decimal numbers on its serial port. Each number consists of four digits encoded as ASCII characters. In order to distinguish between successive 4-digit numbers, a delimiter character H is used. Thus, if two successive numbers are 2143 and 6292, the received sequence will be H2143H6292. Each number is to be displayed on four 7-segment display units. Assume that each display unit has a BCD-to-7-segment decoder circuit associated with it, as shown in Figure P9.1.

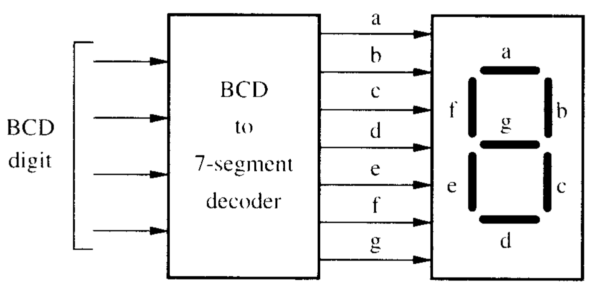

**Figure P9.1** A 7-segment display with a BCD decoder.

Show the necessary connection to the microcontroller. Write a C-language program to perform the required task. Use polling to detect the arrival of each ASCII character.

**9.10** Write an assembly language program to implement the task of Problem 9.9.

**9.11** Solve Problem 9.9 by using interrupts to detect the arrival of each ASCII character.

**9.12** Write an assembly language program to implement the task of Problem 9.11.

**9.13** Repeat Problem 9.9, but assume that each 7-segment display unit has a 7-bit register associated with it, rather than a BCD-to-7-segment decoder. The register has a control input *Load*, such that the seven data bits are loaded into the register when *Load* = 1. Each bit in the register drives one segment of the associated display unit. Figure P9.2 shows the register-display arrangement. Arrange the microcontroller output connections such that the parallel port A provides the data for all four display units.

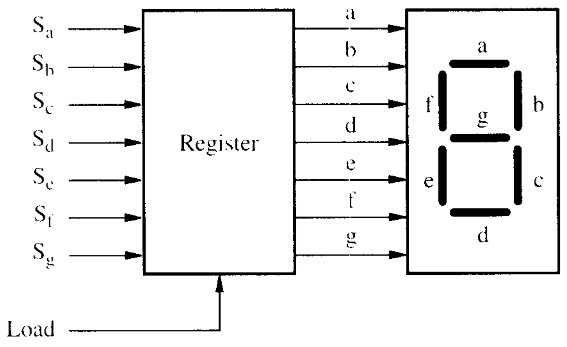

**Figure P9.2** A 7-segment display with a register.

**9.14** Repeat Problem 9.13, but write the program using assembly language.

**9.15** Solve Problem 9.13 by using interrupts to detect the arrival of each ASCII character.

**9.16** Write an assembly language program to implement the task of Problem 9.15.

**9.17** In Section 9.5 we assumed that the source device generates characters in bursts of less than 80 characters. Would the programs in Figures 9.17 and 9.18 work properly if bursts of up to 80 characters are allowed? If not, show a modification to these programs.

**9.18** In the program in Figure 9.17, the test for determining whether the circular buffer is empty is performed by checking if the *fin* and *fout* indexes have the same value. Instead, it is possible to introduce a counter variable, *M*, that indicates the current number of characters in the buffer. Modify the program using this approach.

**9.19** Repeat Problem 9.18 for the program in Figure 9.18.

**9.20** Modify the reaction timer presented in Section 9.6 assuming that the tested person will always respond in less than one second. Thus, the elapsed reaction time should be displayed as two digits representing the hundredths of a second. Connect the two 7-segment display units to Port A and modify the programs in Figures 9.20 and 9.21 to implement the desired operation.

**9.21** In Figure 9.19, the 7-segment display unit for each digit incorporates a BCD-to-7-segment decoder; hence the microcontroller provides simultaneously a 4-bit BCD code for each digit that is to be displayed. Suppose that instead of using the decoder, each 7-segment unit has a 7-bit register with a control input *Load*, such that the seven data bits are loaded into the register when *Load* = 1. Each bit in the register drives one segment of the associated display unit. Figure P9.2 shows the register-display arrangement. Augment the programs in Figures 9.20 and 9.21 for use with this register-display circuit.

**9.22** In Figure 9.21, a binary number representing the elapsed reaction time in hundredths of seconds is converted into an equivalent BCD number using successive divisions by 100 and 10. Another way of implementing this conversion is to perform successive divisions by 10, in which case the remainder obtained after each division is a desired BCD digit. What is the order of the digits produced this way? Modify the program in Figure 9.21 to perform the conversion in this way.

**9.23** Use the microcontroller in Section 9.3 to generate a "time of day" clock. The time (in hours and minutes) is to be displayed on four 7-segment display units. Assume that each display unit has a BCD-to-7-segment decoder associated with it, as shown in Figure P9.1. Assume also that a 100-MHz clock is used. Show the required hardware connections and write a suitable program.

**9.24** Repeat Problem 9.20 assuming that each 7-segment display unit has a register associated with it, as shown in Figure P9.2.

**9.25** In a system implemented on a single chip, the processor and the main memory reside on the same chip. Is there a need for a cache in this system? Explain.

## REFERENCES

1. *Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1, May 1990.
2. W. Wolf, *Computers as Components — Principles of Embedded Computing System Design*, Morgan Kaufmann Publishers, San Francisco, CA., 2001.
3. K. Hintz and D. Tabak, *Microcontrollers*, McGraw-Hill, New York, 1992.
4. J. B. Peatman, *Design with Microcontrollers*, McGraw-Hill, New York, 1988.