Second Semester

MANDALAY TECHNOLOGICAL UNIVERSITY

DEPARTMENT OF MECHATRONIC ENGINEERING

MICROPROCESSOR AND MICROCONTROLLER TECHNOLOGY II

McE 42039

Fourth Year

Motto: Creative, Innovative, Mechatronics

CHAPTER 10

PIC18 SERIAL PORT PROGRAMMING IN ASSEMBLY AND C

OBJECTIVES

Upon completion of this chapter, you will be able to:

- >> **Contrast and compare serial versus parallel communication**
- >> **List the advantages of serial communication over parallel**
- >> **Explain serial communication protocol**
- >> **Contrast synchronous versus asynchronous communication**
- >> **Contrast half- versus full-duplex transmission**
- >> **Explain the process of data framing**
- >> **Describe data transfer rate and bps rate**
- >> **Define the RS232 standard**
- >> **Explain the use of the MAX232 and MAX233 chips**
- >> **Interface the PIC18 with an RS232 connector**
- >> **Discuss the baud rate of the PIC18**
- >> **Describe serial communication features of the PIC18**
- >> **Describe the main registers used by serial communication of the PIC18**
- >> **Program the PIC18 serial port in Assembly and C**

Computers transfer data in two ways: parallel and serial. In parallel data transfers, often eight or more lines (wire conductors) are used to transfer data to a device that is only a few feet away. Devices that use parallel transfers include printers and hard disks; each uses cables with many wire strips. Although a lot of data can be transferred in a short amount of time by using many wires in parallel, the distance cannot be great. To transfer to a device located many meters away, the serial method is used. In serial communication, the data is sent one bit at a time, in contrast to parallel communication, in which the data is sent a byte or more at a time. Serial communication of the PIC18 is the topic of this chapter. The PIC18 has serial communication capability built into it, thereby making possible fast data transfer using only a few wires.

In this chapter we first discuss the basics of serial communication. In Section 10.2, PIC18 interfacing to RS232 connectors via MAX232 line drivers is discussed. Serial port programming of the PIC18 is discussed in Section 10.3. Section 10.4 covers PIC18 C programming for the serial port using the C18 compiler.

SECTION 10.1: BASICS OF SERIAL COMMUNICATION

When a microprocessor communicates with the outside world, it provides the data in byte-sized chunks. In some cases, such as printers, the information is simply grabbed from the 8-bit data bus and presented to the 8-bit data bus of the printer. This can work only if the cable is not too long, because long cables diminish and even distort signals. Furthermore, an 8-bit data path is expensive. For these reasons, serial communication is used for transferring data between two systems located at distances of hundreds of feet to millions of miles apart. Figure 10-1 diagrams serial versus parallel data transfers.

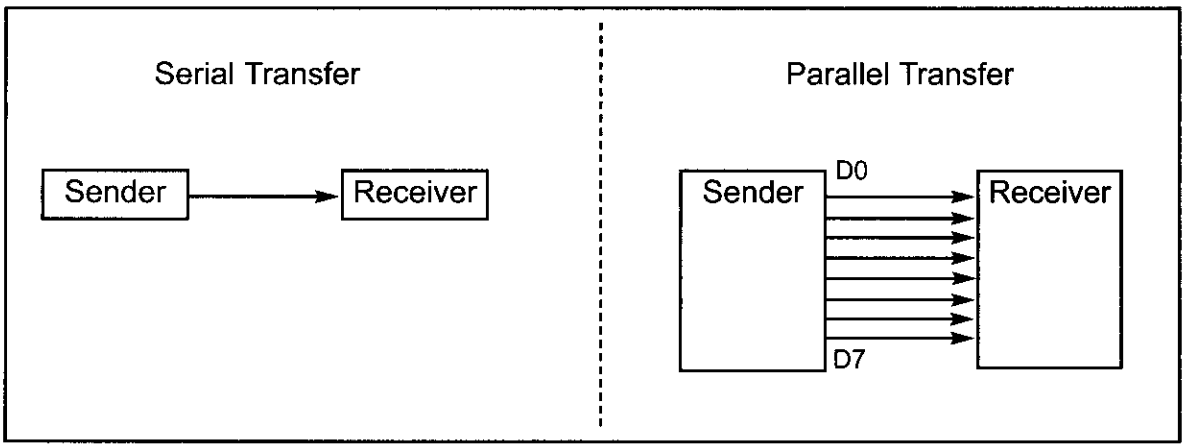


Figure 10-1. Serial versus Parallel Data Transfer

The fact that in a single data line is used in serial communication instead of the 8-bit data line of parallel communication makes serial transfer not only much cheaper but also enables two computers located in two different cities to communicate over the telephone.

For serial data communication to work, the byte of data must be converted to serial bits using a parallel-in-serial-out shift register; then it can be transmitted

over a single data line. This also means that at the receiving end there must be a serial-in-parallel-out shift register to receive the serial data and pack them into a byte. Of course, if data is to be transferred on the telephone line, it must be converted from 0s and 1s to audio tones, which are sinusoidal signals. This conversion is performed by a peripheral device called a *modem*, which stands for "modulator/demodulator."

When the distance is short, the digital signal can be transferred as it is on a simple wire and requires no modulation. This is how IBM PC keyboards transfer data to the motherboard. For long-distance data transfers using communication lines such as a telephone, however, serial data communication requires a modem to *modulate* (convert from 0s and 1s to audio tones) and *demodulate* (convert from audio tones to 0s and 1s).

Serial data communication uses two methods, asynchronous and synchronous. The *synchronous* method transfers a block of data (characters) at a time whereas the *asynchronous* method ransfers a single byte at a time. It is possible to write software to use either of these methods, but the programs can be tedious and long. For this reason, special IC chips are made by many manufacturers for serial data communications. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (universal synchronous-asynchronous receiver-transmitter). The PIC18 chip has a built-in USART, which is discussed in detail in Section 10.3.

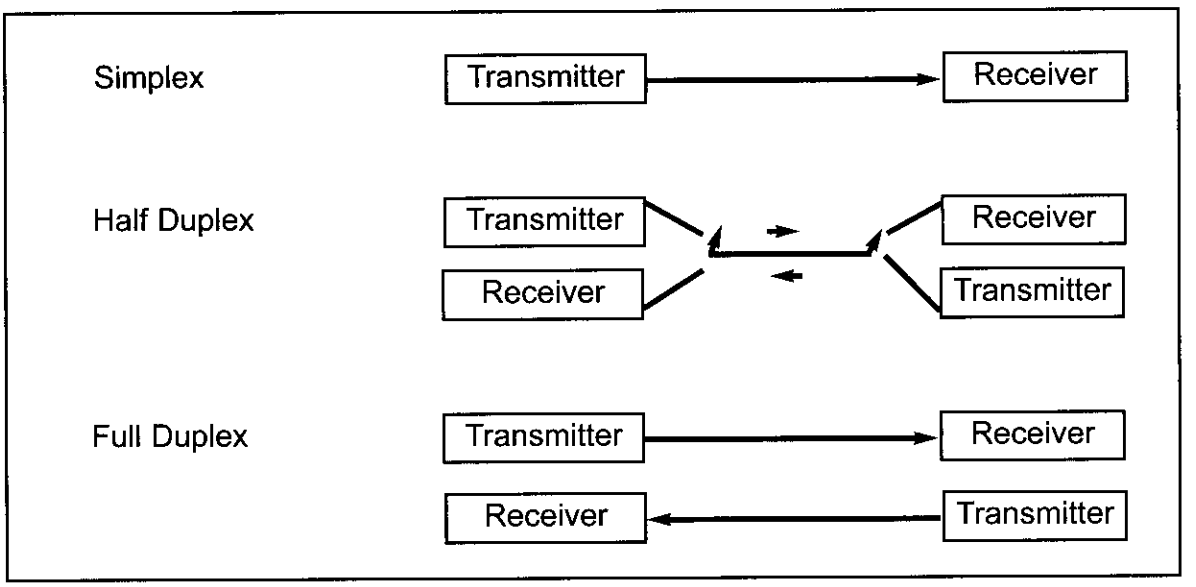


Figure 10-2. Simplex, Half-, and Full-Duplex Transfers

Half- and full-duplex transmission

In data transmission, if the data can be both transmitted and received, it is a *duplex* transmission. This is in contrast to *simplex* transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer can be simultaneous. If data is transmitted one way at a time, it is referred to as *half duplex*. If the data

can go both ways at the same time, it is *full duplex*. Of course, full duplex requires two wire conductors for the data lines (in addition to the signal ground), one for transmission and one for reception, in order to transfer and receive data simultaneously. See Figure 10-2.

Asynchronous serial communication and data framing

The data coming in at the receiving end of the data line in a serial data transfer is all 0s and 1s; it is difficult to make sense of the data unless the sender and receiver agree on a set of rules, a *protocol*, on how the data is packed, how many bits constitute a character, and when the data begins and ends.

Start and stop bits

Asynchronous serial data communication is widely used for character-oriented transmissions, while block-oriented data transfers use the synchronous method. In the asynchronous method, each character is placed between start and stop bits. This is called *framing*. In data framing for asynchronous communications, the data, such as ASCII characters, are packed between a start bit and a stop bit. The start bit is always one bit but the stop bit can be one or two bits. The start bit is always a 0 (low) and the stop bit(s) is 1 (high). For example, look at Figure 10-3 in which the ASCII character "A" (8-bit binary 0100 0001) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first.

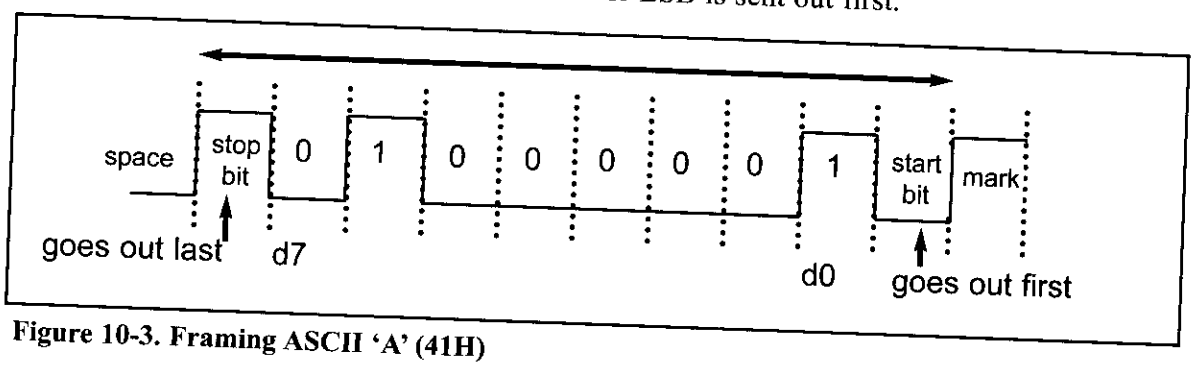


Figure 10-3. Framing ASCII 'A' (41H)

Notice in Figure 10-3 that when there is no transfer, the signal is 1 (high), which is referred to as *mark*. The 0 (low) is referred to as *space*. Notice that the transmission begins with a start bit followed by D0, the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

In asynchronous serial communications, peripheral chips and modems can be programmed for data that is 7 or 8 bits wide. This is in addition to the number of stop bits, 1 or 2. While in older systems ASCII characters were 7-bit, in recent years, 8-bit data has become common due to the extended ASCII characters. In some older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte. In modern PCs, however, the use of one stop bit is standard. Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character: 8 bits for the ASCII code, and 1 bit each for the start and stop bits. Therefore, each 8-bit character has an extra 2 bits, which gives 25% overhead.

In some systems, the parity bit of the character byte is included in the data frame in order to maintain data integrity. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit in addition to start and stop bits. The parity bit is odd or even. In the case of an odd parity bit the number of 1s in the data bits, including the parity bit, is odd. Similarly, in an even parity bit system the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even parity bit. UART chips allow programming of the parity bit for odd-, even-, and no-parity options.

Data transfer rate

The rate of data transfer in serial data communication is stated in *bps* (bits per second). Another widely used terminology for bps is *baud rate*. However, the baud and bps rates are not necessarily equal. This is because baud rate is the modem terminology and is defined as the number of signal changes per second. In modems, sometimes a single change of signal transfers several bits of data. As far as the conductor wire is concerned, the baud rate and bps are the same, and for this reason in this book we use the terms bps and baud interchangeably.

The data transfer rate of a given computer system depends on communication ports incorporated into that system. For example, the early IBM PC/XT could transfer data at the rate of 100 to 9600 bps. In recent years, however, Pentium-based PCs transfer data at rates as high as 56K. Note that in asynchronous serial data communication, the baud rate is generally limited to 100,000 bps.

RS232 standards

To allow compatibility among data communication equipment made by various manufacturers, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. In 1963 it was modified and called RS232A. RS232B and RS232C were issued in 1965 and 1969, respectively. In this book we refer to it simply as RS232. Today, RS232 is the most widely used serial I/O interfacing standard. This standard is used in PCs and numerous types of equipment. Because the standard was set long before the advent of the TTL logic family, however, its input and output voltage levels are not TTL compatible. In RS232, a 1 is represented by –3 to –25 V, while a 0 bit is +3 to +25 V, making –3 to +3 undefined. For this reason, to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic levels to the RS232 voltage level, and vice versa. MAX232 IC chips are commonly referred to as line drivers. RS232 connection to MAX232 is discussed in Section 10.2.

RS232 pins

Table 10-1 provides the pins and their labels for the RS232 cable, commonly referred to as the DB-25 connector. In labeling, DB-25P refers to the plug connector (male) and DB-25S is for the socket connector (female). See Figure 10-4.

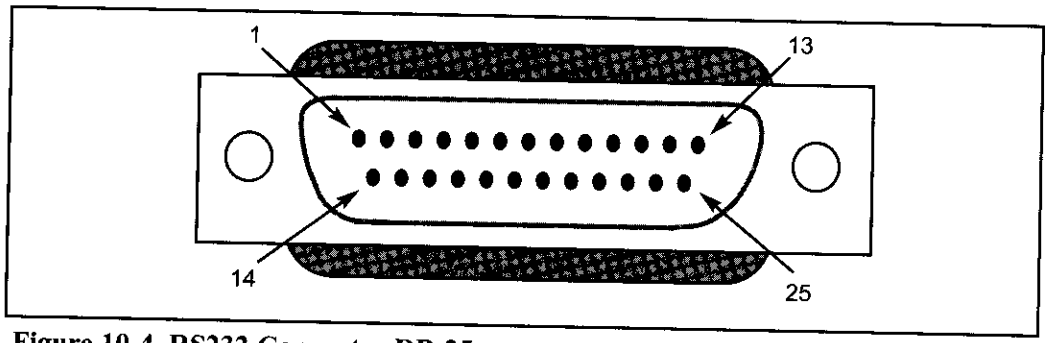


Figure 10-4. RS232 Connector DB-25

Because not all the pins are used in PC cables, IBM introduced the DB-9 version of the serial I/O standard, which uses only 9 pins, as shown in Table 10-2. The DB-9 pins are shown in Figure 10-5.

Data communication classification

Current terminology classifies data communication equipment as DTE (data terminal equipment) or DCE (data communication equipment). DTE refers to terminals and computers that send and receive data, while DCE refers to communication equipment, such as modems, that are responsible for transferring the data. Notice that all the RS232 pin function definitions of Tables 10-1 and 10-2 are from the DTE point of view.

The simplest connection between a PC and a microcontroller requires a minimum of three pins, TX, RX, and ground, as shown in Figure 10-6. Notice in that figure that the RX and TX pins are interchanged.

Table 10-1: RS232 Pins (DB-25)

Pin	Description
1	Protective ground
2	Transmitted data (TxD)
3	Received data (RxD)
4	Request to send ($\overline{\text{RTS}}$)
5	Clear to send ($\overline{\text{CTS}}$)
6	Data set ready ($\overline{\text{DSR}}$)
7	Signal ground (GND)
8	Data carrier detect ($\overline{\text{DCD}}$)
9/10	Reserved for data testing
11	Unassigned
12	Secondary data carrier detect
13	Secondary clear to send
14	Secondary transmitted data
15	Transmit signal element timing
16	Secondary received data
17	Receive signal element timing
18	Unassigned
19	Secondary request to send
20	Data terminal ready ($\overline{\text{DTR}}$)
21	Signal quality detector
22	Ring indicator
23	Data signal rate select
24	Transmit signal element timing
25	Unassigned

Examining RS232 handshaking signals

To ensure fast and reliable data transmission between two devices, the data transfer must be coordinated. Just as in the case of the printer, because the receiving device may have no room for the data in serial data communication, there must be a way to inform the sender to stop sending data. Many of the pins of the RS-232 connector are used for handshaking signals. Their description is provided below only as a reference, and they can be bypassed because they are not supported by the PIC18 UART chip.

1. DTR (data terminal ready). When the terminal (or a PC COM port) is turned on, after going through a self-test, it sends out signal DTR to indicate that it is ready for communication. If there is something wrong with the COM port, this signal will not be activated. This is an active-LOW signal and can be used to inform the modem that the computer is alive and kicking. This is an output pin from DTE (PC COM port) and an input to the modem.
2. DSR (data set ready). When the DCE (modem) is turned on and has gone through the self-test, it asserts DSR to indicate that it is ready to communicate. Thus, it is an output from the modem (DCE) and an input to the PC (DTE). This is an active-LOW signal. If for any reason the modem cannot make a connection to the telephone, this signal remains inactive, indicating to the PC (or terminal) that it cannot accept or send data.
3. RTS (request to send). When the DTE device (such as a PC) has a byte to transmit, it asserts RTS to signal the modem that it has a byte of data to transmit. RTS is an active-LOW output from the DTE and an input to the modem.
4. CTS (clear to send). In response to RTS, when the modem has room to store the data it is to receive, it sends out signal CTS to the DTE (PC) to indicate that it can receive the data now. This input signal to the DTE is used by the DTE to start transmission.
5. DCD (data carrier detect). The modem asserts signal DCD to inform the DTE (PC) that a valid carrier has been detected and that contact between it and the other modem is established. Therefore, DCD is an output from the modem and an input to the PC (DTE).
6. RI (ring indicator). An output from the modem (DCE) and an input to a PC (DTE) indicates that the telephone is ringing. RI goes on and off in synchronization with the ringing sound. Of the six handshake signals, this is the least often used because modems take care of answering the phone. If in a given system the PC is in charge of answering the phone, however, this signal can be used.

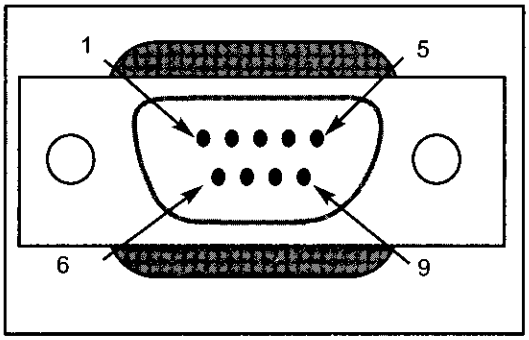


Figure 10-5. DB-9 9-Pin Connector

Table 10-2: IBM PC DB-9 Signals

Pin	Description
1	Data carrier detect (DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (DSR)
7	Request to send (RTS)
8	Clear to send (CTS)
9	Ring indicator (RI)

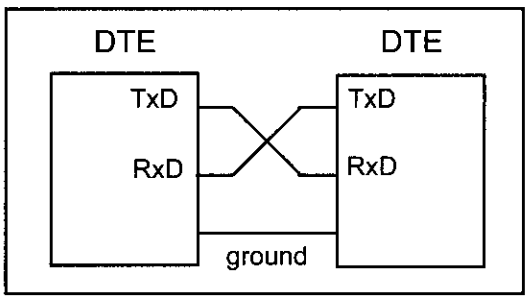


Figure 10-6. Null Modem Connection

From the above description, PC and modem communication can be summarized as follows: While signals DTR and DSR are used by the PC and modem, respectively, to indicate that they are alive and well, it is RTS and CTS that actually control the flow of data. When the PC wants to send data it asserts RTS, and in response, the modem, if it is ready (has room) to accept the data, sends back CTS. If, for lack of room, the modem does not activate CTS, the PC will deassert DTR and try again. RTS and CTS are also referred to as hardware control flow signals.

This concludes the description of the most important pins of the RS232 handshake signals plus TX, RX, and ground. Ground is also referred to as SG (signal ground).

IBM PC/compatible COM ports

IBM PC/compatible computers based on x86 (8086, 286, 386, 486, and all Pentiums) microprocessors used to have two COM ports. Both COM ports were RS232-type connectors. Many PCs used one each of the DB-25 and DB-9 RS232 connectors. The COM ports were designated as COM 1 and COM 2. In recent years, one of these has been replaced with the USB port, and COM 1 is the only serial port available, if any. We can connect the PIC18 serial port to the COM 1 port of a PC for serial communication experiments. In the absence of a COM port, we can use a COM-to-USB converter module.

With this background in serial communication, we are ready to look at the PIC18. In the next section we discuss the physical connection of the PIC18 and RS232 connector, and in Section 10.3 we see how to program the PIC18 serial communication port.

Review Questions

1. The transfer of data using parallel lines is _____ (faster, slower) but _____ (more expensive, less expensive).
2. True or false. Sending data to a printer is duplex.
3. True or false. In full duplex we must have two data lines, one for transfer and one for receive.
4. The start and stop bits are used in the _____ (synchronous, asynchronous) method.
5. Assuming that we are transmitting the ASCII letter "E" (0100 0101 in binary) with no parity bit and one stop bit, show the sequence of bits transferred serially.
6. In Question 5, find the overhead due to framing.
7. Calculate the time it takes to transfer 10,000 characters as in Question 5 if we use 9600 bps. What percentage of time is wasted due to overhead?
8. True or false. RS232 is not TTL compatible.
9. What voltage levels are used for binary 0 in RS232?
10. True or false. The PIC18 has a built-in UART.
11. On the back of x86 PCs, we normally have ____ COM port connectors.
12. The PC COM ports are designated by DOS and Windows as _____ and _____.

SECTION 10.2: PIC18 CONNECTION TO RS232

In this section, the details of the physical connections of the PIC18 to RS232 connectors are given. As stated in Section 10.1, the RS232 standard is not TTL compatible; therefore, a line driver such as the MAX232 chip is required to convert RS232 voltage levels to TTL levels, and vice versa. The interfacing of PIC18 with RS232 connectors via the MAX232 chip is the main topic of this section.

RX and TX pins in the PIC18

The PIC18 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TX and RX and are part of the PORTC group (RC6 and RC7) of the 40-pin package. Pin 25 of the PIC18 (RC7) is assigned to TX and pin 26 (RC6) is designated as RX. These pins are TTL compatible; therefore, they require a line driver to make them RS232 compatible. One such line driver is the MAX232 chip. This is discussed next.

MAX232

Because the RS232 is not compatible with today's microprocessors and microcontrollers, we need a line driver (voltage converter) to convert the RS232's signals to TTL voltage levels that will be acceptable to the PIC18's TX and RX pins. One example of such a converter is MAX232 from Maxim Corp. (www.maxim-ic.com). The MAX232 converts from RS232 voltage levels to TTL voltage levels, and vice versa. One advantage of the MAX232 chip is that it uses a +5 V power source, which is the same as the source voltage for the PIC18. In other words, with a single +5 V power supply we can power both the PIC18 and MAX232, with no need for the dual power supplies that are common in many older systems.

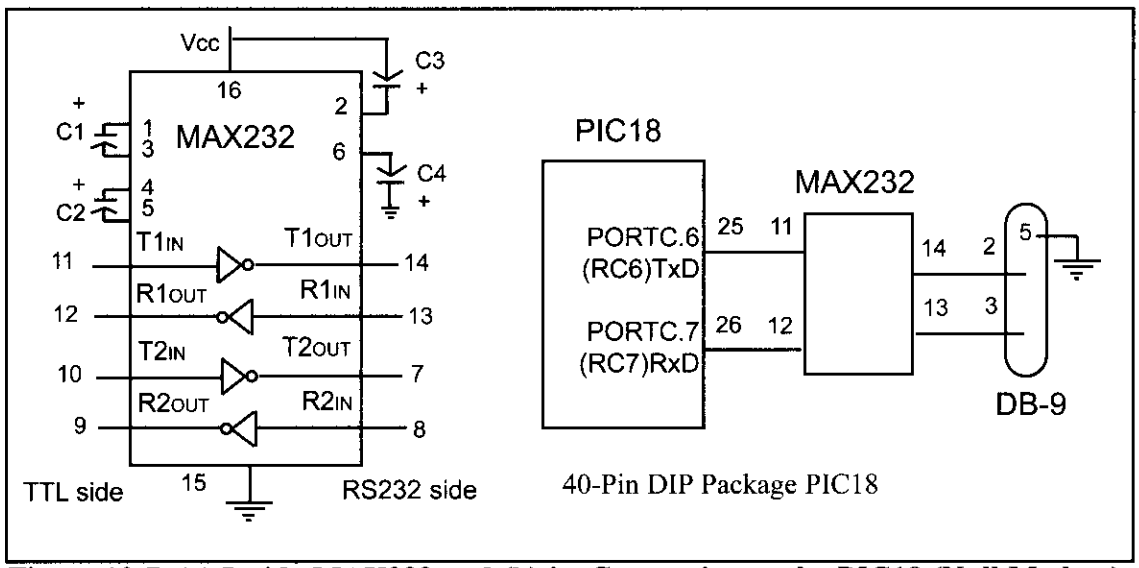


Figure 10-7. (a) Inside MAX232 and (b) its Connection to the PIC18 (Null Modem)

The MAX232 has two sets of line drivers for transferring and receiving data, as shown in Figure 10-7. The line drivers used for TX are called T1 and T2,

while the line drivers for RX are designated as R1 and R2. In many applications only one of each is used. For example, T1 and R1 are used together for TX and RX of the PIC18, and the second set is left unused. Notice in MAX232 that the T1 line driver has a designation of T1in and T1out on pin numbers 11 and 14, respectively. The T1in pin is the TTL side and is connected to TX of the microcontroller, while T1out is the RS232 side that is connected to the RX pin of the RS232 DB connector. The R1 line driver has a designation of R1in and R1out on pin numbers 13 and 12, respectively. The R1in (pin 13) is the RS232 side that is connected to the TX pin of the RS232 DB connector, and R1out (pin 12) is the TTL side that is connected to the RX pin of the microcontroller. See Figure 10-7. Notice the null modem connection where RX for one is TX for the other.

MAX232 requires four capacitors ranging from 1 to 22 μF. The most widely used value for these capacitors is 22 μF.

MAX233

To save board space, some designers use the MAX233 chip from Maxim. The MAX233 performs the same job as the MAX232 but eliminates the need for capacitors. However, the MAX233 chip is much more expensive than the MAX232. Notice that MAX233 and MAX232 are not pin compatible. You cannot take a MAX232 out of a board and replace it with a MAX233. See Figure 10-8 for MAX233 with no capacitor used.

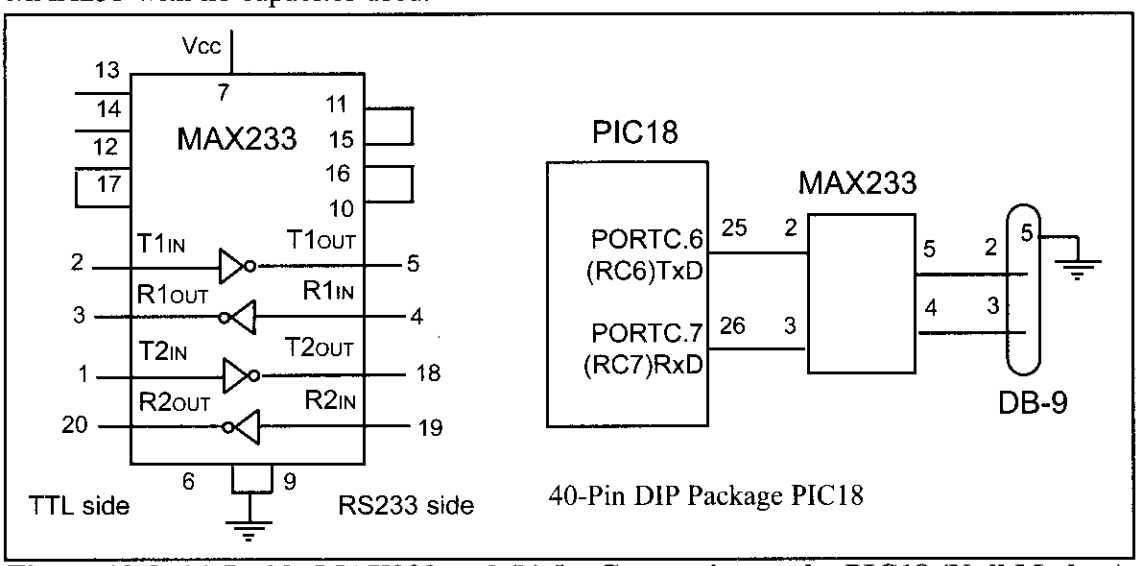


Figure 10-8. (a) Inside MAX233 and (b) Its Connection to the PIC18 (Null Modem)

Review Questions

1. True or false. The PC COM port connector is the RS232 type.
2. Which pins of the PIC18 are set aside for serial communication, and what are their functions?
3. What are line drivers such as MAX 232 used for?
4. MAX232 can support ____ lines for TX and ____ lines for RX.
5. What is the advantage of the MAX233 over the MAX232 chip?

SECTION 10.3: PIC18 SERIAL PORT PROGRAMMING IN ASSEMBLY

In this section we discuss the serial communication registers of the PIC18 and show how to program them to transfer and receive data using asynchronous mode. The USART (universal synchronous asynchronous receiver) in the PIC18 has both the synchronous and asynchronous features. The synchronous mode can be used to transfer data between the PIC and external peripherals such as ADC and EEPROMs. The asynchronous mode is the one we will use to connect the PIC18-based system to the IBM PC serial port for the purpose of full-duplex serial data transfer. In this section we examine the asynchronous mode only. In the PIC microcontroller six major registers are associated with the UART that we deal with in this chapter. They are (a) SPBGR (serial port baud rate generator), (b) TXREG (Transfer register), (c) RCREG (Receive register), (d) TXSTA (transmit status and control register), (e) RCSTA (receive status and control register), and (f) PIR1 (peripheral interrupt request register1). We examine each of them and show how they are used in full-duplex serial data communication.

SPBRG register and baud rate in the PIC18

Because IBM PC/compatible computers are so widely used to communicate with PIC18-based systems, we will emphasize serial communications of the PIC18 with the COM port of the PC. Some of the baud rates supported by PC HyperTerminal are listed in Table 10-3. You can examine these baud rates by going to the Microsoft Windows HyperTerminal program and clicking on the Communication Settings option. The PIC18 transfers and receives data serially at many different baud rates. The baud rate in the PIC18 is programmable. This is done with the help of the 8-bit register called SPBRG. For a given crystal frequency, the value loaded into the SPBRG decides the baud rate. The relation between the value loaded into SPBRG and the Fosc (frequency of oscillator connected to the OSC1 and OSC2 pins) is dictated by the following formula:

Table 10-3: Some PC Baud Rates in HyperTerminal

1,200
2,400
4,800
9,600
19,200
38,400
57,600
115,200

Desired Baud Rate = Fosc/(64X + 64) = Fosc/64(X + 1)

where X is the value we load into the SPBGR register. Assuming that Fosc = 10 MHz, we have the following:

Desired Baud Rate = Fosc/64(X + 1) = 10 MHz/64(X + 1) = 6250 Hz/(X + 1)

To get the X value for different baud rates we can solve the equation as follows:

X = (156250/Desired Baud Rate) – 1

Table 10-4 shows the X values for the different baud rates if Fosc = 10 MHz. Another way to understand the SPBRG values in Table 10-4 is to look at

them from the perspective of the instruction cycle time. As we discussed in previous chapters, the PIC18 divides the crystal frequency (Fosc) by 4 to get the instruction cycle time frequency. In the case of XTAL = 10 MHz, the instruction cycle frequency is 2.5 MHz. The PIC18's UART circuitry divides the instruction cycle frequency by 16 once more before it is used by an internal timer to set the baud rate. Therefore, 2.5 MHz divided by 16 gives 156,250 Hz. This is the number we use to find the SPBRG value shown in Table 10-4. Example 10-1 shows how to verify the data in Table 10-4. Table 10-5 shows the SPBRG values with the crystal frequency of 4 MHz (Fosc = 4 MHz).

Example 10-1

With Fosc = 10 MHz, find the BGRP value needed to have the following baud rates:
(a) 9600 (b) 4800 (c) 2400 (d) 1200

Solution:

Because Fosc = 10 MHz, we have 10 MHz/4 = 2.5 MHz for the instruction cycle frequency. This is divided by 16 once more before it is used by UART. Therefore, we have 2.5 MHz/16= 156250 Hz and X = (156250 Hz/Desired Baud Rate) – 1:

(a) (156250/ 9600) – 1 = 16.27 – 1 = 15.27 = 15 = F (hex) is loaded into SPBRG
(b) (156250/ 4800) – 1 = 32.55 – 1 = 31.55 = 32 = 20 (hex) is loaded into SPBRG
(c) (156250/ 2400) – 1 = 65.1 – 1 = 64.1 = 64 = 40 (hex) is loaded into SPBRG
(d) (156250/ 1200) – 1 = 130.2 – 1 = 129.2 = 129 = 81 (hex) is loaded into SPBRG

Notice that dividing the instruction cycle frequency by 16 is the setting upon Reset. We can get a higher baud rate with the same crystal by changing this default setting. This is done by making bit BRGH = 1 in the TXSTA register. This is explained at the end of this section.

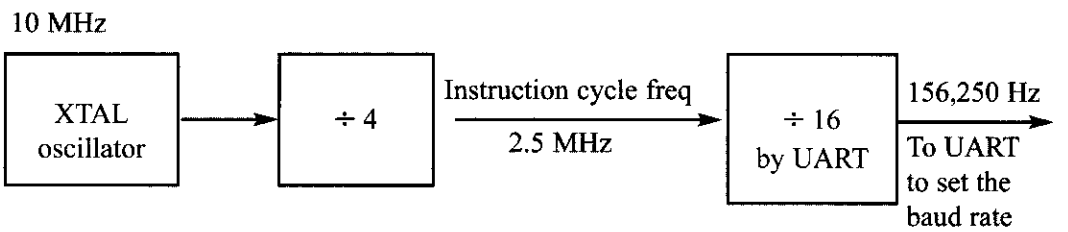


Table 10-4: SPBRG Values for Various Baud Rates (Fosc = 10 MHz, BRGH = 0)

Baud Rate	SPBRG (Decimal Value)	SPBRG (Hex Value)
38400	3	3
19200	7	7
9600	15	F
4800	32	20
2400	64	40
1200	129	81

Note: For Fosc = 10 MHz we have SPBRG = (156,250/BaudRate) – 1

Table 10-5: SPBRG Values for Various Baud Rates (Fosc = 4 MHz, BRGH = 0)

Baud Rate	SPBRG (Decimal Value)	SPBRG (Hex Value)
19200	2	2
9600	5	5
4800	12	0C
2400	25	19
1200	51	33

Note: For Fosc = 4 MHz we have 4 MHz/4 = 1 MHz for instruction cycle freq. The frequency used by the UART is 1 MHz/16 = 62,500 Hz. That means SPBRG = (62500/Baud Rate) – 1

TXREG register

TXREG is another 8-bit register used for serial communication in the PIC18. For a byte of data to be transferred via the TX pin, it must be placed in the TXREG register. TXREG is a special function register (SFR) and can be accessed like any other register in the PIC18. Look at the following examples of how this register is accessed:

```
MOVLW 0x41      ;WREG=41H, ASCII for letter 'A'
MOVWF TXREG     ;copy WREG into TXREG

MOVFF PORTB,TXREG ;copy PORTB contents into TXREG
```

The moment a byte is written into TXREG, it is fetched into a register called TSR (transmit shift register). The TSR frames the 8-bit data with the start and stop bits and the 10-bit data is transferred serially via the TX pin. Notice that while TXREG is accessible by the programmer, TSR is not accessible and is strictly for internal use.

RCREG register

Similarly, when the bits are received serially via the RX pin, the PIC18 deframes them by eliminating the stop and start bits, making a byte out of the data received, and then placing it in the RCREG register. The following code will dump the received byte into PORTB:

```
MOVFF RCREG,PORTB ;copy RXREG to PORTB
```

TXSTA (transmit status and control register)

The TXSTA register is an 8-bit register used to select the synchronous/asynchronous modes and data framing size, among other things. Figure 10-9 describes various bits of the TXSTA register. In this textbook we use the asynchronous mode with a data size of 8 bits. The BRGH bit is used to select a higher speed for transmission. The default is lower baud rate transmission. We will examine the higher transmission rate at the end of this chapter. Notice that D6 of the TXSTA register determines the framing of data by specifying the number of bits per character. We use an 8-bit data size. There are some applications for the 9-bit in which the ninth bit can be used as an address.

CSRC	TX9	TXEN	SYNC	0	BRGH	TRMT	TX9D

CSRC D7 Clock Source Select (not used in asynchronous mode, therefore D7 = 0.)

TX9 D6 9-bit Transmit Enable
1 = Select 9-bit transmission
0 = Select 8-bit transmission (We use this option, therefore D6 = 0.)

TXEN D5 Transmit Enable
1 = Transmit Enabled
0 = Transmit Disabled
We turn "on" and "off" this bit in order to start or stop data transfer.

SYNC D4 USART mode Select (We use asynchronous mode, therefore D4 = 0.)
1 = Synchronous
0 = Asynchronous

0 D3

BRGH D2 High Baud Rate Select
0 = Low Speed (Default)
1 = High Speed
We can double the baud rate with the same Fosc. See the end of this section for further discussion on this bit.

TRMT D1 Transmit Shift Register (TSR) Status
1 = TSR empty
0 = TSR full

The importance of the TSR register. To transfer a byte of data serially, we write it into TXREG. The TSR (transmit shift register) is an internal register whose job is to get the data from the TXREG, frame it with the start and stop bits, and send it out one bit at a time via the TX pin. When the last bit, which is the stop bit, is transmitted, the TRMT flag is raised to indicate that it is empty and ready for the next byte. When TSR fetches the data from TXREG, it clears the TRMT flag to indicate it is full. Notice that TSR is a parallel-in-serial-out shift register and is not accessible to the programmer. We can only write to TXREG. Whenever the TSR is empty, it gets its data from TXREG and clears the TXREG register immediately, so it does not send out the same data twice.

TXD9 D0 9th bit of Transmit Data (Because we use the 8-bit option, we make D0 = 0)
Can be used as an address/data or a parity bit in some applications

Figure 10-9. TXSTA: Transmit Status and Control Register

RCSTA (receive status and control register)

The RCSTA register is an 8-bit register used to enable the serial port to receive data, among other things. Figure 10-10 describes various bits of the RCSTA register. In this section we use the 8-bit data frame.

SPEN	RX9	SREN	CREN	ADDE	FERR	OERR	RX9D

SPEN D7 Serial port enable bit
1 = Serial port enabled, which makes TX and RX pins as serial port pins
0 = Serial port disabled

RX9 D6 9-bit Receive enable bit
1 = Select 9-bit reception
0 = Select 8-bit reception (We use this option; therefore, D6 = 0.)

SREN D5 Single receive enable bit (not used in asynchronous mode D5 = 0)

CREN D4 Continuous receive enable bit
1 = Enable continuous Receive (in asynchronous mode)
0 = Disable continuous Receive (in asynchronous mode)

ADDEN D3 Address delete enable bit (Because used with the 9-bit data frame D3 = 0)

FERR D2 Framing error bit
1 = Framing error
0 = No Framing error

OERR D1 Overrun error bit
1 = Overrun error
0 = No overrun error

TXD9 D0 9th bit of Receive data (Because we use the 8-bit option, we make D0 = 0)
Can be used as an address/data or a parity bit in some applications.

Figure 10-10. RCSTA: Receive Status and Control Register

---	--	RCIF	TXIF	--	---	--	

RCIF Receive interrupt flag bit
1 = The UART has received a byte of data and it is sitting in the RCREG register (receive buffer), waiting to be picked up.
Upon reading the RCREG register, the RCIF is cleared to allow the next byte to be received.
0 = The RCREG is empty.

TXIF Transmit interrupt flag bit
0 = The TXREG register is full.
1 = The TXREG (transmit buffer) register is empty.

The importance of TXIF: To transmit a byte of data, we write it into TXREG. Upon writing a byte into TXREG, the TXIF flag is cleared. When the entire byte is transmitted via the TX pin, the TXIF flag bit is raised to indicate that it is ready for the next byte. So, we must monitor this flag before we write a new byte into TXREG, otherwise, we wipe out the last byte before it is transmitted.

Several bits of this register are used by the timer flag, as we saw in Chapter 9. The location of the flag bits in the PIR1 register is not fixed and can vary in future PIC18 products.

Figure 10-11. PIR1 (Peripheral Interrupt Register 1)

PIR1 (peripheral interrupt request register 1)

In Chapter 9, we saw how some of the bits of PIR1 are used by the timers. Two of the PIR1 register bits are used by the UART. They are TXIF (transmit interrupt flag) and RCIF (receive interrupt flag). See Figure 10-11. We monitor (poll) the TXIF flag bit to make sure that all the bits of the last byte are transmitted before we write another byte into the TXREG. By the same logic, we monitor the RCIF flag to see if a byte of data has come in yet. In Chapter 11 we will see how these flags are used with interrupts instead of polling. Next we will examine how TXIF flags are used in serial data transfer.

Programming the PIC18 to transfer data serially

In programming the PIC18 to transfer character bytes serially, the following steps must be taken:

1. The TXSTA register is loaded with the value 20H, indicating asynchronous mode with 8-bit data frame, low baud rate, and transmit enabled.
2. Make TX pin of PORTC (RC6) an output for data to come out of the PIC.
3. The SPBRG is loaded with one of the values in Table 10-4 (or Table 10-5 if Fosc = 4 MHz) to set the baud rate for serial data transfer.
4. SPEN bit in the RCSTA register is set HIGH to enable the serial port of the PIC18.
5. The character byte to be transmitted serially is written into the TXREG register.
6. Monitor the TXIF bit of the PIR1 register to make sure UART is ready for next byte.
7. To transfer the next character, go to Step 5.

Example 10-2 shows the program to transfer data serially at 9600 baud. Example 10-3 shows how to transfer "YES" continuously.

Example 10-2

Write a program for the PIC18 to transfer the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 10 MHz.

Solution:

```
      MOVLW B'00100000' ;enable transmit and choose low baud rate
      MOVWF TXSTA       ;write to reg
      MOVLW D'15'       ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG       ;write to reg
      BCF TRISC, TX     ;make TX pin of PORTC an output pin
      BSF RCSTA, SPEN   ;enable the entire serial port of PIC18
OVER  MOVLW A'G'        ;ASCII letter 'G' to be transferred
S1    BTFSS PIR1, TXIF  ;wait until the last bit is gone
      BRA S1            ;stay in loop
      MOVWF TXREG       ;load the value to be transferred
      BRA OVER          ;keep sending letter 'G'
```

Example 10-3

Write a program to transmit the message "YES" serially at 9600 baud, 8-bit data, and 1 stop bit. Do this forever.

Solution:

```
      MOVLW B'00100000'       ;enable transmit and choose low baud
      MOVWF TXSTA             ;write to reg
      MOVLW D'15'             ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG             ;write to reg
      BCF TRISC, TX           ;make TX pin of PORTC an output pin
      BSF RCSTA, SPEN         ;enable the serial port
OVER  MOVLW A'Y'              ;ASCII letter 'Y' to be transferred
      CALL  TRANS
      MOVLW A'E'              ;ASCII letter 'E' to be transferred
      CALL  TRANS
      MOVLW A'S'              ;ASCII letter 'S' to be transferred
      CALL  TRANS
      MOVLW 0x0               ;NULL to purge the buffer
      CALL TRANS
      BRA   OVER              ;keep doing it
TRANS ;----serial data transfer subroutine
S1    BTFSS PIR1, TXIF        ;wait until the last bit is gone
      BRA S1                  ;stay in loop
      MOVWF TXREG             ;load the value to be transmitted
      RETURN                  ;return to caller
```

Importance of the TXIF flag

To understand the importance of the role of TXIF, look at the following sequence of steps that the PIC18 goes through in transmitting a character via TX:

1. The byte character to be transmitted is written into the TXREG register.
2. The TXIF flag is set to 1 internally to indicate that TXREG has a byte and will not accept another byte until this one is transmitted.
3. The TSR (Transmit Shift Register) reads the byte from TXREG and begins to transfer the byte starting with the start bit.
4. The TXIF is cleared to indicate that the last byte is being transmitted and TXREG is ready to accept another byte.
5. The 8-bit character is transferred one bit at a time.
6. By monitoring the TXIF flag, we make sure that we are not overloading the TXREG register. If we write another byte into the TXREG register before the TSR has fetched the last one, the old byte could be lost before it is transmitted.

From the above discussion we conclude that by checking the TXIF flag bit, we know whether or not the PIC18 is ready to transfer another byte. The TXIF flag bit can be checked by the instruction "`BTFSS PIR1,TXIF`" or we can use an interrupt, as we will see in Chapter 11. In Chapter 11 we will show how to use interrupts to transfer data serially, and avoid tying down the microcontroller with instructions such as "`BTFSS PIR1, TXIF`".

Programming the PIC18 to receive data serially

In programming the PIC18 to receive character bytes serially, the following steps must be taken:

1. The RCSTA register is loaded with the value 90H, to enable the continuous receive in addition to the 8-bit data size option.
2. The TXSTA register is loaded with the value 00H to choose the low baud rate option.
3. SPBRG is loaded with one of the values in Table 10-4 to set the baud rate (assuming XTAL = 10 MHz).
4. Make the RX pin of PORTC (RC7) an input for data to come into the PIC18.
5. The RCIF flag bit of the PIR1 register is monitored for a HIGH to see if an entire character has been received yet.
6. When RCIF is raised, the RCREG register has the byte. Its contents are moved into a safe place.
7. To receive the next character, go to Step 5.

Example 10-4 shows the coding of the above steps.

Example 10-4

Program the PIC18 to receive bytes of data serially and put them on PORTB. Set the baud rate at 9600, 8-bit data, and 1 stop bit.

Solution:

```
      MOVLW B'10010000'       ;enable receive and serial port itself
      MOVWF RCSTA             ;write to reg
      MOVLW D'15'             ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG             ;write to reg
      BSF   TRISC, RX         ;make RX pin of PORTC an input pin
      CLRF  TRISB             ;make port B an output port
;get a byte from serial port and place it on PORTB
R1    BTFSS PIR1, RCIF        ;check for ready
      BRA   R1                ;stay in loop
      MOVFF RCREG, PORTB      ;save value into PORTB
      BRA   R1                ;keep doing that
```

Importance of the RCIF flag bit

In receiving bits via its RX pin, the PIC18 goes through the following steps:

1. It receives the start bit indicating that the next bit is the first bit of the character byte it is about to receive.
2. The 8-bit character is received one bit at time. When the last bit is received, a byte is formed and placed in RCREG.
3. The stop bit is received. It is during receiving the stop bit that the PIC18 makes RCIF = 1, indicating that an entire character byte has been received and must

be picked up before it gets overwritten by another incoming character.
4. By checking the RCIF flag bit when it is raised, we know that a character has been received and is sitting in the RCREG register. We copy the RCREG contents to a safe place in some other register or memory before it is lost.
5. After the RCREG contents are read (copied) into a safe place, the RCIF flag bit is forced to 0 by the UART itself. This allows the next received character byte to be placed in RCREG, and also prevents the same byte from being picked up multiple times.

From the above discussion we conclude that by checking the RCIFI flag bit we know whether or not the PIC18 has received a character byte. If we fail to copy RCREG into a safe place, we risk the loss of the received byte. More importantly, note that the RCIF flag bit is raised by the PIC18, and it is also cleared by the CPU when the data in the RCREG is picked up. Note also that if we copy RCREG into a safe place before the RCIF flag bit is raised, we risk copying garbage. The RCIF flag bit can be checked by the instruction "`BTFSS PIR1, RCIF`" or by using an interrupt, as we will see in Chapter 11.

Quadrupling the baud rate in the PIC18

There are two ways to increase the baud rate of data transfer in the PIC18:

1. Use a higher-frequency crystal.
2. Change a bit in the TXSTA register, as shown below.

Option 1 is not feasible in many situations because the system crystal is fixed. Therefore, we will explore option 2. There is a software way to quadruple the baud rate of the PIC18 while the crystal frequency stays the same. This is done with the BRGH bit of the TXSTA register. When the PIC18 is powered up, D2 (BRGH bit) of the TXSTA register is zero. We can set it to high by software and thereby quadruple the baud rate.

To see how the baud rate is quadrupled with this method, we show the role of the BRGH bit (D2 bit of the TXSTA register), which can be 0 or 1. We discuss each case.

Baud rates for BRGH = 0

When BRGH = 0, the PIC18 divides Fosc/4 (crystal frequency) by 16 once more and uses that frequency for UART to set the baud rate. In the case of XTAL = 10 MHz we have:

```
Instruction cycle freq. = 10 MHz / 4 = 2.5 kHz
and
2.5 MHz / 16 = 156,250 Hz because BRGH = 0
```

This is the frequency used by UART to set the baud rate. This has been the basis of all the examples so far because it is the default when the PIC18 is powered up. The baud rate for BRGH = 0 was listed in Table 10-4 and Table 10-5.

Baud rates for BRGH = 1

With the fixed crystal frequency, we can quadruple the baud rate by making BRGH = 1. When the BRGH bit (D2 of the TXSTA register) is set to 1, Fosc/4 of XTAL is divided by 4 (instead of 16) once more, and that is the frequency used by UART to set the baud rate. In the case of XTAL = 10 MHz, we have:

```
Instruction cycle freq. = 10 MHz / 4 = 2.5 MHz
and
2.5 MHz / 4 = 625000 Hz because BHRG = 1
```

This is the frequency used by UART to set the baud rate if BHRH = 1.

Table 10-8 shows that the values loaded into SPBREG are the same for both cases; however, the baud rates are quadrupled when BRGH = 1. Look at Examples 10-5 through 10-7 to clarify the data given in Tables 10-6 and 10-7.

Table 10-6: SPBRG Values for Various Baud Rates (Fosc = 10MHz and BRGH = 1)

Baud Rate	SPBRG (Decimal Value)	SPBRG (Hex Value)
57600	10	0A
38400	15	0F
19200	32	20
9600	64	40
4800	129	81

Note: For Fosc = 10 MHz we have SPBRG = (625000/Baud Rate) – 1

Example 10-5

Find the SPBRG value (in both decimal and hex) to set the baud rate to each of the following:
(a) 9600 if BRGH = 1 (b) 4800 if BRGH = 1
Assume that XTAL = 10 MHz.

Solution:
With XTAL = 10 MHz, Fosc/4 = 2.5 MHz. Because BRGH = 1, we have UART frequency = 2.5 MHz/4 = 625,000 Hz.
(a) (625,500 / 9600) – 1 = 64; therefore, SPBRG = 64 or SPBRG = 40H (in hex).
(b) (625,500 / 4800) – 1 = 129; therefore, SPBRG = 129 or SPBRG = 81H (in hex).

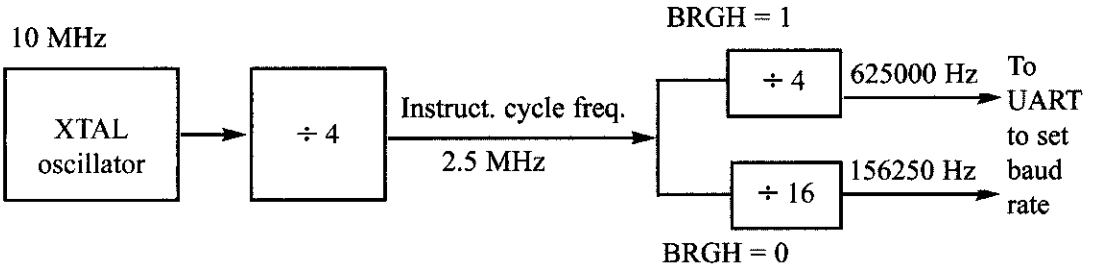


Table 10-7: SPBRG Values for Various Baud Rates (XTAL = 10 MHz)

	BRGH = 0	BRGH = 1
Baud Rate	**SPBRG (Decimal)**	**SPBRG (Decimal)**
57600	2	10
38400	3	15
19200	7	32
9,600	15	64
4,800	32	129
SPBRG = (156250/Baud rate) – 1		SPBRG = (625000/Baud rate) – 1

Table 10-8: SPBRG Values vs. Baud Rates for BRGH = 0 and BRGH = 1 (XTAL = 10 MHz)

	BRGH = 0	BRGH = 1
SPBRG (Decimal)	**Baud Rate**	**Baud Rate**
15	9600	38400
32	4800	19200
64	2400	9600

Table 10-9: SPBRG Values for Various Baud Rates (XTAL = 4 MHz)

	BRGH = 0	BRGH = 1
Baud Rate	**SPBRG (Decimal)**	**SPBRG (Decimal)**
19200	3	12
9,600	6	25
4,800	12	51
2,400	25	103
SPBRG = (62500/Baud rate) – 1		SPBRG = (250000/Baud rate) – 1

Example 10-6

Write a program for the PIC18 to transfer the letter 'G' serially at 57600 baud, continuously. Assume XTAL = 10 MHz. Use the BRGH = 1 mode

Solution:

```
        MOVLW B'00100100' ;enable transmit and choose high baud rate
        MOVWF TXSTA       ;write to reg
        MOVLW D'10'       ;57600 bps (Fosc / (16 * Speed) - 1)
        MOVWF SPBRG       ;write to reg
        BCF TRISC, TX     ;make TX pin of PORTC an output pin
        BSF RCSTA, SPEN   ;enable the entire Serial port of PIC18
OVER    MOVLW A'G'        ;ASCII letter 'G' to be transferred
S1      BTFSS PIR1, TXIF  ;wait until the last bit is gone
        BRA S1            ;stay in loop
        MOVWF TXREG       ;load the value to be transferred
        BRA OVER          ;keep sending letter 'G'
```

Baud rate error calculation

In calculating the baud rate we have used the integer number for the SPBRG register values because PIC microcontrollers can only use integer values. By dropping the decimal portion of the calculated values we run the risk of introducing error into the baud rate. There are several ways to calculate this error. One way would be to use the following formula.

Error = (Calculated value for the SPBRG – Integer part)/Integer part

For example, with the XTAL = 10 MHz and BRGH = 0 we have the following for the 9600 baud rate:

SPBRG value = (156250/9600) – 1 = 16.27 – 1 = 15.27 = 15
and the error is

(15.27 – 15)/16 = 1.7%

Another way to calculate the the error rate is as follows:

Error = (calculated baud rate – desired baud rate) / desired baud rate

Example 10-7

Assuming XTAL = 10 MHz, calculate the baud rate error for the following:
(a) 2400 (b) 1200 (c) 19200 (d) 57600
Use the BRGH = 0 mode.

Solution:

(a) SPBRG Value = (156250/2400) – 1 = 65.1 – 1 = 64.1 = 64

Error = (64.1 – 64)/ 65 = 0.15%

(b) SPBRG Value (156250/1200) – 1 = 130.2 – 1 = 129.2 = 129

Error = (129.2 – 129)/130 = 0.15%

(c) SPBRG Value (156250/19200) – 1 = 8.138 – 1 = 7.138 = 7

Error = (7.138 – 7)/8 = 1.7%

(d) SPBRG Value (156250/57600) – 1 = 2.71 – 1 = 1.71 = 1

Error = (1.71 – 1)/2 = 35%

Such an error rate is too high. Let's round up the number to see what happens.
Error = (3 – 2.7)/3 = 10% This means we use SPBRG = 2 instead of SPBRG = 1.

where the desired baud rate is calculated using X = ((Fosc/Desired Baud rate)64) – 1 and then the integer X (value loaded into SPBRG reg) is used for the calculated baud rate as follows:

calculated baud rate = Fosc/(64(X + 1)) (for BRGH = 0)

For XTAL = 10 MHz and 9600 baud rate, we got X = 15. Therefore, we get the calculated baud rate of 10 MHz/(64(15 + 1)) = 9765. Now the error is calculated as follows:

Error = (9765 – 9600)/9600= 1.7%

which is the same as what we got earlier using the other method. Tables 10-10 and 10-11 provide the error rates for SPBRG values of 10 MHz and 4 MHz crystal frequencies, respectively. Compare Examples 10-7 and 10-8 to see how to calculate the error rates two different ways.

Example 10-8

Assuming XTAL = 10 MHz, calculate the baud rate error for the following:
(a) 2400 (b) 1200
Assume BRGH = 0
Solution:

(a) SPBRG Value = (156250/2400) – 1 = 65.1 – 1 = 64.1 = 64
and calculated baud rate is 156250/(64 + 1) = 2403
Error = (2403 – 2400)/2400 = 0.12%

(b) SPBRG Value (156250/1200) – 1 = 130.2 – 1 = 129.2 = 129
where the calculated baud rate is 156250/(129 + 1) = 1202
Error = (1202 – 1200)/1200 = 0.16%

Table 10-10: SPBRG Values for Various Baud Rates (XTAL = 10 MHz)

	BRGH = 0		BRGH = 1	
Baud Rate	**SPBRG**	**Error**	**SPBRG**	**Error**
38400	3	1.5%	15	1.7%
19200	7	1.7%	32	1.3%
9,600	15	1.7%	64	0.15%
4,800	32	1.3%	129	0.15%
SPBRG = (156250/Baud rate) – 1			SPBRG = (625000/Baud rate) – 1	

Table 10-11: SPBRG Values for Various Baud Rates (XTAL = 4 MHz)

	BRGH = 0		BRGH = 1	
Baud Rate	**SPBRG**	**Error**	**SPBRG**	**Error**
19200	2	8.3%	12	0.15%
9,600	6	8%	25	0.15%
4,800	12	0.15%	51	0.15%
2,400	25	0.16%	103	0.16%
SPBRG = (62500/Baud rate) – 1			SPBRG = (250000/Baud rate) – 1	

Examine the next few examples to master the topic of PIC18 serial port programming.

Example 10-9

Assume a switch is connected to pin RD7. Write a program to monitor its status and send two messages to the serial port continuously as follows:
SW = 0 send "NO"
SW = 1 send "YES"
Assume XTAL = 10 MHz, and set the baud rate to 9,600.

Solution:

```
       BSF TRISD,7          ;PORTD.7 as in input for SW
       MOVLW 0x20           ;enable transmit and choose low baud rate
       MOVWF TXSTA          ;write to reg
       MOVLW D'15'          ;9600 bps (Fosc / (64 * Speed) - 1)
       MOVWF SPBRG          ;write to reg
       BCF   TRISC, TX      ;make TX pin of PORTC an output pin
       BSF   RCSTA, SPEN    ;enable the entire serial port of PIC18
OVER   BTFSS PORTD,7
       BRA   NEXT
       MOVLW high(MESS1)    ;if SW = 0 display "NO"
       MOVWF TBLPTRH
       MOVLW low(MESS1)
       MOVWF TBLPTRL
FN     TBLRD*+              ;read the character
       MOVF  TABLAT,W
       BZ    EXIT           ;check for end of line
       CALL  SENDCOM        ;send character to serial port
       BRA   FN             ;repeat
NEXT   MOVLW high(MESS2)    ;if SW = 1 display "YES"
       MOVWF TBLPTRH
       MOVLW low(MESS2)
       MOVWF TBLPTRL
LN     TBLRD*+              ;read the character
       MOVF  TABLAT,W       ;Z = 1 if NULL
       BZ    EXIT           ;check for end of line
       CALL  SENDCOM        ;send character to serial port
       BRA   LN             ;repeat
EXIT   MOVLW 0x20           ;send space
       CALL  SENDCOM
       GOTO  OVER
;-------------
SENDCOM
S1     BTFSS PIR1, TXIF     ;wait until the last bit is gone
       BRA   S1             ;stay in loop
       MOVWF TXREG          ;load the value to be transferred
      RETURN                ;return to caller
;------------------
MESS1 DB     "NO",0
MESS2 DB     "YES",0
```

Example 10-10

Write a program to send the message "The Earth is but One Country" to the serial port continuously. Assume a SW is connected to pin RB0. Monitor its status and set the baud rate as follows:
SW = 0 9600 baud rate
SW = 1 38400 baud rate
Assume XTAL = 10 MHz.

Solution:

As shown in Table 10-8, we can quadruple the baud rate by changing the BRGH bit of the TXSTA register.

```
      BSF   TRISB,0      ;PORTB.0 as in input for SW
      BCF   TRISC, TX    ;make TX pin of PORTC an output pin
      BSF   RCSTA, SPEN ;enable the entire serial port of PIC18
      MOVLW 0x20         ;transmit at low baud rate
      MOVWF TXSTA        ;write to reg
      MOVLW D'15'        ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG        ;write to reg
OVER  BTFSC PORTB,0      ;test bit PORTB.0 and skip if LOW
      BSF   TXSTA,BRGH   ;transmit at high rate  by making BRGH = 1
      MOVLW upper(MESSAGE)
      MOVWF TBLPTRU
      MOVLW high(MESSAGE)
      MOVWF TBLPTRH
      MOVLW low(MESSAGE)
      MOVWF TBLPTRL
NEXT  TBLRD*+            ;read the character
      MOVF  TABLAT,W     ;place it in WREG
      BZ    OVER         ;if end of line, start over
      CALL  SENDCOM      ;send char to serial port
      BRA   NEXT         ;repeat for the next character
;-------------
SENDCOM
S1    BTFSS PIR1, TXIF   ;wait until the last bit is gone
      BRA   S1           ;stay in loop
      MOVWF TXREG        ;load the value to be transmitted
      RETURN             ;return to caller
;------------------
MESSAGE DB  "The Earth is but One Country",0
```

Transmit and receive

Assume that the PIC18 serial port is connected to the COM port of the IBM PC, and we are using the HyperTerminal program on the PC to send and receive data serially. The ports PORTB and PORTD of the PIC18 are connected to LEDs and switches, respectively. Program 10-1 shows a PIC18 program with the following parts: (a) sends the message "YES" once to the PC screen, (b) gets data on switches and transmits it via the serial port to the PC's screen, and (c) receives any key press sent by HyperTerminal and puts it on LEDs. The program performs part (a) once, but parts (b) and (c) continuously. It uses the 9600 baud rate for XTAL = 10 MHz.

```
;Program 10-1 Transmit and Receive
      ORG 0
  ;initialize the serial ports for both transmit and receive
      MOVLW B'00100100'       ;enable transmit, choose high baud
      MOVWF TXSTA             ;write to reg
      MOVLW B'10010000'       ;enable receive, serial port itself
      MOVWF RCSTA             ;write to reg
      MOVLW D'15'             ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG             ;write to reg
      BSF   RCSTA, SPEN       ;enable the serial port itself
      BCF   TRISC, TX         ;make TX pin of PORTC an output
      BSF   TRISC, RX         ;make RX pin of PORTC an input
      CLRF  TRISB             ;make port B an output port
      SETF  TRISD             ;make port D an input port
  ;send the message "YES"
      MOVLW 'Y'               ;ASCII letter 'Y' to be transferred
      CALL  TRANS
      MOVLW 'E'               ;ASCII letter 'E' to be transferred
      CALL  TRANS
      MOVLW 'S'               ;ASCII letter 'S' to be transferred
      CALL  TRANS
  ;get a byte from switches and transmit data to PC screen
OVER  MOVF PORTD,W            ;get a byte from SW of PORTD
      CALL  TRANS             ;transmit it via serial port
  ;as keys are pressed on PC receive data and put it on LEDs
      CALL  RECV              ;receive the byte from serial port
      MOVWF PORTB             ;display it on LEDS of PORTB
      BRA   OVER              ;keep doing it
;--serial transfer (WREG needs the byte to be transmitted)
TRANS
S1    BTFSS PIR1, TXIF        ;wait until the last bit is gone
      BRA   S1                ;
      MOVWF TXREG             ;load the value to be transferred
      RETURN                  ;return to caller
;----serial data receive subroutine (WREG = received byte)
```

```
RECV  BTFSS PIR1, RCIF        ;check for ready
      BRA   RECV              ;stay in loop
      MOVF RCREG,W            ;save value in WREG
      RETURN
```

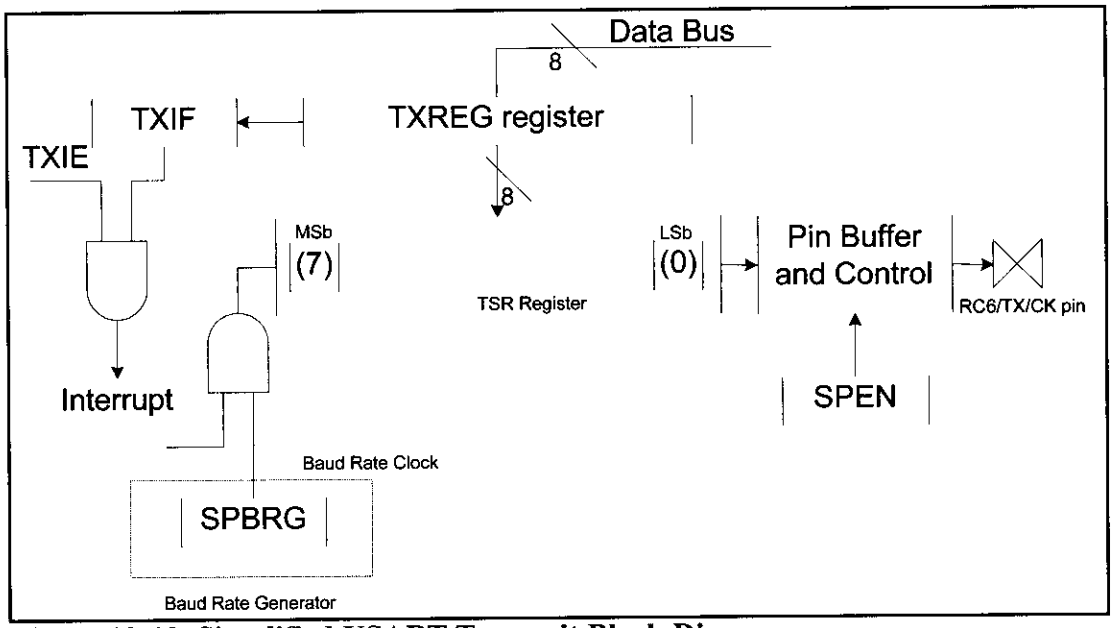


Figure 10-12. Simplified USART Transmit Block Diagram

Interrupt-based data transfer

By now you might have noticed that it is a waste of the microcontroller's time to poll the TXIF and RXIF flags. In order to avoid wasting the microcontroller's time we use interrupts instead of polling. In Chapter 11, we will show how to use interrupts to program the PIC18's serial communication port.

Review Questions

1. Which register of the PIC18 is used to set the baud rate?
2. If XTAL = 10 MHz, what frequency is used by the UART to set the baud rate (assuming default mode)?
3. Which bit of the TXSTA register is used to set the low or high baud rate?
4. With XTAL = 10 MHz, what value should be loaded into SPBRG to have a 9600 baud rate? Give the answer in both decimal and hex.
5. To transmit a byte of data serially, it must be placed in register _____.
6. TXSTA stands for _____ and it is a(n) ____-bit register.
7. Which register is used to set the data frame size?
8. True or false. TXSTA is a bit-addressable register.
9. When is TXIF raised? When is it cleared?
10. Which register has the BRGH bit, and what is its status when the PIC18 is powered up?

SECTION 10.4: PIC18 SERIAL PORT PROGRAMMING IN C

This section shows C programming of the serial ports for the PIC18 chip.

Transmitting and receiving data in PIC18 C

As we saw in Chapter 7, all the special function registers (SFRs) of the PIC18 are accessible directly in C18 compilers by using the appropriate header file. Examples 10-11 through 10-15 show how to program the serial port in PIC18 C. Connect your PIC18 Trainer to the PC's COM port and use HyperTerminal to test the operation of these examples. Notice that Examples 10-11 through 10-15 are C versions of the Assembly programs in the last section.

Example 10-11

Write a C program for the PIC18 to transfer the letter 'G' serially at 9600 baud, continuously. Use 8-bit data and 1 stop bit. Assume XTAL = 10 MHz.

Solution:

```
#include <P18F4580.h>
void main(void)
  {
    TXSTA=0x20;              //choose low baud rate,8-bit
    SPBRG=15;                //9600 baud rate/ XTAL = 10 MHz
    TXSTAbits.TXEN=1;
    RCSTAbits.SPEN=1;

    while(1)
      {
        TXREG='G';           //place value in buffer
        while(PIR1bits.TXIF==0);      //wait until all gone
      }
  }
```

Review Questions

1. True or false. All the SFR registers of PIC18 are accessible in the C18 C compiler.
2. True or false. C18 compilers support the bit-addressable registers of the PIC18.
3. True or false. The TXIF flag is cleared the moment we write a character to the TXREG register.
4. Which register is used to set the baud rate?
5. To which register does the BRGH bit belong, and what is its role?

Example 10-12

Write a PIC18 C program to transfer the message "YES" serially at 9600 baud, 8-bit data, and 1 stop bit. Do this continuously.

Solution:

```
#include <P18F458.h>
void SerTx(unsigned char);
void main(void)
  {
    TXSTA=0x20;              //choose low baud rate,8-bit
    SPBRG=15;                //9600 baud rate/ XTAL = 10 MHz
    TXSTAbits.TXEN=1;
    RCSTAbits.SPEN=1;
    while(1)
      {
        SerTx('Y');
        SerTx('E');
        SerTx('S');
      }
  }
void SerTx(unsigned char c)
  {
    while(PIR1bits.TXIF==0);     //wait until transmitted
    TXREG=c;                     //place character in buffer
  }
```

Example 10-13

Program the PIC18 in C to receive bytes of data serially and put them on PORTB. Set the baud rate at 9600, 8-bit data, and 1 stop bit.

Solution:

```
#include <P18F458.h>
void main (void)
  {
    TRISB = 0;                   //PORTB an output
    RCSTA=0x90;                  //enable serial port and receiver
    SPBRG=15;                    //9600 baud rate/ XTAL = 10 MHz
    while(1)                     //repeat forever
      {
        while(PIR1bits.RCIF==0); //wait to receive
        PORTB=RCREG;             //save value
      }
  }
```

Example 10-14

Write an C18 program to send two different strings to the serial port. Assuming that SW is connected to pin PORTB.5, monitor its status and make a decision as follows:
SW = 0: send your first name
SW = 1: send your last name
Assume XTAL = 10 MHz, baud rate of 9600, and 8-bit data.

Solution:

```
#include <P18F458.h>
#define MYSW PORTBbits.RB5        //INPUT SWITCH
void main(void)
  {
    unsigned char z;
    unsigned char fname[]="ALI";
    unsigned char lname[]="SMITH";
    TRISBbits.TRISB5 = 1;  //an input
    TXSTA=0x20;            //choose low baud rate, 8-bit
    SPBRG=15;              //9600 baud rate/ XTAL = 10 MHz
    TXSTAbits.TXEN=1;
    RCSTAbits.SPEN=1;
    if(MYSW==0)                          //check switch
      {
        for(z=0;z<3;z++)                 //write name
          {
            while(PIR1bits.TXIF==0); //wait for transmit
            TXREG=fname[z];          //place char in buffer
          }
      }
    else
      {
        for(z=0;z<5;z++)                 //write name
          {
            while(PIR1bits.TXIF==0); //wait for transmit
            TXREG=lname[z];          //place value in buffer
          }
      }
     while(1);
  }
```

Example 10-15

Write a PIC18 C program to send the two messages "Normal Speed" and "High Speed" to the serial port. Assuming that SW is connected to pin PORTB.0, monitor its status and set the baud rate as follows:
SW = 0 9600 baud rate
SW = 1 38400 baud rate
Assume that XTAL = 10 MHz for both cases.

Solution:

```
#include <P18F458.h>
#define MYSW PORTBbits.RB5        //INPUT SWITCH
void main(void)
  {
    unsigned char z;
    unsigned char Mess1[]="Normal Speed";
    unsigned char Mess2[]="High Speed";
    TRISBbits.TRISB5 = 1;  //an input
    TXSTA=0x20;            //choose low baud rate, 8-bit
    SPBRG=15;              //9600 baud rate/ XTAL = 10 MHz
    TXSTAbits.TXEN=1;
    RCSTAbits.SPEN=1;
    if(MYSW==0)
      {
        for(z=0;z<12;z++)
          {
            while(PIR1bits.TXIF==0); //wait for transmit
            TXREG=Mess1[z];          //place value in buffer
          }
      }
    else
      {
        TXSTA=TXSTA|0x4;             //for high speed
        for(z=0;z<10;z++)
          {
            while(PIR1bits.TXIF==0); //wait for transmit
            TXREG=Mess2[z];          //place value in buffer
          }
      }
     while(1);
  }
```

SUMMARY

This chapter began with an introduction to the fundamentals of serial communication. Serial communication, in which data is sent one bit a time, is used in situations where data is sent over significant distances because in parallel communication, where data is sent a byte or more a time, great distances can cause distortion of the data. Serial communication has the additional advantage of allowing transmission over phone lines. Serial communication uses two methods: synchronous and asynchronous. In synchronous communication, data is sent in blocks of bytes; in asynchronous, data is sent one byte at a time. Data communication can be simplex (can send but cannot receive), half duplex (can send and receive, but not at the same time), or full duplex (can send and receive at the same time). RS232 is a standard for serial communication connectors.

The PIC18's UART was discussed. We showed how to interface the PIC18 with an RS232 connector and change the baud rate of the PIC18. In addition, we described the serial communication features of the PIC18, and programmed the PIC18 for serial data communication. We also showed how to program the serial port of the PIC18 chip in Assembly and C.

PROBLEMS

SECTION 10.1: BASICS OF SERIAL COMMUNICATION

1. Which is more expensive, parallel or serial data transfer?
2. True or false. 0- and 5-V digital pulses can be transferred on the telephone without being converted (modulated).
3. Show the framing of the letter ASCII 'Z' (0101 1010), no parity, 1 stop bit.
4. If there is no data transfer and the line is high, it is called _____ (mark, space).
5. True or false. The stop bit can be 1, 2, or none at all.
6. Calculate the overhead percentage if the data size is 7, 1 stop bit, no parity.
7. True or false. RS232 voltage specification is TTL compatible.
8. What is the function of the MAX 232 chip?
9. True or false. DB-25 and DB-9 are pin compatible for the first 9 pins.
10. How many pins of the RS232 are used by the IBM serial cable, and why?
11. True or false. The longer the cable, the higher the data transfer baud rate.
12. State the absolute minimum number of signals needed to transfer data between two PCs connected serially. What are those?
13. If two PCs are connected through the RS232 without the modem, both are configured as a _____ (DTE, DCE) -to- _____ (DTE, DCE) connection.
14. State the nine most important signals of the RS232.
15. Calculate the total number of bits transferred if 200 pages of ASCII data are sent using asynchronous serial data transfer. Assume a data size of 8 bits, 1 stop bit, and no parity. Assume each page has 80x25 of text characters.
16. In Problem 15, how long will the data transfer take if the baud rate is 9,600?

SECTION 10.2: PIC18 CONNECTION TO RS232

17. The MAX232 DIP package has _____ pins.
18. For the MAX232, indicate the V_{CC} and GND pins.
19. The MAX233 DIP package has ____ pins.
20. For the MAX233, indicate the V_{CC} and GND pins.
21. Is the MAX232 pin compatible with the MAX233?
22. State the advantages and disadvantages of the MAX232 and MAX233.
23. MAX232/233 has ____ line driver(s) for the RX wire.
24. MAX232/233 has ____ line driver(s) for the TX wire.
25. Show the connection of pins TX and RX of the PIC18 to a DB-9 RS232 connector via the second set of line drivers of MAX232.
26. Show the connection of the TX and RX pins of the PIC18 to a DB-9 RS232 connector via the second set of line drivers of MAX233.
27. What is the advantage of the MAX233 over the MAX232 chip?
28. Which pins of the PIC18 are set aside for serial communication, and what are their functions?

SECTION 10.3: PIC18 SERIAL PORT PROGRAMMING IN ASSEMBLY

29. Which of the following baud rates are supported by the HyperTerminal program in PC?
    (a) 4,800 (b) 3,600 (c) 9,600
    (d) 1,800 (e) 1,200 (f) 19,200
30. Which timer of the PIC18 is used for baud rate programming?
31. Which bit of the TXSTA is used for baud rate speed?
32. What is the role of the TXREG register in serial data transfer?
33. TXREG is a(n) ____-bit register.
34. What is the role of the TXSTA register in serial data transfer?
35. TXSTA is a(n) ____-bit register.
36. For XTAL = 10 MHz, find the SPBRG value (in both decimal and hex) for each of the following baud rates.
    (a) 9,600 (b) 4,800 (c) 1,200
37. What is the baud rate if we use SPBRG = 15 to program the baud rate? Assume XTAL = 10 MHz.
38. Write a PIC18 program to transfer serially the letter 'Z' continuously at 1,200 baud rate. Assume XTAL = 10 MHz.
39. Write a PIC18 program to transfer serially the message "The earth is but one country and mankind its citizens" continuously at 57,600 baud rate. Assume XTAL = 10 MHz.
40. When is the TXIF flag bit raised or cleared?
41. When is the RCIF flag bit raised or cleared?
42. To which register do RCIF and TXIF belong? Is that register bit-addressable?
43. What is the role of the SPEN bit in the RCSTA register?
44. In a given situation we cannot accept reception of any serial data. How do you block such a reception with a single instruction?

45. To which register does the BRGH bit belong? State its role in rate of data transfer.
46. Is the BRGH bit HIGH or LOW when the PIC18 is powered up?
47. Find the SPBRG for the following baud rates if XTAL = 16 MHz and BRGH = 0.
    (a) 9600 (b) 19200
    (c) 38400 (d) 57600
48. Find the SPBRG for the following baud rates if XTAL = 16 MHz and BRGH = 1.
    (a) 9600 (b) 19200
    (c) 38400 (d) 57600
49. Find the SPBRG for the following baud rates if XTAL = 20 MHz and BRGH = 0.
    (a) 9600 (b) 19200
    (c) 38400 (d) 57600
50. Find the SPBRG for the following baud rates if XTAL = 20 MHz and BRGH = 1.
    (a) 9600 (b) 19200
    (c) 38400 (d) 57600
51. Find the baud rate error for Problem 47.
52. Find the baud rate error for Problem 48.

SECTION 10.4: PIC18 SERIAL PORT PROGRAMMING IN C

53. Write an PIC18 C program to transfer serially the letter 'Z' continuously at 1,200 baud rate.
54. Write an PIC18 C program to transfer serially the message "The earth is but one country and mankind its citizens" continuously at 57,600 baud rate.

ANSWERS TO REVIEW QUESTIONS

SECTION 10.1: BASICS OF SERIAL COMMUNICATION

1. Faster, more expensive
2. False; it is simplex.
3. True
4. Asynchronous
5. With 0100 0101 binary the bits are transmitted in the sequence:
   (a) 0 (start bit) (b) 1 (c) 0 (d) 1 (e) 0 (f) 0 (g) 0 (h) 1 (i) 0 (j) 1 (stop bit)
6. 2 bits (one for the start bit and one for the stop bit). Therefore, for each 8-bit character, a total of 10 bits is transferred.
7. 10000 × 10 = 100000 total bits transmitted. 100000 / 9600 = 10.4 seconds; 2 / 10 = 20%.
8. True
9. +3 to +25 V
10. True
11. One
12. COM 1, COM 2

SECTION 10.2: PIC18 CONNECTION TO RS232

1. True
2. Pins RC6 and RC7. Pin RC6 is for TX and pin RC7 for RX.
3. They are used for converting from RS232 voltage levels to TTL voltage levels and vice versa.
4. Two, two
5. It does not need the four capacitors that MAX232 must have.

SECTION 10.3: PIC18 SERIAL PORT PROGRAMMING IN ASSEMBLY

1. SPBRG
2. 156,250 Hz
3. BRGH
4. 15 in decimal (or F in hex) because 156,250/9600 – 1 = 15
5. TXREG
6. Transmit Status and Control Register, 8
7. TXSTA
8. True
9. It is raised during transfer of the stop bit. It is cleared when we write a byte to TXREG to be transmitted.
10. TXSTA; it is low upon power-on reset.

SECTION 10.4: PIC18 SERIAL PORT PROGRAMMING IN C

1. True
2. True
3. True
4. SPBRG
5. TXSTA. It allows us to quadruple the baud rate with the same crystal frequency.

CHAPTER 11

INTERRUPT PROGRAMMING IN ASSEMBLY AND C

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Contrast and compare interrupts versus polling**
- **>> Explain the purpose of the ISR (interrupt service routine)**
- **>> List all the major interrupts of the PIC18**
- **>> Explain the purpose of the interrupt vector table**
- **>> Enable or disable PIC18 interrupts**
- **>> Program the PIC18 timers using interrupts**
- **>> Describe the external hardware interrupts of the PIC18**
- **>> Program the PIC18 for interrupt-based serial communication**
- **>> Define the interrupt priority of the PIC18**
- **>> Program PIC interrupts in C**

In this chapter we explore the concept of the interrupt and interrupt programming. In Section 11.1, the basics of PIC18 interrupts are discussed. In Section 11.2, interrupts belonging to timers are discussed. External hardware interrupts are discussed in Section 11.3, while the interrupt related to serial communication is presented in Section 11.4. In Section 11.5, we cover the interrupt associated with PORTB. In Section 11.6, we cover interrupt priority. Throughout this chapter, we provide examples in both Assembly and C.

SECTION 11.1: PIC18 INTERRUPTS

In this section, first we examine the difference between polling and interrupts and then describe the various interrupts of the PIC18.

Interrupts vs. polling

A single microcontroller can serve several devices. There are two methods by which devices receive service from the microcontroller: interrupts or polling. In the *interrupt* method, whenever any device needs the microcontroller's service, the device notifies it by sending an interrupt signal. Upon receiving an interrupt signal, the microcontroller stops whatever it is doing and serves the device. The program associated with the interrupt is called the *interrupt service routine* (ISR) or *interrupt handler*. In *polling*, the microcontroller continuously monitors the status of a given device; when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced. Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller. The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time, of course); each device can get the attention of the microcontroller based on the priority assigned to it. The polling method cannot assign priority because it checks all devices in a round-robin fashion. More importantly, in the interrupt method the microcontroller can also ignore (mask) a device request for service. This also is not possible with the polling method. The most important reason that the interrupt method is preferable is that the polling method wastes much of the microcontroller's time by polling devices that do not need service. So interrupts are used to avoid tying down the microcontroller. For example, in discussing timers in Chapter 9 we used the bit test instruction "`BTFSS TMR0IF`" and waited until the timer rolled over, and while we were waiting we could not do anything else. That is a waste of microcontroller time that could have been used to perform some useful tasks. In the case of the timer, if we use the interrupt method, the microcontroller can go about doing other tasks, and when the TMR0IF flag is raised, the timer will interrupt the microcontroller in whatever it is doing.

Interrupt service routine

For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt is invoked, the microcontroller runs the interrupt service routine. Generally, in most microprocessors, for every interrupt there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the *interrupt vector*

table. In the case of the PIC18, there are only two locations for the interrupt vector table, locations 0008 and 0018, as shown in Table 11-1. We will discuss the difference between these two in Section 11.6 when we cover interrupt priority.

Table 11-1: Interrupt Vector Table for the PIC18

Interrupt	ROM Location (Hex)
Power-on Reset	0000
High Priority Interrupt	0008 (Default upon power-on reset)
Low Priority Interrupt	0018 (See Section 11.6)

Steps in executing an interrupt

Upon activation of an interrupt, the microcontroller goes through the following steps:

1. It finishes the instruction it is executing and saves the address of the next instruction (program counter) on the stack.
2. It jumps to a fixed location in memory called the interrupt vector table. The interrupt vector table directs the microcontroller to the address of the interrupt service routine (ISR).
3. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETFIE (return from interrupt exit).
4. Upon executing the RETFIE instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then it starts to execute from that address.

Notice from Step 4 the critical role of the stack. For this reason, we must be careful in manipulating the stack contents in the ISR. Specifically, in the ISR, just as in any CALL subroutine, the number of pushes and pops must be equal.

Sources of interrupts in the PIC18

There are many sources of interrupts in the PIC18, depending on which peripheral is incorporated into the chip. The following are some of the most widely used sources of interrupts in the PIC18:

1. There is an interrupt set aside for each of the timers, Timers 0, 1, 2, and so on. See Section 11.2.
2. Three interrupts are set aside for external hardware interrupts. Pins RB0 (PORTB.0), RB1 (PORTB.1), and RB2 (PORTB.2) are for the external hardware interrupts INT0, INT1, and INT2, respectively. See Section 11.3.
3. Serial communication's USART has two interrupts, one for receive and another for transmit. See Section 11.4.
4. The PORTB-Change interrupt. See Section 11.5.
5. The ADC (analog-to-digital converter). See Chapter 13.
6. The CCP (compare capture pulse-width-modulation). See Chapters 15 and 17.

The PIC18 has many more interrupts than the above list shows. We will cover them throughout the book as we study the peripherals of the PIC18. Notice in Table 11-1 that a limited number of bytes is set aside for high-priority interrupts. For example, a total of 8 bytes, from location 0008 to 000017H, are set aside for high-priority interrupts. Normally, the service routine for an interrupt is too long to fit in the memory space allocated. For that reason, a GOTO instruction is placed in the vector table to point to the address of the ISR. The rest of the bytes allocated to the interrupt are unused. In upcoming sections of this chapter, we will see many examples of interrupt programming that clarify these concepts.

From Table 11-1, also notice that only 8 bytes of ROM space are assigned to the reset pin. They are ROM address locations 0–7. For this reason, in our program we put the GOTO as the first instruction and redirect the processor away from the interrupt vector table, as shown in Figure 11-1. In the next section we will see how this works in the context of some examples.

```
          ORG  0     ;wake-up ROM reset location
          GOTO MAIN  ;bypass interrupt vector table

;---- the wake-up program
          ORG  100H
MAIN:     .... ;enable interrupt flags
          ....
          END
```

Figure 11-1. Redirecting the PIC18 from the Interrupt Vector Table at Power-Up

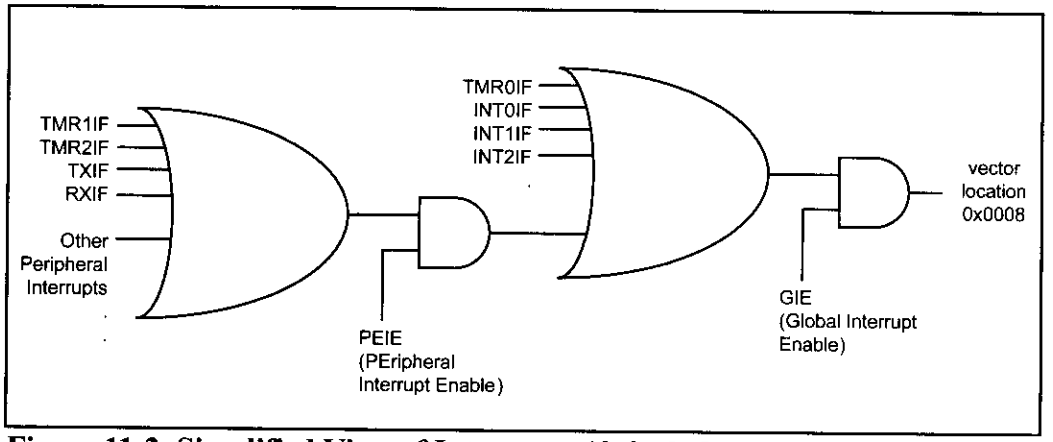


Figure 11-2. Simplified View of Interrupts (default for power-on reset)

Enabling and disabling an interrupt

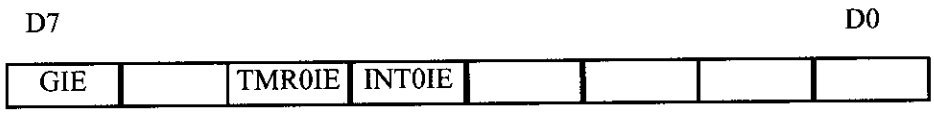
Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated. The interrupts must be enabled (unmasked) by software in order for the microcontroller to respond to them. The D7 bit of the INTCON (Interrupt Control) register is responsible for

enabling and disabling the interrupts globally. Figure 11-3 shows the INTCON register. The GIE bit makes the job of disabling all the interrupts easy. With a single instruction (BCF INTCON,GIE), we can make GIE = 0 during the operation of a critical task. See Figure 11-2.

Steps in enabling an interrupt

To enable any one of the interrupts, we take the following steps:

1. Bit D7 (GIE) of the INTCON register must be set to HIGH to allow the interrupts to happen. This is done with the "BSF INTCON, GIE" instruction.
2. If GIE = 1, each interrupt is enabled by setting to HIGH the interrupt enable (IE) flag bit for that interrupt. Because there are a large number of interrupts in the PIC18, we have many registers holding the interrupt enable bit. Figure 11-2 shows that the INTCON has interrupt enable bits for Timer0 (TMR0IE) and external interrupt 0 (INT0IE). As we study each of peripherals throughout the book we will examine the registers holding the interrupt enable bits. It must be noted that if GIE = 0, no interrupt will be responded to, even if the corresponding interrupt enable bit is high. To understand this important point look at Example 11-1.
3. As shown in Figures 11-2 and 11-3, for some of the peripheral interrupts such as TMR1IF, TMR2IF, and TXIF, we have to enable the PEIE flag in addition to the GIE bit.



GIE (Global Interrupt Enable)
GIE = 0 Disables all interrupts. If GIE = 0, no interrupt is acknowledged, even if they are enabled individually.
If GIE = 1, interrupts are allowed to happen. Each interrupt source is enabled by setting the corresponding interrupt enable bit.

TMR0IE Timer0 interrupt enable
= 0 Disables Timer0 overflow interrupt
= 1 Enables Timer0 overflow interrupt

INT0IE Enables or disables external interrupt 0
= 0 Disables external interrupt 0
= 1 Enables external interrupt 0

These bits, along with the GIE, must be set high for an interrupt to be responded to. Upon activation of the interrupt, the GIE bit is cleared by the PIC18 itself to make sure another interrupt cannot interrupt the microcontroller while it is servicing the current one. At the end of the ISR, the RETFIE instruction will make GIE = 1 to allow another interrupt to come in.

PEIE (PEripheral Interrupt Enable)
For many of the peripherals, such as Timers 1, 2, .. and the serial port, we must enable this bit in addition to the GIE bit. (See Figure 11-2.)

Figure 11-3. INTCON (Interrupt Control) Register

Example 11-1

Show the instructions to (a) enable (unmask) the Timer0 interrupt and external hardware interrupt 0 (INT0), and (b) disable (mask) the Timer0 interrupt, then (c) show how to disable (mask) all the interrupts with a single instruction.

Solution:

```
(a)   BSF INTCON,TMR0IE   ;enable(unmask) Timer0 interrupt
      BSF INTCON,INT0IE ;enable external interrupt 1(INT0)
      BSF INTCON,GIE ;allow interrupts to come in
```

We can perform the above actions with the following two instructions:

```
      MOVLW B'10110000'   ;GIE = 1, TMR0IF = 1,INTIF0 = 1
      MOVWF INTCON        ;load the INTCON reg

(b)   BCF INTCON,TMR0IE ;mask (disable) Timer0 interrupt

(c)   BCF INTCON,GIE      ;mask all interrupts globally
```

Review Questions

1. Of the interrupt and polling methods, which one avoids tying down the microcontroller?
2. Give the name of the interrupts in the INTCON register.
3. Upon power-on reset of the PIC18, what memory area is assigned to the interrupt vector table? Can the programmer change the memory space assigned to the table?
4. What is the content of D7 (GIE) of the INTCON register upon reset, and what does it mean?
5. Show the instruction needed to enable the TMR0 interrupt.
6. What address in the interrupt vector table is assigned to high-priority and low-priority interrupts?

SECTION 11.2: PROGRAMMING TIMER INTERRUPTS

In Chapter 9 we discussed how to use Timers 0, 1, 2, and 3 with the polling method. In this section we use interrupts to program the PIC18 timers. Please review Chapter 9 before you study this section.

Rollover timer flag and interrupt

In Chapter 9 we stated that the timer flag is raised when the timer rolls over. In that chapter, we also showed how to monitor the timer flag with the instruction "`BTFSS TMR0IF`". In polling TMR0IF, we have to wait until TMR0IF is raised. The problem with this method is that the microcontroller is tied down waiting for TMR0IF to be raised, and cannot do anything else. Using interrupts avoids tying down the controller. If the timer interrupt in the interrupt register is enabled, TMR0IF is raised whenever the timer rolls over and the microcontroller jumps to the interrupt vector table to service the ISR. In this way, the microcontroller can do other things until it is notified that the timer has rolled over. To use an interrupt in place of polling, first we must enable the interrupt because all the interrupts are masked upon power-on reset. The TMRxIE bit enables the interrupt for a given timer. TMRxIE bits are held by various registers as shown in Table 11-2. In the case of Timer0, the INTCON register (Figure 11-4) contains the TMR0IE bit, and PIE1 (peripheral interrupt enable) holds the TMR1IE bit. See Figure 11-5 and Program 11-1.

Table 11-2: Timer Interrupt Flag Bits and Associated Registers

Interrupt	Flag Bit	Register	Enable Bit	Register
Timer0	TMR0IF	INTCON	TMR0IE	INTCON
Timer1	TMR1IF	PIR1	TMR1IE	PIE1
Timer2	TMR2IF	PIR1	TMR2IE	PIE1
Timer3	TMR3IF	PIR3	TMR3IE	PIE2

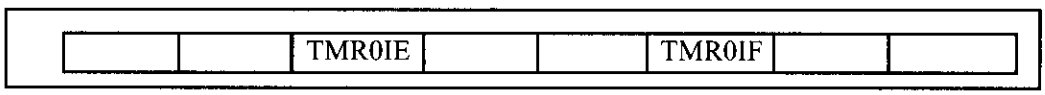


Figure 11-4. INTCON Register with Timer0 Interrupt Enable and Interrupt Flag

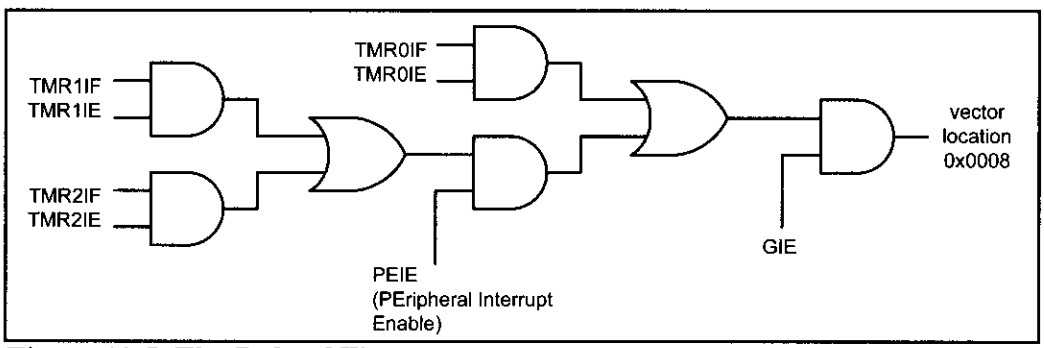


Figure 11-5. The Role of Timer Interrupt Enable Flag (TMRxIE)

Note: The TMRxIP (timer interrupt priority) flag is not shown. TMRxIP is used to force the interrupt to land at vector location 0x0018. See Section 11.6.

Notice the following points about Program 11-1:

1. We must avoid using the memory space allocated to the interrupt vector table. Therefore, we place all the initialization codes in memory starting at an address such as 100H. The GOTO instruction is the first instruction that the PIC18 executes when it is awakened at address 00000 upon power-on reset (POR). The GOTO instruction at address 00000 redirects the controller away from the interrupt vector table.
2. In the MAIN program, we enable (unmask) the Timer0 interrupt with instruction "`BSF INTCON, TMR0IE`" followed by the instruction "`BSF INTCON, GIE`" to enable all interrupts globally.
3. In the MAIN program, we initialize the Timer0 register and then enter an infinite loop to keep the CPU busy. This could be a real-world application being executed by the CPU. In this case, the loop gets data from PORTC and sends it to PORTD. While the PORTC data is brought in and issued to PORTD continuously, the TMR0IF flag is raised as soon as Timer0 rolls over, and the microcontroller gets out of the loop and goes to 00008H to execute the ISR associated with Timer0. At this point, the PIC18 clears the GIE bit (D7 of INTCON) to indicate that it is currently serving an interrupt and cannot be interrupted again; in other words, no interrupt inside the interrupt. In Section 11.6, we show how to allow an interrupt inside an interrupt.
4. The ISR for Timer0 is located starting at memory location 00200H because it is too large to fit into address space 08–17H, the address allocated to high-priority interrupts.
5. In the ISR for Timer0, notice that the "`BCF INTCON, TMR0IF`" instruction is needed before the RETFIE instruction. This will ensure that a single interrupt is serviced once and is not recognized as multiple interrupts.
6. RETFIE must be the last instruction of the ISR. Upon execution of the RETFIE instruction, the PIC18 automatically enables the GIE (D7 of the INTCON register) to indicate that it can accept new interrupts.

Program 11-1: For this program, we assume that PORTC is connected to 8 switches and PORTD to 8 LEDs. This program uses Timer0 to generate a square wave on pin PORTB.5, while at the same time data is being transferred from PORTC to PORTD.

```
;Program 11-1
          ORG   0000H
          GOTO  MAIN          ;bypass interrupt vector table
     ;--on default all interrupts land at address 00008
          ORG   0008H         ;interrupt vector table
          BTFSS INTCON,TMR0IF   ;Timer0 interrupt?
          RETFIE                ;No. Then return to main
          GOTO  T0_ISR          ;Yes. Then go Timer0 ISR
     ;--main program for initialization and keeping CPU busy
          ORG   00100H        ;after vector table space
     MAIN BCF   TRISB,5       ;PB5 as an output
          CLRF  TRISD         ;make PORTD output
          SETF  TRISC         ;make PORTC input
```

```
        MOVLW 0x08        ;Timer0,16-bit,
                          ;no prescale,internal clk
        MOVWF T0CON       ;load T0CON reg
        MOVLW 0xFF        ;TMR0H = FFH, the high byte
        MOVWF TMR0H       ;load Timer0 high byte
        MOVLW 0xF2        ;TMR0L = F2H, the low byte
        MOVWF TMR0L       ;load Timer0 low byte
        BCF INTCON,TMR0IF;clear timer interrupt flag bit
        BSF T0CON,TMR0ON      ;start Timer0
        BSF INTCON,TMR0IE     ;enable Timer 0 interrupt
        BSF INTCON,GIE   ;enable interrupts globally
;--keeping CPU busy waiting for interrupt
OVER  MOVFF PORTC,PORTD ;send data from PORTC to PORTD
      BRA OVER              ;stay in this loop forever
;--------------------------ISR for Timer 0
T0_ISR
      ORG 200H
      MOVLW 0xFF        ;TMR0H = FFH, the high byte
      MOVWF TMR0H       ;load Timer0 high byte
      MOVLW 0xF2        ;TMR0L = F2H, the low byte
      MOVWF TMR0L       ;load Timer0 low byte
      BTG   PORTB,5     ;toggle RB5
      BCF INTCON,TMR0IF ;clear timer interrupt flag bit
EXIT  RETFIE ;return from interrupt (See Example 11-2)
      END
```

Example 11-2

What is the difference between the RETURN and RETFIE instructions? Explain why we cannot use RETURN instead of RETFIE as the last instruction of an ISR.

Solution:

Both perform the same actions of popping off the top bytes of the stack into the program counter, and making the PIC18 return to where it left off. However, RETFIE also performs the additional task of clearing the GIE flag, indicating that the servicing of the interrupt is over and the PIC18 now can accept a new interrupt. If you use RETURN instead of RETFIE as the last instruction of the interrupt service routine, you simply block any new interrupt after the first interrupt, because the GIE would indicate that the interrupt is still being serviced.

In Program 11-1, the Timer0 ISR (interrupt service routine) was too long to be placed in memory locations allocated to the high interrupt (addresses of 0008–00017H). There was enough space, however, to test to see which interrupt was the cause of landing at the 0008 address. Very often, we go from address 0008 to another address with a larger space to check the source of the interrupt, given the fact that the PIC18 has so many interrupts and we have limited space at address 0008. See Program 11-2.

Program 11-2 uses Timer0 and Timer1 interrupts to generate square waves on pins RB1 and RB7 respectively, while data is being transferred from PORTC to PORTD.

```
;Program 11-2
          ORG   0000H
          GOTO  MAIN          ;bypass interrupt vector table
     ;--on default all interrupts land at address 00008
          ORG   0008H         ;interrupt vector table
          GOTO CHK_INT        ;go to an address with more space
     ;--check to see the source of interrupt
          ORG   0040H         ;we got here from 0008
     CHK_INT
          BTFSC INTCON,TMR0IF   ;Is it Timer0 interrupt?
          BRA   T0_ISR          ;Yes. Then branch to T0_ISR
          BTFSC PIR1,TMR1IF     ;Is it Timer1 interrupt?
          BRA   T1_ISR          ;Yes. Then branch to T1_ISR
          RETFIE                ;No. Then return to main
     ;-main program for initialization and keeping CPU busy
          ORG   0100H  ;somewhere after vector table space
     MAIN BCF   TRISB,1     ;PB1 as an output
          BCF   TRISB,7     ;PB7 as an output
          CLRF  TRISD       ;make PORTD output
          SETF  TRISC       ;make PORTC input
          MOVLW 0x08        ;Timer0,16-bit,
                            ;no prescale,internal clk
          MOVWF T0CON       ;load T0CON reg
          MOVLW  0xFF       ;TMR0H = FFH, the high byte
          MOVWF  TMR0H      ;load Timer0 high byte
          MOVLW 0xF2        ;TMR0L = F2H, the low byte
          MOVWF TMR0L       ;load Timer0 low byte
          BCF INTCON,TMR0IF;clear Timer0 interrupt flag bit
          MOVLW 0x0         ;Timer1,16-bit,
                            ;no prescale,internal clk
          MOVWF T1CON       ;load T1CON reg
          MOVLW  0xFF       ;TMR1H = FFH, the high byte
          MOVWF  TMR1H      ;load Timer0 high byte
          MOVLW 0xF2        ;TMR1L = F2H, the low byte
          MOVWF TMR1L       ;load Timer1 low byte
          BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
          BSF INTCON,TMR0IE     ;enable Timer0 interrupt
          BSF PIE1,TMR1IE       ;enable Timer1 interrupt
          BSF INTCON,PEIE ;enable peripheral interrupts
          BSF INTCON,GIE        ;enable interrupts globally
          BSF T0CON,TMR0ON      ;start Timer0
          BSF T1CON,TMR1ON      ;start Timer1
     ;--keeping CPU busy waiting for interrupt
     OVER MOVFF PORTC,PORTD ;send data from PORTC to PORTD
          BRA OVER              ;stay in this loop forever
     ;--------------------------ISR for Timer 0
     T0_ISR
          ORG 200H
```

```
        MOVLW  0xFF       ;TMR0H = FFH, the high byte
        MOVWF  TMR0H      ;load Timer0 high byte
        MOVLW 0xF2        ;TMR0L = F2H, the low byte
        MOVWF TMR0L       ;load Timer0 low byte
        BTG   PORTB,1     ;toggle PB1
        BCF INTCON,TMR0IF ;clear timer interrupt flag bit
        GOTO CHK_INT
;--------------------------ISR for Timer1
T1_ISR
        ORG 300H
        MOVLW  0xFF       ;TMR1H = FFH, the high byte
        MOVWF  TMR1H      ;load Timer0 high byte
        MOVLW 0xF2        ;TMR1L = F2H, the low byte
        MOVWF TMR1L       ;load Timer1 low byte
        BTG PORTB,7
        BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
        GOTO CHK_INT
        END
```

Notice that the addresses 0040H, 0100H, 00200H, and 0300H that we used in Program 11-2 are all arbitrary and can be changed to any addresses we want. The only addresses that we have no choice on are the power-on reset location of 0000 and high-priority address of 0008 because they were fixed at the time of the PIC18 design.

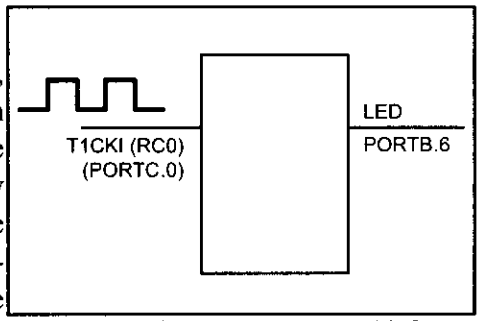


Figure 11-6. For Program 11-3

Program 11-3 has two interrupts: (1) PORTD counts up everytime Timer0 overflows. It uses 16-bit mode of Timer0 with the largest prescale possible; (2) 1-Hz pulse is fed into Timer1 where Timer1 is used as counter and counts up. Whenever the count reaches 200, it will toggle the pin PORTB.6.

```
;Program 11-3
        ORG   0000H
        GOTO  MAIN        ;bypass interrupt vector table
;--on default all interrupts land at address 00008
        ORG   0008H
        GOTO CHK_INT
;-----------------find the interrupt source
        ORG   0040H
CHK_INT
        BTFSC INTCON,TMR0IF   ;Is it Timer0 interrupt?
        BRA   T0_ISR          ;Yes. Then branch to T0_ISR
        BTFSC PIR1,TMR1IF     ;Is it Timer1 interrupt?
        BRA   T1_ISR          ;Yes. Then branch to T1_ISR
        RETFIE                ;No. Then return to main
;--the main program for initialization
        ORG   00100H      ;after vector table space
MAIN    BSF   TRISC,T13CKI     ;PORTC.0 as an input
        CLRF TRISD        ;make PORTD output
        BCF TRISB,6       ;make RB6 output
```

```
        MOVLW 0x08        ;16-bit, prescale = 256,
                          ;internal clk
        MOVWF T0CON       ;load T0CON reg
        MOVLW  0x00       ;TMR0H = 00H, the high byte
        MOVWF  TMR0H      ;load Timer0 high byte
        MOVLW 0x00        ;TMR0L = 0, the low byte
        MOVWF TMR0L       ;load Timer0 low byte
        BCF INTCON,TMR0IF ;clear timer interrupt flag bit
        MOVLW 0x6         ;Timer1, no prescale,
                          ;ext. clock
        MOVWF T1CON       ;load T1CON reg
        MOVLW D'255'      ;TMR1H = 255
        MOVWF TMR1H       ;load Timer1 high byte
        MOVLW -D'200'     ;TMR1L = 0
        MOVWF TMR1L       ;load Timer1 low byte
        BCF PIR1,TMR1IF ;clear timer interrupt flag bit
        BSF T0CON,TMR0ON      ;start Timer0
        BSF T1CON,TMR1ON      ;start Timer1
        BSF INTCON,TMR0IE     ;enable Timer0 interrupt
        BSF PIE1,TMR1IE       ;enable Timer1 interrupt
        BSF INTCON,PEIE ;enable peripheral interrupts
        BSF INTCON,GIE  ;enable interrupts globally
OVER    BRA OVER          ;stay in this loop forever
;--------------------------ISR for Timer0
T0_ISR
        ORG 200H
        INCF PORTD        ;increment PORTD
        MOVLW 0x00        ;TMR0L = 0, the low byte
        MOVWF TMR0L       ;load Timer0 low byte
        MOVLW  0x00       ;TMR0H = 00, the high byte
        MOVWF  TMR0H      ;load Timer0 high byte
        BCF INTCON,TMR0IF ;clear timer interrupt flag bit
        GOTO CHK_INT
;--------------------------ISR for Timer2
T1_ISR
        ORG 300H
        BTG PORTB,6       ;toggle PORTC.6
        MOVLW D'255'      ;TMR1H = 255
        MOVWF TMR1H       ;load Timer1 high byte
        MOVLW -D'200'     ;TMR1L = 0
        MOVWF TMR1L       ;load Timer1 low byte
        BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
        GOTO CHK_INT
        END
```

Notice in Programs 11-2 and 11-3 that we use the "GOTO CHK_INT" instruction instead of RETFIE as the last instruction of the ISR. This is because we are checking for activation of multiple interrupts.

PIC18 interrupt programming in C using C18 compiler

In Chapter 7, we discussed how the C18 compiler uses "#pragma code" to place code at a specific ROM address. Because the C18 does not place an ISR at the interrupt vector table automatically, we must use Assembly language instruction GOTO at the interrupt vector to transfer control to the ISR. This is done as follows:

```
#pragma code high_vector =0x0008 // High-priority interrupt location
void My_HiVect_Int (void)
{
_asm
GOTO my_isr
_endasm
}
#pragma code                     // End of code
```

Now we redirect it from address location 00008 to another program to find the source of the interrupt and finally to the ISR. This is done with the help of the keyword **interrupt** as follows:

```
#pragma interrupt my_isr   //interrupt is reserved keyword
void my_isr (void)         //used for high-priority interrupt
{

//C18 places RETFIE here automatically due to
//interrupt keyword
}
```

Note that "pragma", "code", and "interrupts" are reserved keywords while the choice of all other labels is up to us. Examine Programs 11-2C and 11-3C. They are the C versions of Programs 11-2 and 11-3.

Program 11-2C uses Timer0 and Timer1 interrupts to generate square waves on pins RB1 and RB7, respectively, while data is being transferred from PORTC to PORTD. This is a C version of Program 11-2.

```
//Program 11-2C (C version of Program 11-2)
#include <p18F458.h>
#define myPB1bit PORTBbits.RB1
#define myPB7bit PORTBbits.RB7

void T0_ISR(void);
void T1_ISR(void);

#pragma interrupt chk_isr  //used for high-priority
                           //interrupt only
void chk_isr (void)
{
```

```
      if (INTCONbits.TMR0IF==1)  //Timer0 causes interrupt?
           T0_ISR();             //Yes. Execute Timer0 ISR
      if(PIR1bits.TMR1IF==1)     //Or was it Timer1?
           T1_ISR();             // Yes. Execute Timer1 ISR
}

#pragma code My_HiPrio_Int=0x08//high-priority interrupt
void My_HiPrio_Int (void)
{
  _asm
    GOTO chk_isr
  _endasm
}
#pragma code
void main(void)
  {
    TRISBbits.TRISB1=0;    //RB1 = OUTPUT
    TRISBbits.TRISB7=0;    //RB7 = OUTPUT
    TRISC = 255;           //PORTC = INPUT
    TRISD = 0;        //PORTD = OUTPUT
    T0CON=0x0;        //Timer 0, 16-bit mode, no prescaler
    TMR0H=0x35;       //load TH0
    TMR0L=0x00;       //load TL0
    T1CON=0x88;       //Timer 1, 16-bit mode, no prescaler
    TMR1H=0x35;       //load TH1
    TMR1L=0x00;       //load TL1
    INTCONbits.TMR0IF=0;   //clear TF0
    PIR1bits.TMR1IF=0;     //clear TF1
    INTCONbits.TMR0IE=1;   //enable Timer0 interrupt
    INTCONbits.TMR0IE=1;   //enable Timer1 interrupt
    T0CONbits.TMR0ON=1;    //turn on Timer0
    T1CONbits.TMR1ON=1;    //turn on Timer1
    INTCONbits.PEIE=1;//enable all peripheral interrupts
    INTCONbits.GIE=1; //enable all interrupts globally
   while(1)     //keep looping until interrupt comes
    {
      PORTD=PORTC;   //send data from PORTC to PORTD
    }
  }

void T0_ISR(void)
  {
    myPB1bit=~myPB1bit;    //toggle PORTB.1
    TMR0H=0x35;            //load TH0
    TMR0L=0x00;            //load TL0
    INTCONbits.TMR0IF=0;   //clear TF0
  }

void T1_ISR(void)
  {
    myPB7bit=~myPB7bit;    //toggle PORTB.7
```

```
    TMR1H=0x35;              //load TH0
    TMR1L=0x00;              //load TL0
    PIR1bits.TMR1IF=0;       //clear TF1
  }
```

Program 11-3C shows the C version of Program 11-3.

Program 11-3C has two interrupts: (1) PORTC counts up every time Timer0 overflows. It uses the 16-bit mode of Timer0 with the largest prescale possible; (2) a 1-Hz pulse is fed into Timer1 where Timer1 is used as a counter and counts up. Whenever the count reaches 200, it will toggle pin RB6.

```
//Program 11-3C
#include <p18F458.h>
#define myPB6bit PORTBbits.RB6

void chk_isr(void);
void T0_ISR(void);
void T1_ISR(void);
#pragma interrupt chk_isr //for high-priority interrupt only
void chk_isr (void)
{
if (INTCONbits.TMR0IF==1)  //Timer0 causes interrupt?
T0_ISR( );                 //Yes. Execute Timer0 program
     if(PIR1bits.TMR1IF==1)//Or was it Timer2?
     T1_ISR();             //Yes. Execute Timer2 program
     }

#pragma code My_HiPrio_Int=0x0008 //high-priority interrupt
void My_HiPrio_Int (void)
{
_asm
GOTO chk_isr
_endasm
}
#pragma code

void main(void)
  {
     TRISBbits.TRISB6=0;   //RB6 = OUTPUT
     TRISCbits.TRISC0=1;   //PORTC0 = INPUT
     TRISD=0;
     T0CON=0x08;           //Timer0, 16-bit mode,
                           //no prescaler
     TMR0H=0;              //load Timer0 high byte
     TMR0L=0;              //load Timer0 low byte
     T1CON=0x06;           //Timer 2, no prescaler
     TMR1H=255;            //load Timer1 high byte
     TMR1L=-200;           //load Timer1 low byte
     INTCONbits.TMR0IF=0;  //clear TF0
     PIR1bits.TMR1IF=0;    //clear TF1
```

```
    INTCONbits.TMR0IE=1;  //enable Timer0 interrupt
    PIE1bits.TMR1IE=1;    //enable Timer1 interrupt
    T0CONbits.TMR0ON=1;   //turn on Timer0
    T1CONbits.TMR1ON=1;   //turn on Timer1
    INTCONbits.PEIE=1;//enable all peripheral interrupts
    INTCONbits.GIE=1;     //enable all interrupts globally
    while(1);  //keep looping until interrupt comes
  }

void T0_ISR(void)
  {
    PORTD++;              //count up PORTD
    TMR0H=0;              //load Timer0 high byte
    TMR0L=0;              //load Timer0 low byte
    INTCONbits.TMR0IF=0;  //clear TF0
  }

void T1_ISR(void)
  {
    myPB6bit=~myPB6bit;   //toggle PB.6
    TMR1H=255;            //load Timer1 high byte
    TMR1L=-200;           //load Timer1 low byte
    PIR1bits.TMR2IF=0;    //clear TF1
  }
```

Review Questions

1. True or false. A unique address in the interrupt vector table is assigned to each of Timer0–Timer3.
2. Upon power-on reset, what address in the interrupt vector table is assigned to the high-priority interrupt?
3. Which register does TMR1IE belong to? Show how it is enabled.
4. Assume that Timer1 is programmed in 8-bit mode, TMR1L = F5H, and the TMR1IF bit is enabled. Explain how the interrupt for the timer works.
5. True or false. The last two instructions of the ISR for Timer0 are:

```
BCF   INTCON,TMR0IF
RETFIE
```

SECTION 11.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

The PIC18 has three external hardware interrupts. Pins RB0 (PORTB.0), RB1 (PORTB.1), and RB2 (PORTB.2), designated as INT0, INT1, and INT2 respectively, are used as external hardware interrupts. Upon activation of these pins, the PIC18 gets interrupted in whatever it is doing and jumps to the vector table to perform the interrupt service routine. In this section we study these three external hardware interrupts of the PIC18 with some examples in both Assembly and C.

External interrupts INT0, INT1, and INT2

There are three external hardware interrupts in the PIC18: INT0, INT1, and INT2. They are located on pins RB0, RB1, and RB2, respectively. See Figures 11-7 and 11-8. On default, all three hardware interrupts are directed to vector table location 0008H, unless we specify otherwise. They must be enabled before they can take effect. This is done using the INTxIE bit. The registers associated with INTxIE bits are shown in Table 11-3. For example, the instruction "`BSF INTCON,INT0IE`" enables INT0. The INT0 is a *positive-edge-triggered interrupt*, which means, when a low-to-high signal is applied to pin RB0 (PORTB.0), the INT0IF is raised, causing the controller to be interrupted. The raising of INT0IF forces the PIC18 to jump to location 0008H in the vector table to service the ISR. In Table 11-3, notice the INTxIF bits and the registers they belong to. Upon power-on reset, the PIC18 makes INT0, INT1, and INT2 rising (positive) edge-triggered interrupts. To make them falling (negative) edge-triggered interrupts, we must program the INTEDGx bits, as we will see shortly.

Examine Program 11-4 and its C version, Program 11-4C, to gain insight into external hardware interrupts.

Table 11-3: Hardware Interrupt Flag Bits and Associated Registers

Interrupt (Pin)	Flag bit	Register	Enable bit	Register
INT0 (RB0)	INT0IF	INTCON	INT0IE	INTCON
INT1 (RB1)	INT1IF	INTCON3	INT1IE	INTCON3
INT2 (RB2)	INT2IF	INTCON3	INT2IE	INTCON3

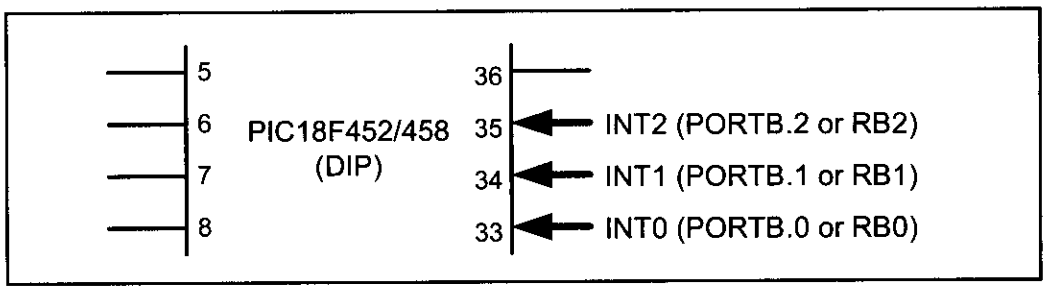


Figure 11-7. PIC18 External Hardware Interrupt Pins

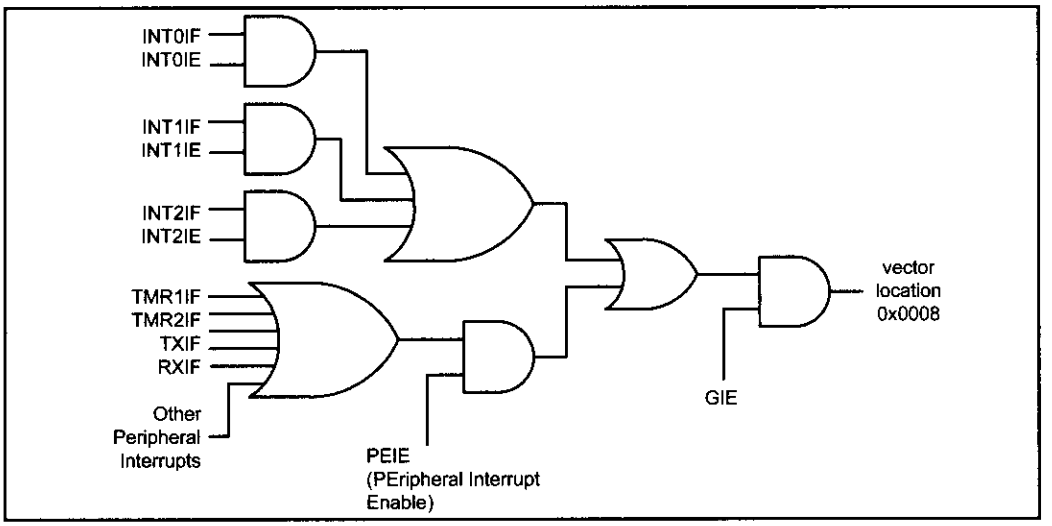


Figure 11-8. INT0–INT2 Hardware Interrupts

Program 11-4 connects a switch to INT0 and an LED to pin RB7. In this program, every time INT0 is activated, it toggles the LED, while at the same time data is being transferred from PORTC to PORTD.

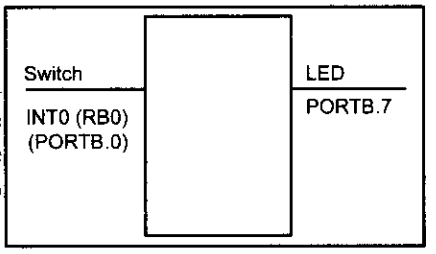


Figure 11-9. For Program 11-4

```
;Program 11-4
      ORG   0000H
      GOTO  MAIN              ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
      ORG   0008H             ;interrupt vector table
      BTFSS INTCON,INT0IF     ;Did we get here due to INT0?
      RETFIE                  ;No. Then return to main
      GOTO  INT0_ISR          ;Yes. Then go INT0 ISR
;;--the main program for initialization
      ORG   00100H
MAIN  BCF   TRISB,7           ;PB7 as an output
      BSF TRISB,INT0          ;make INT0 an input pin
      CLRF TRISD              ;make PORTD output
      SETF TRISC              ;make PORTC input
      BSF INTCON,INT0IE       ;enable INT0 interrupt
      BSF INTCON,GIE          ;enable interrupts globally
OVER  MOVFF PORTC,PORTD       ;send data from PORTC to PORTD
      BRA OVER                ;stay in this loop forever
;---------------------------ISR for INT0
INT0_ISR
      ORG 200H
      BTG PORTB,7             ;toggle PB7
      BCF INTCON,INT0IF       ;clear INT0 interrupt flag bit
      RETFIE                  ;return from ISR
      END
```

Look at Program 11-4. When a rising edge of the signal is applied to pin INT0, the LED will toggle. In this example, to toggle the LED again, the INT0 pulse must be brought back LOW and then forced HIGH to create a rising edge to activate the interrupt.

```
//Program 11-4C (This is the C version of Program 11-4)
      #include <p18F4580.h>
      #define mybit PORTBbits.RB7
      void chk_isr(void);
      void INT0_ISR(void);
      #pragma interrupt chk_isr//used for high-priority int
      void chk_isr (void)
      {
      if (INTCONbits.INT0IF==1)   //INT0 caused interrupt?
      INT0_ISR( );                //Yes. Execute INT0 program
      }
      #pragma code My_HiPrio_Int=0x08  //high-priority
                                  //interrupt location
      void My_HiPrio_Int (void)
      {
      _asm
      GOTO chk_isr
      _endasm
      }
      #pragma code
      void main(void)
        {
           TRISBbits.TRISB7=0;    //RB7 = OUTPUT
           TRISBbits.TRISB0=1;    //INT0 = INPUT
           TRISC = 0xFF;          //PORTC = INPUT
           TRISD = 0;             //PORTD = OUTPUT
           INTCONbits.INT0IF=0;   //clear TF1
           INTCONbits.INT0IE=1;   //enable Timer0 interrupt
           INTCONbits.GIE=1;      //enable all interrupts
         while(1)      //keep looping until interrupt comes
          {
           PORTD=PORTC;
          }
        }
      void INT0_ISR(void)
        {
          mybit=~mybit;
          INTCONbits.INT0IF=0;          //clear INT0 flag
        }
```

Negative edge-triggered interrupts

Upon power-on reset, the PIC18 makes INT0, INT1, and INT2 positive (rising) edge-triggered interrupts. To make any of them a negative (falling) edge-triggered interrupt, we must program the corresponding bit called INTEDGx, where x can be 0, 1, or 2. The INTCON2 register holds, among other bits, the INTEDG0, INTEDG1, and INTEDG flag bits as shown in Figure 11-10. INTEDG0, INTEDG1, and INTEDG2 are bits D4, D5, and D6 of the INTCON2 register, respectively, as shown in Figure 11-10. The status of these bits determines the negative or positive edge-triggered mode of the hardware interrupts. Upon reset, INTEDGx bits are all 1s, meaning that the external hardware interrupts are positive edge-triggered. By making the INTEDG0 bit LOW, the external hardware interrupts of INT0 become *negative edge-triggered interrupts*. For example, the instruction "`BSF INTCON2,INTEDG1`" makes INTEDG1 a negative edge-triggered interrupt, in which, when a high-to-low signal is applied to pin RB1 (PORTB.1), the controller will be interrupted and forced to jump to location 0008H in the vector table to service the ISR (assuming that the GIE and INT0IE bits are enabled). This is shown in Program 11-5. Its C version is shown in Program 11-5C.

	INTEDG0	INTEDG1	INTEDG2				

INTEDGx External Hardware Interrupt Edge trigger bit
0 = Interrupt on negative (falling) edge
1 = Interrupt on positive (rising) edge (Default for power-on reset)

Figure 11-10. INTCON2 Register INTEDG Allows Positive or Negative Edge Trigger

In Program 11-5 we assume that pin RB1 (INT1) is connected to a pulse generator and the pin RB7 is connected to an LED. The program will toggle the LED on the falling edge of the pulse. In other words, the LED is turned on and off at the same rate as the pulses are applied to the INT1 pin.

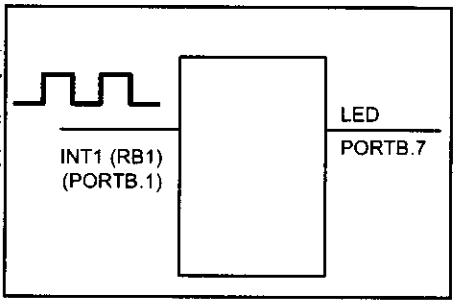


Figure 11-11. For Program 11-5

```
;Program 11-5
          ORG   0000H
          GOTO  MAIN          ;bypass interrupt vector table
     ;--on default all interrupts go to to address 00008
          ORG   0008H         ;interrupt vector table
          BTFSS INTCON3,INT1IF  ;Did we get here due to
                                ;INT1 interrupt?
          RETFIE                ;No. Then return to main
          GOTO  INT1_ISR        ;Yes. Then go INT1 ISR
     ;--the main program for initialization
          ORG   00100H
     MAIN BCF   TRISB,7       ;PB7 as an output
          BSF   TRISB,INT1    ;make INT1 an input pin
```

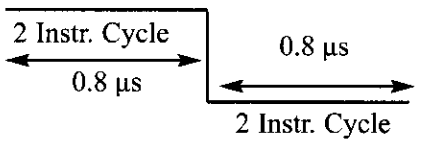
```
        BSF  INTCON3,INT1IE   ;enable INT1 interrupt
        BCF INTCON2,INTEDG1   ;make it negative
                              ;edge-triggered
        BSF INTCON,GIE        ;enable interrupts globally
OVER    BRA OVER              ;stay in this loop forever
;--------------------------ISR for INT1
INT1_ISR
        ORG 200H
        BTG PORTB,7           ;toggle on RB7
        BCF INTCON3,INT1IF ;clear INT1 interrupt flag bit
        RETFIE
        END

//Program 11-5C (This is the C version of Program 11-5)
      #include <p18F4580.h>
      #define mybit PORTBbits.RB7
      void chk_isr(void);
      void INT1_ISR(void);
      #pragma code My_HiPrio_Int =0x0008    //high-priority
interrupt location
      void My_HiPrio_Int (void)
      {
      _asm
      GOTO chk_isr
      _endasm
      }
      #pragma code
      #pragma interrupt chk_isr  //used for high-priority
                                 //interrupt only
      void chk_isr (void)
      {
      if (INTCON3bits.INT1IF==1) //INT1 causes interrupt?
      INT1_ISR( );     //Yes. Execute INT1 program
           }
      void main(void)
        {
           TRISBbits.TRISB7=0;   //RB7 = OUTPUT
           TRISBbits.TRISB1=1;   //INT1 = INPUT
           INTCON3bits.INT1IF=0; //clear INT1
           INTCON3bits.INT1IE=1; //enable INT1 interrupt
           INTCON2bits.INTEDG1=0;//make it negative edge
           INTCONbits.GIE=1;     //enable all interrupts
           while(1);  //keep looping until interrupt comes
        }

      void INT1_ISR(void)
        {
          mybit=~mybit;
          INTCON3bits.INT1IF=0;        //clear INT1 flag
        }
```

Sampling the edge-triggered interrupt

Before ending this section, we need to answer the question of how often the edge-triggered interrupt is sampled. In edge-triggered interrupts, the external source must be held HIGH for at least two instruction cycles, and then held LOW for at least two instruction cycles to ensure that the transition is seen by the microcontroller.

The rising edge (or the falling edge) is latched by the PIC18 and is held by the INTxIF bits. The INT0IF, INT1IF, and INTIF2 bits hold the latched rising (or falling, depending on the INTEDGx bit) edge of pins RB0–RB2. The INT0IF–INT2IF bits function as interrupt-in-service flags. When an interrupt-in-service flag is raised, it indicates to the external world that the interrupt is being serviced and no new interrupt on this INT*n* pin will be responded to until this service is finished. This is just like the busy signal you get when calling a telephone number that is in use. Regarding the INT0IF–INT2IF one more point must be emphasized. The point is that before the ISRs are finished (that is, before execution of instruction RETFIE), these bits (INT0IF–INT2IF) must be cleared, indicating that the interrupt is finished and the PIC18 is ready to respond to another interrupt on that pin. For another interrupt to be recognized, the pin must go back to a logic LOW state and be brought back HIGH to be considered a positive edge-triggered interrupt.

Minimum pulse duration to detect edge triggered interrupts = 2 instruction cycles
For XTAL = 10 MHz, we have an instruction cycle time of 400 ns = 0.4 μs



Review Questions

1. True or false. Upon reset, all external hardware interrupts INT0–INT2 go to the interrupt vector table address of 0008.
2. For PIC18F458, what pins are assigned to INT0–INT2?
3. Show how to enable the INT1.
4. Assume that the INT0IE bit for the external hardware interrupt INT0 is enabled. Explain how this interrupt works when it is activated.
5. True or false. Upon reset, the external hardware interrupt is negative edge-triggered.
6. How do we make sure that a single interrupt is not recognized as multiple interrupts?
7. True or false. The last two instructions of the ISR for INT0 are:

```
BCF   INTCON2,INT0IF
RETFIE
```

SECTION 11.4: PROGRAMMING THE SERIAL COMMUNICATION INTERRUPTS

In Chapter 10 we studied the serial communication of the PIC18. All examples in that chapter used the polling method. In this section we explore interrupt-based serial communication, which allows the PIC18 to do many things, in addition to sending and receiving data from the serial communication port.

RCIF and TXIF flags and interrupts

As you may recall from Chapter 10, TXIF (transfer interrupt) is raised when the last bit of the framed data, the stop bit, is transferred, indicating that the TXREG register is ready to transfer the next byte. RCIF (received interrupt) is raised when the entire frame of data, including the stop bit, is received. In other words, when the RCREG register has a byte, RCIF is raised to indicate that the received byte needs to be picked up before it is lost (overrun) by new incoming serial data. As far as serial communication is concerned, all the above concepts apply equally when using either polling or an interrupt. The only difference is in how the serial communication needs are served. In the polling method, we wait for the flag (TXIF or RCIF) to be raised; while we wait we cannot do anything else. In the interrupt method, we are notified when the PIC18 has received a byte, or is ready to send the next byte; we can do other things while the serial communication needs are served.

In the PIC18 two interrupts are set aside for serial communication. One interrupt is used for send and the other for receive. If the corresponding interrupt bit of TXIE or RCIE is enabled, when TXIF or RCIF is raised the PIC18 gets interrupted and jumps to memory address location 0008H to execute the ISR.

Table 11-4: Serial Port Interrupt Flag Bits and their Associated Registers

Interrupt	Flag bit	Register	Enable bit	Register
TXIF (Transmit)	TXIF	PIR1	TXIE	PIE1
RCIF (Receive)	RCIF	PIR1	RCIE	PIE1

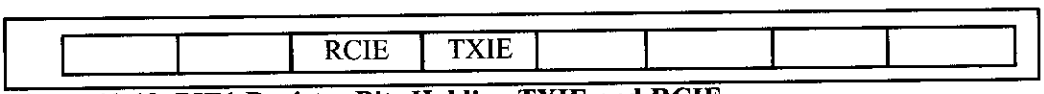


Figure 11-12. PIE1 Register Bits Holding TXIE and RCIE

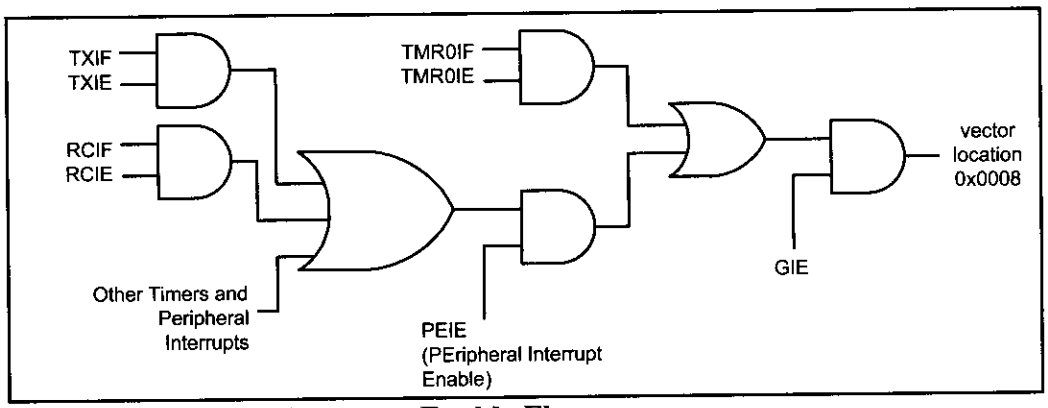


Figure 11-13: Serial Interrupt Enable Flags

Use of serial COM in the PIC18

In the vast majority of applications, the serial interrupt is used mainly for receiving data and is seldom used for sending data serially. This is like receiving a telephone call, where we need a ring to be notified of an incoming call. If we need to make a phone call there are other ways to remind ourselves and so no need for ringing. In receiving the phone call, however, we must respond immediately no matter what we are doing or we will miss the call. Similarly, we use the serial interrupt to receive incoming data so that it is not lost. Look at Program 11-6. Notice that the last instruction of the ISR is RETFIE and there is no clearing of the TXIF flag, since it is done by writing a byte to TXREG.

In Figure 11-13, notice the role of PEIE (PEripheral Interrupt Enable) in allowing serial communication interrupts and other interrupts to come in. This is in addition to the GIE bit discussed in Section 11-1.

For Program 11-6 we assume an 8-bit switch is connected to PORTD. In this program, the PIC18 reads data from PORTD and writes it to TXREG continuously to be transmitted serially. We assume that XTAL = 10 MHz. The baud rate is set at 9600.

```
;Program 11-6
          ORG   0000H
          GOTO  MAIN          ;bypass interrupt vector table
     ;--on default all interrupts go to to address 00008
          ORG   0008H         ;interrupt vector table
          BTFSC PIR1,TXIF     ;Is interrupt due to transmit?
          BRA TX_ISR          ;Yes. Then go to ISR
          RETFIE              ;No. Then return
          ORG 0040H
     TX_ISR                   ;service routine for TXIF
          MOVWFF PORTD,TXREG;load new value, clear TXIF
          RETFIE              ;then return to main
     ;--the main program for initialization
          ORG   00100H
     MAIN SETF TRISD          ;make PORTD input
          MOVLW 0x20          ;enable transmit and choose
                              ;low baud
          MOVWF TXSTA         ;write to reg
          MOVLW D'15'         ;9600 bps
                              ;(Fosc / (64 * Speed) - 1)
          MOVWF SPBRG         ;write to reg
          BCF TRISC, TX       ;make TX pin of PORTC an
                              ;output pin
          BSF RCSTA, SPEN     ;enable the serial port
          BSF PIE1,TXIE       ;enable TX interrupt
          BSF INTCON,PEIE     ;enable peripheral interrupts
          BSF INTCON,GIE      ;enable interrupts globally
     OVER BRA OVER            ;stay in this loop forever
          END
```

Program 11-7 is a modification of Program 11-6 with receive interrupt. In this program, the PIC18 gets data from PORTD and sends it to TXREG continuously while incoming data from the serial port is sent to PORTB. We assume that XTAL = 10 MHz and the baud rate = 9600. This program can be verified by connecting your PICTrainer to the serial port of the x86 IBM PC and using HyperTerminal to send and receive data between the PIC Trainer and the IBM PC.

```
;Program 11-7
      ORG   0000H
      GOTO  MAIN              ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
      ORG   0008H             ;interrupt vector table
HI_ISR BTFSC PIR1,TXIF        ;is it TX interrupt?
      BRA   TX_ISR            ;Yes. Then branch to TX_ISR
      BTFSC PIR1,RCIF         ;Is it RC interrupt?
      BRA   RC_ISR            ;Yes. Then branch to RC_ISR
      RETFIE                  ;No. Then return to main
TX_ISR MOVFF PORTD,TXREG      ;loading TXREG clears TXIF
      GOTO HI_ISR
RC_ISR
      MOVFF RCREG,PORTB       ;copy received data to PORTB
      GOTO HI_ISR
;--the main program for initialization
      ORG   00100H
MAIN  CLRF TRISB        ;PORTB as an output
      SETF TRISD        ;make PORTD input
      MOVLW 0x20        ;enable transmit and choose low baud
      MOVWF TXSTA       ;write to reg
      MOVLW D'15'       ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG       ;write to reg
      BCF TRISC,TX      ;make TX pin of PORTC an output pin
      BSF TRISC,RX      ;make RCV pin of PORTC an input pin
      MOVLW 0x90        ;enable receive and serial port
      MOVWF RCSTA       ;write to reg
      BSF PIE1,TXIE     ;enable TX interrupt
      BSF PIE1,RCIE     ;enable receive interrupt
      BSF INTCON,PEIE   ;enable peripheral interrupts
      BSF INTCON,GIE    ;enable interrupts globally
OVER  BRA OVER          ;stay in this loop forever
      END
```

```
//Program 11-7C (This is the C version of Program 11-7)
#include <p18F458.h>
void chk_isr(void);
void TX_ISR(void);
void RC_ISR(void);
#pragma code My_HiPrio_Int=0x08 //high-priority interrupt
void My_HiPrio_Int (void)
{
      _asm
            GOTO chk_isr
      _endasm
```

```
}
#pragma code
#pragma interrupt chk_isr//used for high-priority interrupt
void chk_isr (void)
{
if (PIR1bits.TXIF==1)        //Transmit caused interrupt?
  TX_ISR( );                 //Yes. Execute Transmit program
if (PIR1bits.RCIF==1)        //Receive caused interrupt?
  RC_ISR( );                 //Yes. Execute Receive program
}
void main(void)
  {
     TRISD = 0xFF;           //PORTD = INPUT
     TRISB = 0;              //PORTB = OUTPUT
     TRISCbits.TRISC6=0;     //TX pin = OUTPUT
     TRISCbits.TRISC7=1;     //RCV pin = INPUT
     TXSTA=0x20;             //choose low baud rate, 8-bit
     SPBRG=15;               //9600 baud rate/ XTAL = 10 MHz
     RCSTAbits.CREN=1;
     RCSTAbits.SPEN=1;
     TXSTAbits.TXEN=1;
     PIE1bits.RCIE=1;        //enable RCV interrupt
     PIE1bits.TXIE=1;        //enable TX interrupt
     INTCONbits.PEIE=1;      //enable peripheral interrupts
     INTCONbits.GIE=1;       //enable all interrupts globally
     while(1);          //keep looping until interrupt comes
  }
void TX_ISR(void)
  {
     TXREG=PORTD;
  }
void RC_ISR(void)
  {
     PORTB=RCREG;
  }
  }
```

Review Questions

1. True or false. All interrupts, including the TXIF and RXIF, are directed to a single location in the interrupt vector table.
2. What address in the interrupt vector table is assigned to the serial interrupt?
3. Which register do the TXIF and RXIF flags belong to? Show how they are enabled.
4. Assume that the RCIF bit is enabled. Explain how this interrupt gets activated and its actions upon activation.
5. True or false. Upon reset, the serial interrupts are active and ready to go.
6. True or false. The last two instructions of the ISR for the receive interrupt are:

```
BCF  RIR1,RCIF
RETFIE
```

7. Answer Question 6 for the transmit interrupt.

SECTION 11.5: PORTB-CHANGE INTERRUPT

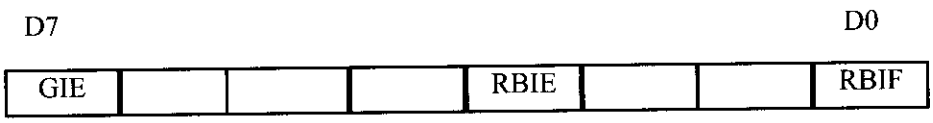
The four pins of the PORTB (RB4–RB7) can cause an interrupt when any changes are detected on any one of them. They are referred to as "PORTB-Change interrupt" to distinguish them from the INT0–INT2 interrupts, which are also located on PORTB (RB0–RB2). See Figure 11-15. The PORTB-Change interrupt has a single interrupt flag called RBIF and is located in the INTCON register. This is shown in Figure 11-14. In Figure 11-14, also notice the RBIE bit for enabling the PORTB-Change interrupt. In Section 11.3 we discussed the external hardware interrupts of INT0, INT1, and INT2. Notice the following differences between the PORTB-Change interrupt and INT0–INT2 interrupts:

(a) Each of the INT0–INT2 interrupts has its own pin and is independent of the others. These interrupts use pins PORTB.0 (RB0), PORTB.1 (RB1), and PORTB.2 (RB2), respectively. The PORTB-change interrupt uses all four of the PORTB pins RB4–PB7 and is considered to be a single interrupt even though it can use up to four pins.

(b) While each of the INT0–INT2 interrupts has its own flag, and is independent of the others, there is only a single flag for the PORTB-Change interrupt.

(c) While each of the INT0–INT2 interrupts can be programmed to trigger on the negative or positive edge, the PORTB-Change interrupt causes an interrupt if any of its pins changes status from HIGH to LOW, or LOW to HIGH. See Figure 11-16.

PORTB-Change is widely used in keypad interfacing as we will see in Chapter 12. Another way to use the PORTB-Change interrupt is shown in Program 11-8. In that program, we assume a door sensor is connected to pin RB4 and upon opening or closing the door, the buzzer will sound. See Figure 11-17.



GIE (Global Interrupt Enable)
GIE = 0 Disables all interrupts. If GIE = 0, no interrupt is acknowledged, even if they are enabled individually.
If GIE = 1, interrupts are allowed to happen. Each interrupt source is enabled by setting the corresponding interrupt enable bit.

RBIE PORTB-Change Interrupt Enable
= 0 Disables PORTB-Change interrupt
= 1 Enables PORTB-Change interrupt

RBIF PORTB-Change Interrupt Flag.
= 0 None of the RB4–RB7 pins have changed state
= 1 At least one of the RB4–RB7 pins have changed state

The RBIE bit, along with the GIE, must be set high for any changes on the pins RB4–RB7 to cause an interrupt. The RB4–RB7 pins must also have been configured as input pins for this interrupt to work. In order to clear the RBIF flag we must read the pins of RB4–RB7 **and** use the instruction "BCF INTCON,RBIF".

Figure 11-14. INTCON (Interrupt Control) Register

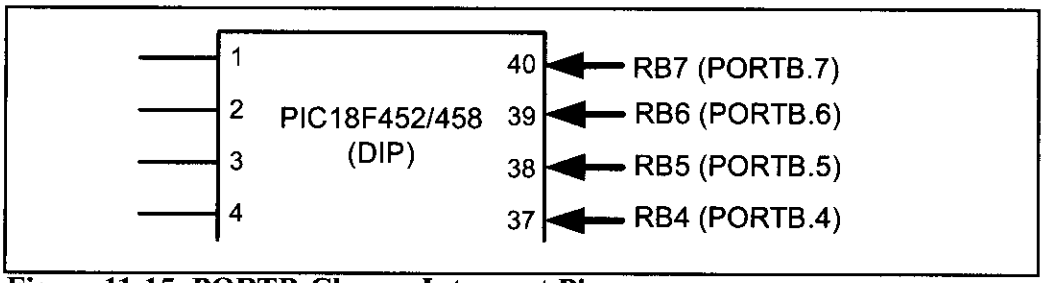


Figure 11-15. PORTB-Change Interrupt Pins

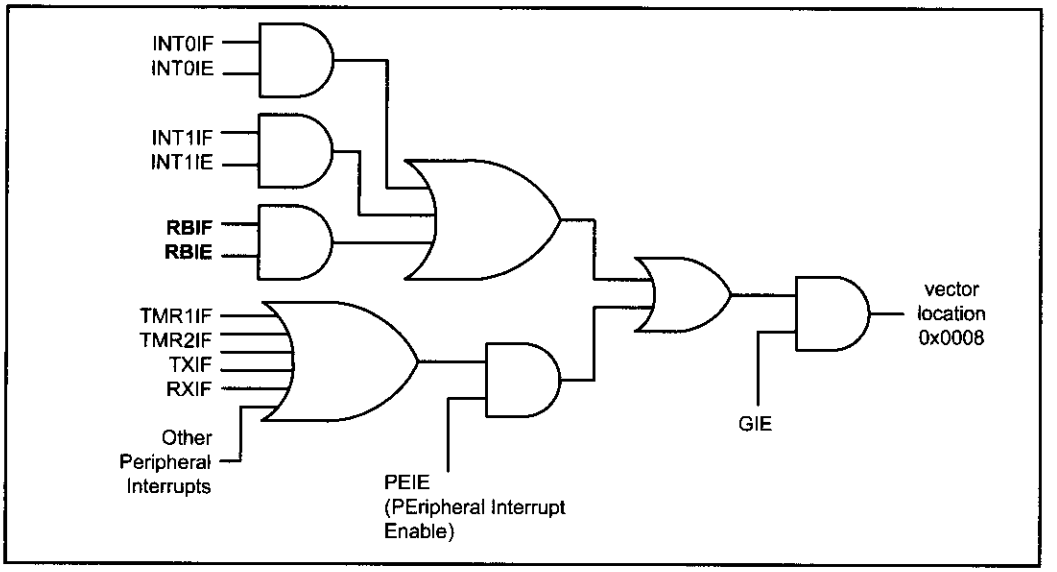


Figure 11-16. PORTB-Change Interrupt (RBIF)

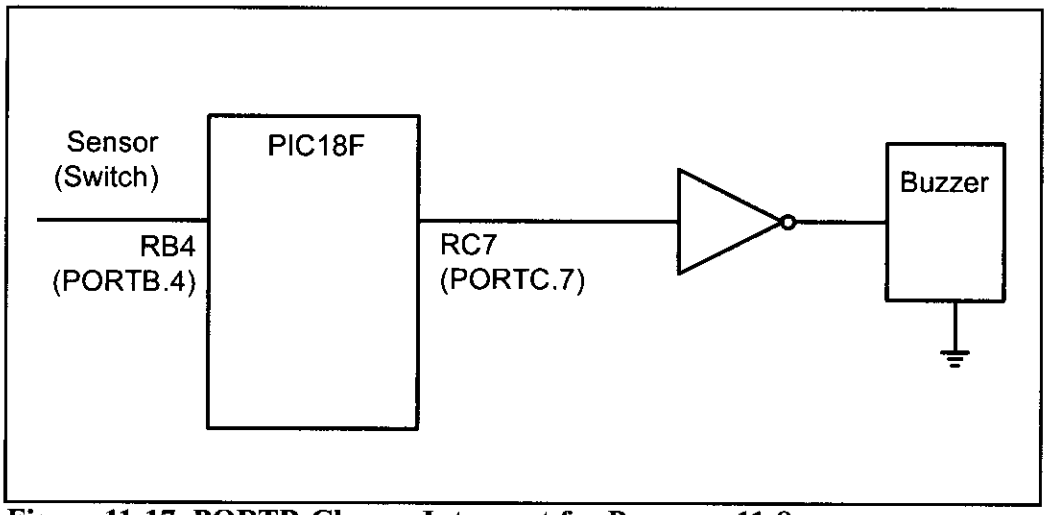


Figure 11-17. PORTB-Change Interrupt for Program 11-8

For Program 11-8 we have connected a door sensor to pin RB4 and a buzzer to pin RC7. In this program, every time the door is opened, it sounds the buzzer by sending it a square wave frequency.

```
;Program 11-8
MYREG EQU   0x20        ;set aside a couple of registers
DELRG EQU 0x80          ;for buzzer time delay
      ORG   0000H
      GOTO  MAIN        ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
      ORG   0008H             ;interrupt vector table
      BTFSS INTCON,RBIF       ;Did we get here due to RBIF?
      RETFIE                  ;No. Then return to main
      GOTO  PB_ISR            ;Yes. Then go ISR
;--the main program for initialization
      ORG   00100H
MAIN  BCF   TRISC,7     ;PORTC.7 as an output for buzzer
      BSF   TRISB,4     ;PORTB.4 as an input for interrupt
      BSF INTCON,RBIE ;enable PORTB-Change interrupt
      BSF INTCON,GIE  ;enable interrupts globally
OVER  BRA OVER          ;stay in this loop forever
;-------------------------ISR for PORTB-Change INT
PB_ISR
      ORG 200H
      MOVF  PORTB,W           ;we must read PORTB
      MOVLW D'250'            ;for delay
      MOVWF MYREG
BUZZ  BTG PORTC,7             ;toggle PC7 for the buzzer
      MOVLW D'255'            ;for delay
      MOVWF DELRG
DELAY DECF DELRG,F
      BNZ DELAY               ;keep sounding the buzzer
      DECF MYREG,F
      BNZ BUZZ
      BCF INTCON,RBIF ;and clear RBIF interrupt flag bit
      RETFIE
      END
```

Notice for the PORTB-Change interrupt, there is no need to enable the PEIE; however, we still need to enable the GIE bit.

It must be noted again that, while the INT0–INT2 interrupts each have their own interrupt flags, there is only a single interrupt flag (RBIF) for all the four pins of the RB4–RB7. Examine Program 11-9. For this program, we assume that each of the pins RB4 and RB5 is connected to an external switch. Upon activation of the SW, an LED reflects the status. See Figure 11-18.

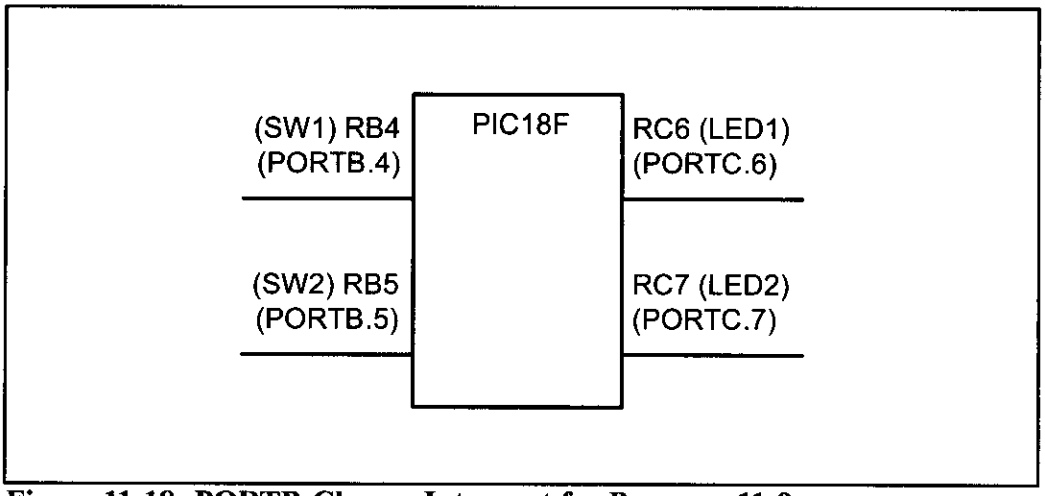


Figure 11-18: PORTB-Change Interrupt for Program 11-9

For Program 11-9 we have connected SW1 and SW2 to pins RB4 and RB5 respectively. In this program, the activation of SW1 and SW2 will result in changing the state of LED1 and LED2 respectively.

```
;Program 11-9
          ORG   0000H
          GOTO  MAIN  ;bypass interrupt vector table
     ;--on default all interrupts go to to address 00008
          ORG   0008H       ;interrupt vector table
          BTFSS INTCON,RBIF;Did we get here due to RBIF?
          RETFIE            ;No. Then return to main
          GOTO  PB_ISR      ;Yes. Then go ISR
     ;--the main program for initialization
          ORG   0100H
     MAIN BCF TRISC,4       ;PC4 as an output
          BCF TRISC,5       ;PC5 as an output
          BSF TRISB,4 ;PB4 as an input for the interrupt
          BSF TRISB,5 ;PB5 as an input for the interrupt
          BSF INTCON,RBIE ;enable PORTB interrupt
          BSF INTCON,GIE  ;enable interrupts globally
     OVER BRA OVER        ;stay in this loop forever
     ;--------------------------ISR for PORTB_Change
     PB_ISR
          ORG 200H
          MOVFF PORTB,W ;get the status of switches
          ANDLW 0x30 ;mask unneeded bits
          MOVFF W,PORTC ;update LEDs
          BCF INTCON,RBIF ;clear RBIF interrupt flag bit
          RETFIE
          END
```

```
//Program 11-9C (This is the C version of Program 11-9)
#include <p18F458.h>
#define LED1 PORTCbits.RC4
#define LED2 PORTCbits.RC5

void chk_isr(void);
void RBINT_ISR(void);

#pragma code My_HiPrio_Int =0x0008 //high-priority int
void My_HiPrio_Int (void)
{
     _asm
          GOTO chk_isr
     _endasm
}
#pragma code
#pragma interrupt chk_isr  //used for high-priority int
void chk_isr (void)
{
     if (INTCONbits.RBIF==1)     //RBIF caused interrupt?
     RBINT_ISR( );          //Yes. Execute ISR program
}
void main(void)
  {
     TRISCbits.TRISC4=0;   //RC4 = OUTPUT
     TRISCbits.TRISC5=0;   //RC5 = OUTPUT
     TRISBbits.TRISB4 = 1; //RB4 = INPUT for interrupt
     TRISBbits.TRISB5 = 1; //RB5 = INPUT for interrupt
     INTCONbits.RBIF=0;    //clear RBIF
     INTCONbits.RBIE=1;    //enable RB interrupt
     INTCONbits.GIE=1;     //enable all interrupts globally
     while(1);  //keep looping until interrupt comes
  }
void RBINT_ISR(void)
  {
     LED1=PORTBbits.RB4;
     LED2=PORTBbits.RB5;
     INTCONbits.RBIF=0;    //clear RBIF flag
  }
```

Review Questions

1. True or false. There is a single interrupt for each of the PORTB pins.
2. What address in the interrupt vector table is assigned to the PORTB-Change interrupt?
3. Which register do the RBIF and RBIE flags belong to? Show how RBIE is enabled.
4. Give the last two instructions of the ISR for the PORTB-Change interrupt.
5. True or false. Upon reset, the RBIF interrupt is active and ready to go.

SECTION 11.6: INTERRUPT PRIORITY IN THE PIC18

The next topic that we must deal with is what happens if two interrupts are activated at the same time? Which of these two interrupts is responded to first? Interrupt priority is the main topic of discussion in this section.

Setting interrupt priority

In the PIC18 microcontroller, there are only two levels of interrupt priority: (a) low level, and (b) high level. While address 0008 is assigned to high-priority interrupts, the low-priority interrupts are directed to address 00018 in the interrupt vector table. See Table 11-5. Upon power-on reset, all interrupts are automatically designated as high priority and will go to address 00008H. This is done to make the PIC18 compatible with the earlier generation of PIC microcontrollers such as PIC16xxx. We can make the PIC18 a two-level priority system by way of programming the IPEN (interrupt priority enable) bit in the RCON register. Figure 11-19 shows the IPEN bit of the RCON register. Upon power-on reset, the IPEN bit contains 0, making the PIC18 a single priority level chip, just like the PIC16xxx. To make the PIC18 a two-level priority system, we must first set the IPEN bit to HIGH. It is only after making IPEN = 1 that we can assign a low priority to any of the interrupts by programming the bits called IP (interrupt priority). Figure 11-20 shows IPR1 (interrupt priority register) with the IP bits for TXIP, RCIP, TMR1IP, and TMR2IP. If IPEN = 1, then the IP bit will take effect and will assign a given interrupt a low priority. As a result of assigning a low priority to a given interrupt, it will land at the address 0018 instead of 0008 in the interrupt vector table. The IP (interrupt priority) bit along with the IF (interrupt flag) and IE (interrupt enable) bits will complete all the flags needed to program the interrupts for the PIC18. Table 11-6 shows the three flags and the registers they belong to for some of the interrupts used in this chapter. In Table 11-6, notice the absence of the INT0 priority flag. The INT0 has only one priority and that is high priority. That means all the PIC18 interrupts can be assigned a low or high priority level, except the external hardware interrupt of INT0. Study Figures 11-22 through 11-25 very carefully. When examining these figures, the following point must be noted. By making IPEN = 1, we enable the interrupt priority feature. Now we must also enable two bits to enable the interrupts: (a) We must set GIEH = 1. The GIEH bit is part of the INTCON register (Figure 11-21) and is the same as GIE, which we have used in previous sections. In this regard there is no difference between the priority and no-priority systems. (b) The second bit we must set high is GIEL (part of INTCON). Making GIEL = 1 will enable all the interrupts whose IP = 0. That means all the interrupts that have been given the low priority will be forced to vector location 00018H.

Table 11-5: Interrupt Vector Table for the PIC18

Interrupt	ROM Location (Hex)
Power-on-Reset	0000
High-priority Interrupt	0008 (Default upon power-on reset)
Low-priority Interrupt	0018 (Selected with IP bit)

IPEN							

IPEN Interrupt Priority Enable bit
0 = All the interrupts are directed to the vector location 0008 (default).
1 = Interrupts can be assigned a low or high priority.

The importance of IPEN: Upon power-on reset, all the interrupts of PIC18 are directed to location 0008, making it a single-priority system, just like PIC16xxx. To prioritize the PIC18 interrupts into low- and high-level priorities, we must make IPEN = 1. When IPEN = 1, we can assign either a low or a high priority to any of the interrupts by manipulating the corresponding bit in the IPR (interrupt priority register) for that interrupt. When interrupt priority is enabled (IPEN = 1), we must set both the GIEH and GIEL bits to high in order to enable the interrupts globally. Notice in Figure 11-21 that GIE is the same as GIEH.

Figure 11-19. RCON Register. IPEN Allows Prioritizing the Interrupt into 2 Levels

		RCIP	TXIP			TMR2IP	TMR1IP

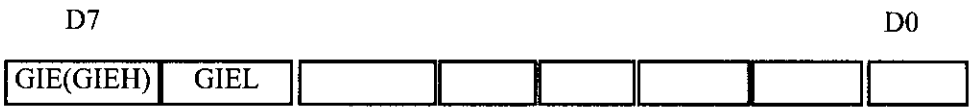
RCIP USART (Serial COM) Receive Interrupt Priority bit
0 = Low priority
1 = High priority
TXIP USART (Serial COM) Transmit Interrupt Priority bit
0 = Low priority
1 = High priority
TMR2IP Timer2 Interrupt Priority bit
0 = Low priority
1 = High priority
TMR1IP Timer1 Interrupt Priority bit
0 = Low priority
1 = High priority

Figure 11-20. IPR1 Peripheral Interrupt Priority Register 1

Table 11-6: Interrupt Flag Bits for PIC18 Timers

Interrupt	Flag bit (Register)	Enable bit (Register)	Priority (Register)
Timer0	TMR0IF (INTCON)	TMR0IE (INTCON)	TMR0IP (INTCON2)
Timer1	TMR1IF (PIR1)	TMR1IE (PIE1)	TMR1IP (IPR1)
Timer2	TMR2IF (PIR1)	TMR2IE (PIE1)	TMR2IP (IPR1)
Timer3	TMR3IF (PIR3)	TMR3IE (PIE2)	TMR3IP (IPR2)
INT1	INT1IF (PIR1)	INT1IE (PIE1)	INT1IP (INTCON3)
INT2	INT2IF (PIR1)	INT2IE (PIE1)	INT2IP (INTCON)
TXIF	TXIF (PIR1)	TXIE (PIE1)	TXIP (IPR1)
RCIF	RCIF (PIR1)	RCIE (PIE1)	RCIP (IPR1)
RB_INT	RBIF (INTCON)	RBIE (INTCON)	RBIP (INTCON2)

Note: INT0 has only the high-level priority.



GIE (Global Interrupt Enable) This is also referred to as GIEH
GIE = 0 Disables all interrupts. If GIE = 0, no interrupt is acknowledged, even if they are enabled individually.
If GIE = 1, interrupts are allowed to happen. Each interrupt source is enabled by setting the corresponding interrupt enable bit.
GIEL (This is called Global Interrupt Enable Low to distinguish it from D7)
While GIH (D7) is used to enable or disable all the interrupts (both low and high priority), the GIEL (D6) is used to enable or disable only the low-priority interrupts. That means GIEL works only when the IPEN (Interrupt Priority Enable) bit is enabled.
If IPEN = 0
GIEL = 0 Disables all peripheral interrupts (same as PEIE)
GIEL = 1 Enables all peripheral interrupts (same as PEIE)
If IPEN = 1
GIEL = 0 Disables the Low-priority Interrupt
GIEL = 1 Enables the Low-priority Interrupt

Figure 11-21. INTCON (Interrupt Control) Register

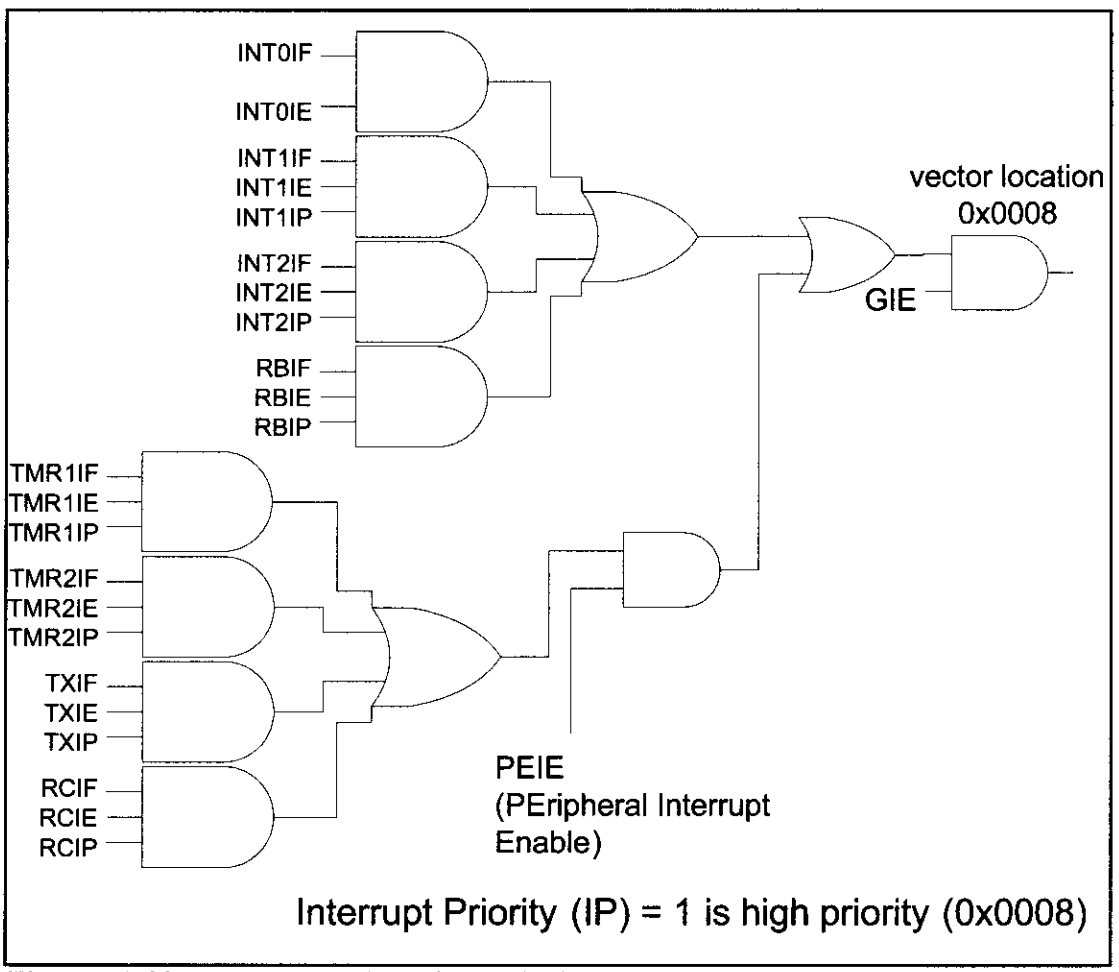


Figure 11-22. Interrupts with High-Priority (IP) Flag

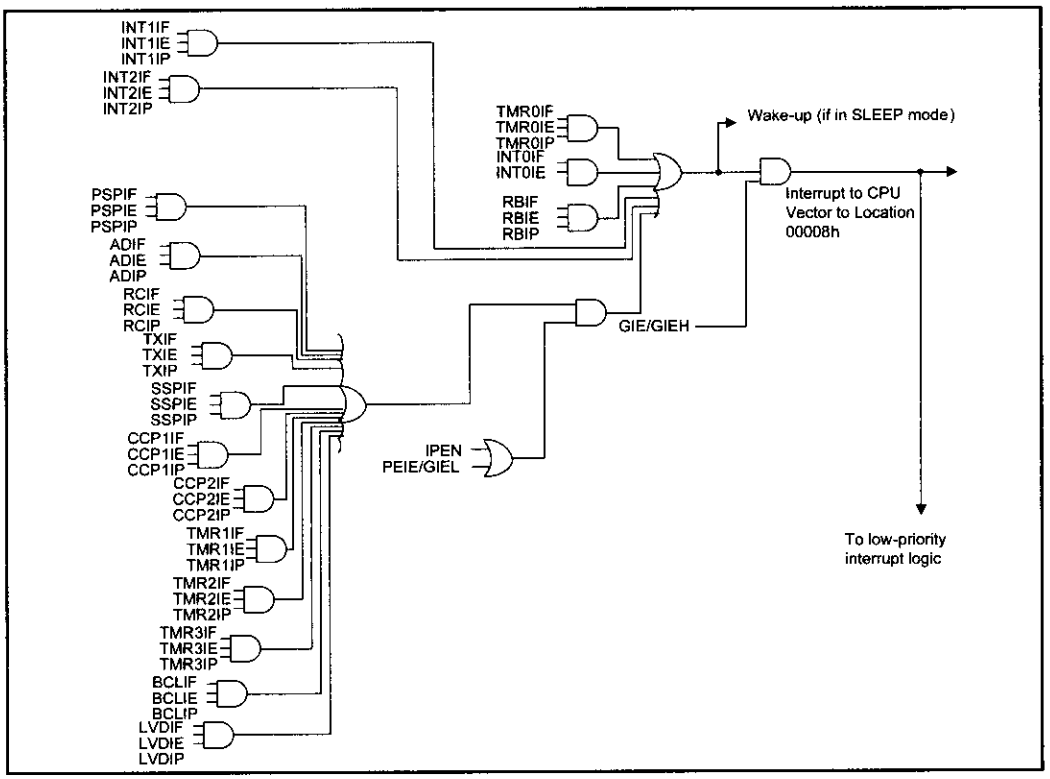


Figure 11-23. High-Priority Interrupts (Redrawn from PIC18 Manual)

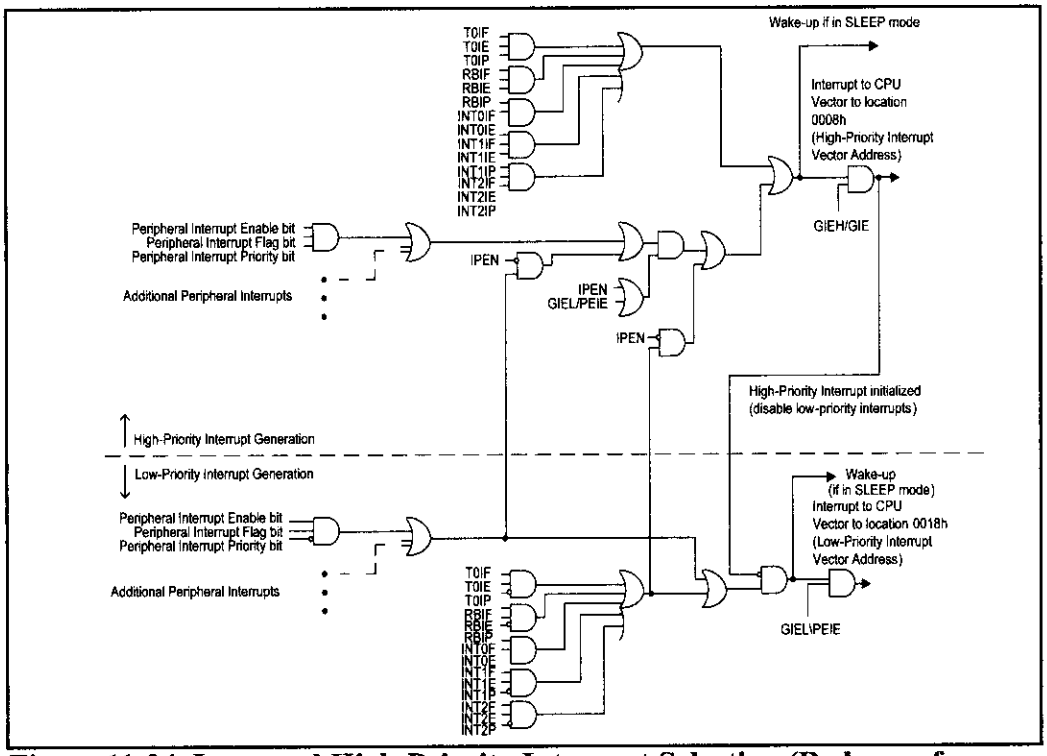


Figure 11-24. Low- and High-Priority Interrupt Selection (Redrawn from PIC18 Manual)

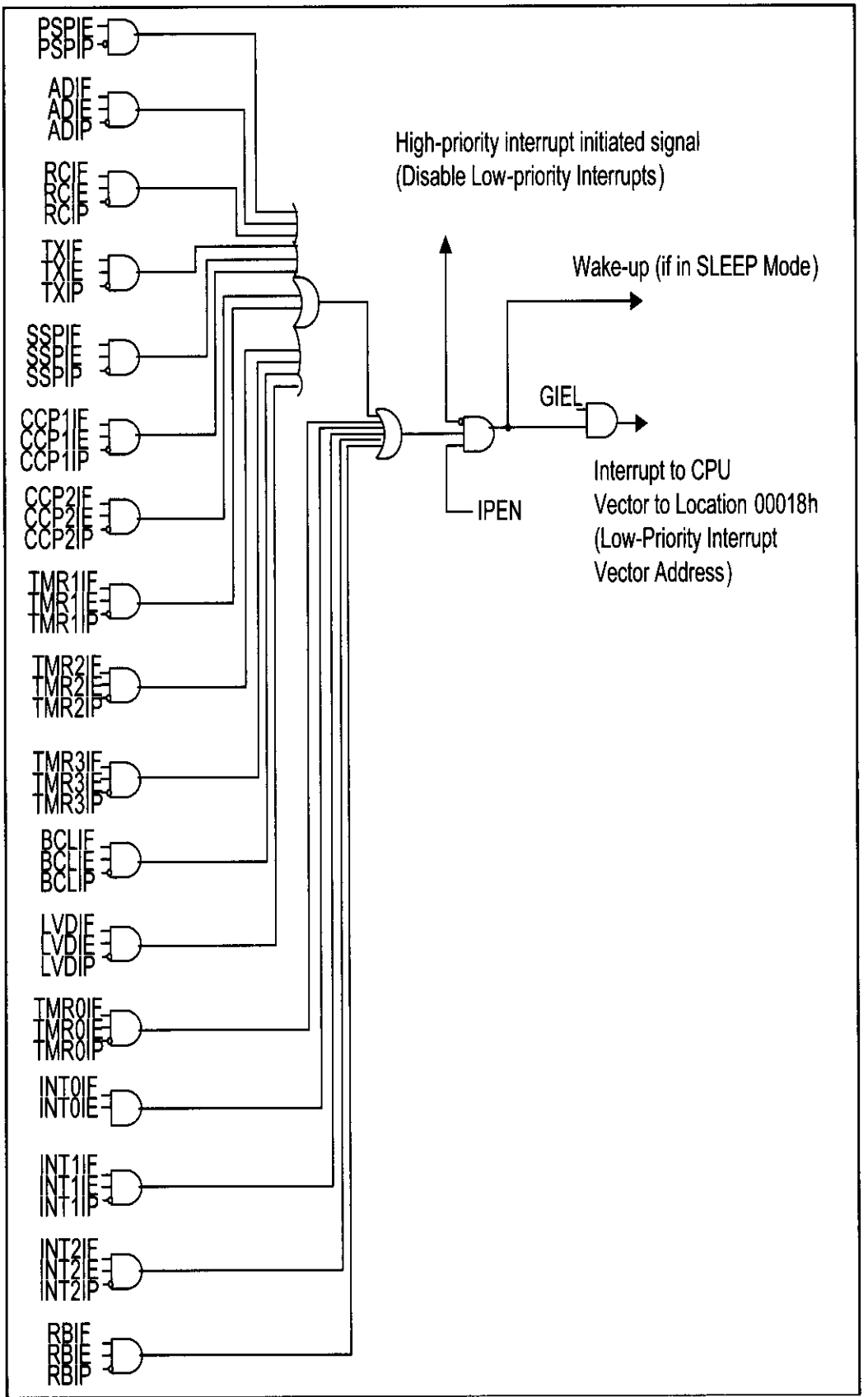


Figure 11-25. Low-Priority Interrupt Selection with IP Flag (Redrawn from PIC18 Manual)

Program 11-10 uses Timer0 and Timer1 interrupts to generate square waves on pins RB1 and RB7 respectively, while data is being transferred from PORTC to PORTD. This is a repeat of Program 11-2, except Timer1 has been assigned to low priority.

```
;Program 11-10
      ORG   0000H
      GOTO  MAIN          ;bypass interrupt vector table
;--high-priority interrupts go to address 00008
      ORG   0008H         ;high-priority interrupt vector table
      BTFSC INTCON,TMR0IF   ;Is it Timer0 interrupt?
      BRA T0_ISR            ;Yes. Then branch to T0_ISR
      RETFIE 0x01           ;No. Then fast return to main
;--low-priority interrupts go to address 00018
      ORG   0018H         ;low-priority interrupt vector table
      BTFSC PIR1,TMR1IF     ;Is it Timer1 interrupt?
      BRA T1_ISR            ;Yes. Then branch to T1_ISR
      RETFIE                ;No. Then return to main

;-main program for initialization and keeping CPU busy
      ORG   0100H  ;somewhere after vector table space
MAIN  BCF   TRISB,1     ;PB1 as an output
      BCF   TRISB,7     ;PB7 as an output
      CLRF  TRISD       ;make PORTD output
      SETF  TRISC       ;make PORTC input
      MOVLW 0x08        ;Timer0, 16-bit, no prescale,
                        ;internal clk
      MOVWF T0CON       ;load T0CON reg
      MOVLW  0xFF       ;TMR0H = FFH, the high byte
      MOVWF  TMR0H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR0L = 00H, the low byte
      MOVWF TMR0L       ;load Timer0 low byte
      BCF INTCON,TMR0IF ;clear Timer0 interrupt flag bit
      BSF INTCON,TMR0IE ;enable Timer0 interrupt
      MOVLW 0x0         ;Timer1, 16-bit, no prescale,
                        ;internal clk
      MOVWF T1CON       ;load T1CON reg
      MOVLW  0xFF       ;TMR1H = FFH, the high byte
      MOVWF  TMR1H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR1L = 00H, the low byte
      MOVWF TMR1L       ;load Timer1 low byte
      BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
      BSF PIE1,TMR1IE ;enable Timer1 interrupt
      BCF IPR1,TMR1IP ;make Timer1 low-priority interrupt
      BSF   RCON,IPEN  ;enable priority levels
      BSF INTCON,GIEL
      BSF INTCON,GIEH ;enable interrupts globally
      BSF T1CON,TMR1ON;start Timer1
      BSF T0CON,TMR0ON;start Timer0
;--keeping CPU busy waiting for interrupt
OVER  MOVFF PORTC,PORTD ;send data from PORTC to PORTD
      BRA OVER          ;stay in this loop forever
```

```
;--------------------------ISR for Timer0
T0_ISR
      ORG 200H
      MOVLW  0xFF       ;TMR0H = FFH, the high byte
      MOVWF  TMR0H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR0L = 00H, the low byte
      MOVWF TMR0L       ;load Timer0 low byte
      BTG   PORTB,1     ;toggle RB1
      BCF INTCON,TMR0IF ;clear timer interrupt flag bit
      RETFIE 0x01
;--------------------------ISR for Timer1
T1_ISR
      ORG 300H
      MOVLW  0xFF       ;TMR1H = FFH, the high byte
      MOVWF  TMR1H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR1L = 00H, the low byte
      MOVWF TMR1L       ;load Timer1 low byte
      BTG PORTB,7
      BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
      RETFIE
      END
```

Program 11-11 has four interrupts. It uses Timer0 and Timer1 interrupts to generate square waves on pins RC0 and RC1 respectively. It also uses the transmit and receive interrupts to send and receive data serially. Data from PORTD is transmitted and the received byte is placed on PORTB. Timer0 and receive ISRs have the high priority while Timer1 and send ISRs are assigned a low priority level.

```
;Program 11-11
      ORG   0000H
      GOTO  MAIN        ;bypass interrupt vector table
;--high-priority interrupts go to address 00008
      ORG   0008H       ;need to redirect because not
                        ;enough space
      GOTO CHK_HI_PRIO
;--no need to redirect because we have plenty of space
      ORG   00018       ;low-priority interrupt vector table
      BTFSC PIR1,TMR1IF     ;is it Timer1 interrupt?
      BRA   T1_ISR      ;Yes. Then branch to T1_ISR
      BTFSC PIR1,TXIF ;Did we get here due to TxD?
      BRA   TX_ISR      ;Yes. Then branch to TX_ISR
      RETFIE            ;No. Then return to main
;----------------
CHK_HI_PRIO      ORG   0x50
      BTFSC INTCON,TMR0IF   ;Is it Timer0 interrupt?
      BRA   T0_ISR      ;Yes. Then branch to T0_ISR
      BTFSC PIR1,RCIF ;Did we get here due to RCV int?
      BRA RC_ISR        ;Yes. Then branch to RC_ISR
      RETFIE 0x01       ;No. Then return to main
```

```
;-main program for initialization and keeping CPU busy
      ORG   0100H       ;somewhere after vector table space
MAIN  BCF   TRISC,RC0
      BCF   TRISC,RC1
      CLRF  TRISB       ;make PORTB output
      SETF  TRISD       ;make PORTD input
      MOVLW 0x08        ;Timer0, 16-bit, no prescale,
                        ;internal clk
      MOVWF T0CON       ;load T0CON reg
      MOVLW  0xFF       ;TMR0H = FFH, the high byte
      MOVWF  TMR0H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR0L = 00H, the low byte
      MOVWF TMR0L       ;load Timer0 low byte
      BCF INTCON,TMR0IF ;clear Timer0 interrupt flag bit
      MOVLW 0x0         ;Timer1, 16-bit, no prescale,
                        ;internal clk
      MOVWF T1CON       ;load T1CON reg
      MOVLW  0xFF       ;TMR1H = FFH, the high byte
      MOVWF  TMR1H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR1L = 00H, the low byte
      MOVWF TMR1L       ;load Timer1 low byte
      BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
      MOVLW 0x20        ;enable transmit and choose low baud
      MOVWF TXSTA       ;write to reg
      MOVLW D'15'       ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG       ;write to reg
      MOVLW 0x90        ;enable receive and serial port
      MOVWF RCSTA       ;write to reg
      BCF TRISC, TX     ;make TX pin of PORTC an output pin
      BSF TRISC, RX     ;make RCV pin of PORTC an input pin
      BSF   RCON,IPEN
      BSF   PIE1,RCIE
      BSF PIE1,TXIE     ;enable TX interrupt
      BSF INTCON,TMR0IE     ;enable Timer0 interrupt
      BSF PIE1,TMR1IE ;enable Timer1 interrupt
      BCF IPR1,TMR1IP ;make Timer1 a low-priority interrupt
      BCF IPR1,TXIP ;make Transmit a low-priority interrupt
      BSF T0CON,TMR0ON      ;start Timer0
      BSF T1CON,TMR1ON      ;start Timer1
      BSF INTCON,GIEL       ;enable low-priority interrupts
      BSF INTCON,GIEH       ;enable high-priority interrupts
;--keeping CPU busy waiting for interrupt
OVER BRA    OVER            ;stay in this loop forever

;--------------------------ISR for Timer0
T0_ISR     ORG 200H
      MOVLW  0xFF       ;TMR0H = FFH, the high byte
      MOVWF  TMR0H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR0L = 00H, the low byte
      MOVWF TMR0L       ;load Timer0 low byte
      BTG   PORTC,0     ;toggle RB1
```

```
      BCF INTCON,TMR0IF ;clear timer interrupt flag bit
      RETFIE 0x01
;--------------------------ISR for Timer1
T1_ISR     ORG 300H
      MOVLW  0xFF       ;TMR1H = FFH, the high byte
      MOVWF  TMR1H      ;load Timer0 high byte
      MOVLW 0x00        ;TMR1L = 00H, the low byte
      MOVWF TMR1L       ;load Timer1 low byte
      BTG PORTC,1
      BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
      RETFIE
;----------------Transmit ISR
TX_ISR
      BCF PIR1,TXIF ;clear TX interrupt flag bit
      MOVFF PORTD,TXREG
      RETFIE
;------------
RC_ISR
      MOVFF RCREG,PORTB   ;copy received data to PORTD
      BCF   PIR1, RCIF       ;clear RCIF
      RETFIE 1
      END
```

Example 11-3

For Program 11-11 (or Program 11-11C), discuss what happens: (a) if interrupt RCIF is activated when the PIC18 is serving the Timer1 interrupt, (b) Timer 1 is activated when Timer0 is being served, (c) RCIF and TMR0IF and TMR1IF are activated at the same time.

Solution:

In Program 11-11, notice that the Receive (RCIF) and Timer0 (TMR0IF) interrupts are assigned to high priority while the Transmit (TXIF) and Timer1 (TMR1IF) interrupts have low priority. As a result we have the following:

(a) if the RCIF is activated during the execution of the Timer1 ISR, the Receive interrupt comes in and its ISR is executed first because it has a higher priority. After it is finished, the PIC18 goes back and finishes the Timer1 ISR.

(b) if the TMR1IF is activated during the execution of the Timer0 ISR, it is ignored because it has lower priority. After the Timer0 ISR is finished, the PIC18 will execute the Timer1 ISR.

(c) If all three, RCIF, TMR0IF, and TMR1IF, are activated at the same time, the Received ISR and and Timer0 ISR are taken care of first because they are assigned to high priority. Between the Receive and Timer0 interrupts, the Timer0 ISR is served first due to the programming sequence we have set in the interrupt vector table for the high-priority interrupts. That means if these three interrupts are activated at the same time, they are executed in the following sequence: Timer0 ISR, Receive ISR, and Timer1 ISR.

Low-priority interrupt programming in C

As we saw in the last four sections, the C18 compiler uses the reserved keyword **interrupt** to designate an interrupt as high priority. To assign low priority level to a given interrupt, it uses the keyword **interruptlow**. See Table 11-7. This is shown in Program 11-11C, which is a repeat of Program 11-11 in C.

Table 11-7: Interrupt Vector Table for the PIC18 using C18 Syntax

Interrupt	ROM Location	C18 keyword
High-priority Interrupt	0x0008 (Default)	**interrupt**
Low-priority Interrupt	0x0018 (Selected with IP bit)	**interruptlow**

Program 11-11C has four interrupts. It uses Timer0 and Timer 1 interrupts to generate square waves on pins RC1 and RC7, respectively. It also uses the transmit and receive interrupts to send and receive data serially. Data from PORTB is transmitted and the received byte is placed on PORTD. Timer0 and receive ISRs have the high priority while Timer1 and transmit ISRs are assigned low priority level.

```
//Program 11-11C (This is the C version of Program 11-11)
#include <p18F458.h>
#define myPC0bit PORTCbits.RC0
#define myPC1bit PORTCbits.RC1

void chk_isr(void);
void chk_low_isr(void);
void T0_ISR(void);
void T1_ISR(void);
void TX_ISR(void);
void RC_ISR(void);

#pragma code My_HiPrio_Int =0x0008 //high-priority int
void My_HiPrio_Int (void)
{
_asm
GOTO chk_isr
_endasm
}

#pragma code My_Lo_Prio_Int =0x00018 //low-priority int
void My_Lo_Prio_Int (void)
{
      _asm
            GOTO chk_low_isr
      _endasm
}

#pragma interruptlow chk_low_isr //used for low-priority
void chk_low_isr (void)
{
```

```
     if(PIR1bits.TMR1IF==1)//Timer1 causes interrupt?
       T1_ISR();           //Yes. Execute Timer1 ISR
     if (PIR1bits.TXIF==1) //Transmit causes interrupt?
       TX_ISR( );          //Yes. Execute Transmit ISR
}

#pragma interrupt chk_isr//used for high-priority interrupt
void chk_isr (void)
{
     if (PIR1bits.TMR1IF==1)//Timer0 causes interrupt?
          T0_ISR( );       //Yes. Execute Timer0 ISR
     if (PIR1bits.RCIF==1) //Receiver causes interrupt?
          RC_ISR( );       //Yes. Execute Receiver ISR
}

void main(void)
  {
     TRISCbits.TRISC0=0;   //RC0 = OUTPUT
     TRISCbits.TRISC1=0;   //RC1 = OUTPUT
     TRISD = 255;          //PORTD = INPUT
     TRISB = 0;            //PORTB = OUTPUT
     T0CON=0x08;           //Timer0, 16-bit mode,
                           //no prescaler
     TMR0H=0xFF;           //load TH0
     TMR0L=0x00;           //load TL0
     INTCONbits.TMR0IF=0;  //clear TF1
     T1CON=0x0; //Timer 1, 16-bit mode, no prescaler
     TMR1H=0xFF;           //load TH1
     TMR1L=0x00;           //load TL1
     PIR1bits.TMR1IF=0;    //clear TF1
     TXSTA=0x20;           //choose low baud rate,8-bit
     SPBRG=15;             //9600 baud rate/ XTAL = 10 MHz
     RCSTAbits.CREN=1;
     RCSTAbits.SPEN=1;
     TRISCbits.TRISC6=0;   //TX pin = OUTPUT
     TRISCbits.TRISC7=1;   //RCV pin = INPUT
     RCONbits.IPEN=1;
     PIE1bits.RCIE=1;
     PIE1bits.TXIE=1;      //enable TX interrupt
     INTCONbits.TMR0IE=1;
     PIE1bits.TMR1IE=1;
     IPR1bits.TMR1IP=0;    //make Timer1 a low-priority
     IPR1bits.TXIP=0;      //make TX a low-priority
     T0CONbits.TMR0ON=1;   //turn on T0
     T1CONbits.TMR1ON=1;   //turn on T1
     INTCONbits.GIEL=1;    //enable low-priority interrupts
     INTCONbits.GIEH=1;//enable high-priority interrupts
     while(1);       //keep looping until interrupt comes
  }
```

```
//-------------ISR for Timer0
void T0_ISR(void)
  {
     TMR0H=0xFF;            //load TH0
     TMR0L=0x00;            //load TL0
     myPC0bit=~myPC0bit;    //toggle RB1
     INTCONbits.TMR0IF=0;   //clear TF0
  }
//-------------ISR for Timer1
void T1_ISR(void)
  {
     TMR1H=0xFF;            //load TH0
     TMR1L=0x00;            //load TL0
     PIR1bits.TMR1IF=0;     //clear TF1
     myPC1bit=~myPC1bit;    //toggle RB1
  }
//-------------ISR for Transmit
void TX_ISR(void)
  {
    TXREG=PORTD;      //clear Tx Interrupt flag
  }
//-------------ISR for Receive
void RC_ISR(void)
  {
     PORTB=RCREG;
     RCSTAbits.CREN=0;//clear CREN to clear any error
     RCSTAbits.CREN=1;//set CREN for continuous reception
  }
```

Interrupt inside an interrupt

What happens if the PIC18 is executing an ISR belonging to an interrupt and another interrupt is activated? In such cases, a high-priority interrupt can interrupt a low-priority interrupt. This is an interrupt inside an interrupt. In the PIC18 a low-priority interrupt can be interrupted by a higher-priority interrupt, but not by another low-priority interrupt. Although all the interrupts are latched and kept internally, no low-priority interrupt can get the immediate attention of the CPU until the PIC18 has finished servicing all the high-priority interrupts. The GIE (which is also called GIEH) and GIEL bits play an important role in the process of the interrupt inside the interrupt. Regarding the interrupt inside an interrupt concept, the following points must be emphasized:

1. When a high-priority interrupt is vectored into address 0008H, the GIE bit is disabled (GIEH = 0), thereby blocking another interrupt (low or high) from coming in. The RETFIE instruction at the end of the ISR will enable the GIE (GIE = 1) automatically, which allows interrupts to come in again. If we want to allow another high-priority interrupt to come in during the execution of the current ISR, then we must make GIE = 1 at the beginning of the current ISR.
2. When a low-priority interrupt is vectored into address 0018H, the GIEL bit is disabled (GIEL = 0), thereby blocking another low-priority interrupt from

coming in. The RETFIE instruction at the end of the ISR will enable the GIEL (GIEL = 1) automatically, which allows low-priority interrupts to come in again. Notice that the low-priority interrupt cannot block a high-priority interrupt from coming in during the execution of the current low-priority ISR because GIEH is still set to one (GIEH = 1).

3. When two or more interrupts have the same priority level. In this case, they are serviced according to the sequence by which the program checks them in the interrupt vector table. We saw many examples of that in this chapter. See Example 11-3.

Fast context saving in task switching

In many applications, such as multitasking real-time operating systems (RTOS), the CPU brings in one task (job or process) at a time and executes it before it moves to the next one. In executing each task, which is often organized as the interrupt service routine, access to all the resources of the CPU is critical in performing the task in a timely manner. In early CPUs, the limited number of registers forced programmers to save the entire contents of the CPU on the stack before execution of the new task. This saving of the CPU contents before switching to a new task is called *context saving* (or *context switching*). The use of the stack as a place to save the CPU's contents is tedious, time consuming, and slow. For this reason some CPUs such as x86 microprocessors have instructions such as PUSHA (Push All) and POPA (Pop All), which will push and pop all the main registers onto the stack with a single instruction. Because the PIC18 has numerous general purpose registers, there is no need for using the stack to save the CPU's general purpose registers. However, each task generally needs the key registers of WREG, BSR, and STATUS. For that reason the PIC18 automatically saves these three registers internally in shadow registers when a high-priority interrupt is activated. This way, the three key registers of the main task are saved internally. To restore the original contents of these three key registers, one must use instruction "RETFIE 0x01" instead of "RETFIE" at the end of the high-priority ISR. The "RETFIE 0x01" is called fast context saving in PIC18 literature. Regarding fast context saving in the PIC18, two important points must be noted:

1. It is not available for the low-priority interrupts, and works only for high-priority interrupts. That means that when a low-priority interrupt is activated, there is no fast context saving and we must save these three registers at the beginning of the low-priority ISR, if they are being used by the low-priority ISR.
2. The shadow registers keeping these three key registers have a depth of one, meaning that there is only one of them. For that reason, the fast context saving works only when a high-priority ISR is activated during the main subroutine. If two or three high-priority interrupts are activated at the same time, only the first ISR can use the fast context saving because the depth of shadow registers is only one. In that case, the second and third ISRs must save these key registers at the beginning of the body of their ISRs. This should not be difficult because we know the sequence by which the ISRs are executed, as we saw in many examples in this chapter.

Interrupt latency

The time from the moment an interrupt is activated to the moment the CPU starts to execute the code at the vector address of 0008H (or 0x0018H) is called the *interrupt latency*. This latency can be anywhere from 2 to 4 instruction cycle times depending on whether the source of the interrupt is an internal (e.g., timers) or external hardware (e.g., hardware INTx and PORTB-Change) interrupt. The duration of an interrupt latency can also be affected by the type of the instruction in which the CPU was executing when the interrupt comes in. It takes slightly longer in cases where the instruction being executed lasts for two instruction cycles (e.g., MOVFF reg,reg) compared to the instructions that last for only one instruction cycle time (e.g., ADDWL). See PIC18 for the timing data sheet.

Triggering the interrupt by software

Sometimes when we need to test an ISR by way of simulation. This can be done with simple instructions to set the interrupts HIGH and thereby cause the PIC18 to jump to the interrupt vector table. For example, if the TMR1IE bit for Timer 1 is set, an instruction such as "`BSF INTCON,TMR1IF`" will interrupt the PIC18 in whatever it is doing and force it to jump to the interrupt vector table. In other words, we do not need to wait for Timer 1 to roll over to have an interrupt. We can cause an interrupt with an instruction that raises the interrupt flag.

Review Questions

1. True or false. Upon reset, all interrupts have the same priority.
2. Which bit of what register is used to enable the interrupt priority option in the PIC18? Is it a bit-addressable register?
3. Which register has the TXIP bit? Show how to assign it low priority.
4. Assume that INT0 and INT1 have the same low priority. Explain what happens if both INT0 and INT1 are activated at the same time. Also assume that INT0 is checked first in the program for the interrupt vector table.
5. Explain what happens if a higher-priority interrupt is activated while the PIC18 is serving a lower-priority interrupt (i.e., executing a lower-priority ISR).

SUMMARY

An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service. Every interrupt has a program associated with it called the ISR, or interrupt service routine. The PIC18 has many sources of interrupts, depending on the family members. Some of the most widely used interrupts are for the timers, external hardware interrupts, and serial communication. When an interrupt is activated, the IF (Interrupt flag) bit is raised.

The PIC18 can be programmed to enable (unmask) or disable (mask) an interrupt, which is done with the help of the GIE (global interrupt enable) and IE (interrupt enable) bits. The PIC18 has two levels of priority, low and high. Upon power-on reset, all the interrupts are designated as high priority and are directed to address 0008 in the interrupt vector table. This default setting can be altered with the help of the IP (interrupt priority) bits. By programming the IP bit, we can make

an interrupt a low priority and force it to land at address 0x00018 in the interrupt vector table. This chapter also showed how to program PIC18 interrupts in both Assembly and C languages.

PROBLEMS

SECTION 11.1: PIC18 INTERRUPTS

1. Which technique, interrupt or polling, avoids tying down the microcontroller?
2. List some of the interrupt sources in the PIC18.
3. In the PIC18 what memory area is assigned to the interrupt vector table?
4. True or false. The PIC18 programmer cannot change the memory address location assigned to the interrupt vector table.
5. What memory address in the interrupt vector table is assigned to low-priority interrupts?
6. What memory address in the interrupt vector table is assigned to high-priority interrupts?
7. Do we have a memory address in the interrupt vector table assigned to the Timer0 interrupt?
8. Do we have a memory address in the interrupt vector table assigned to the INT1 interrupt?
9. To which register does the GIE bit belong?
10. Why do we put a GOTO instruction at address 0?
11. What is the state of the GIE bit upon power-on reset, and what does it mean?
12. Show the instruction to enable the INT0 interrupt.
13. Show the instruction to enable the Timer0 interrupt.
14. The TMR0IE bit belongs to register_____.
15. How many bytes of address space in the interrupt vector table are assigned to the high-priority interrupt?
16. How many bytes of address space in the interrupt vector table are assigned to the low-priority interrupt?
17. To put the entire interrupt service routine in the interrupt vector table for high priority, it must be no more than _____ bytes in size.
18. True or false. The INTCON register is not a bit-addressable register.
19. With a single instruction, show how to disable all the interrupts.
20. With a single instruction, show how to disable the INT0 interrupt.
21. True or false. Upon reset, all interrupts are enabled by the PIC18.
22. In the PIC18, how many bytes of ROM space are assigned to the reset?

SECTION 11.2: PROGRAMMING TIMER INTERRUPTS

23. True or false. For each of Timer 0 and Timer1, there is a unique address in the interrupt vector table.
24. What address in the interrupt vector table is assigned to Timer1?
25. Show how to enable the Timer0 interrupt.
26. Which bit of INTCON belongs to the Timer0 interrupt? Show how it is enabled.

27. Assume that Timer0 is programmed in 8-bit mode, TMR0H = F0H, and the TMR0IE bit is enabled. Explain how the interrupt for the timer works.
28. True or false. The last two instructions of the ISR for Timer1 are:
```
BCF   PIR1,TMR1IF
RETFIE
```
29. Assume that Timer1 is programmed for 16-bit mode, TMR1H = FFH, TMR1L = F8H, and the TMR1IE bit is enabled. Explain how the interrupt is activated.
30. If Timer 1 is programmed for interrupts in 8-bit mode, explain when the interrupt is activated.
31. Write a program using the Timer0 interrupt to create a square wave of 1 Hz on pin RB7 while data from PORTC is being sent to PORTD. Assume XTAL = 10 MHz.
32. Write a program using the Timer1 interrupt to create a square wave of 3 kHz on pin RB7 while data from PORTC is being sent to PORTD. Assume XTAL = 10 MHz.

SECTION 11.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

33. True or false. An address location is assigned to each of the external hardware interrupts INT0, INT1, and INT2.
34. What address in the interrupt vector table is assigned to INT0, INT1 and INT2? How about the pin numbers on PORTB?
35. To which register does the INT0IE bit belong? Show how it is enabled.
36. To which register does the INT1IE bit belong? Show how it is enabled.
37. Show how to enable all three external hardware interrupts.
38. Assume that the INT0IE bit for external hardware interrupt INT0 is enabled and is negative edge-triggered. Explain how this interrupt works when it is activated.
39. True or false. Upon reset, all the external hardware interrupts are negative edge-triggered.
40. In Question 38, how do we make sure that a single interrupt is not recognized as multiple interrupts?
41. The INT0IF bit belongs to the _____ register.
42. The INT1IF bit belongs to the _____ register.
43. True or false. The last two instructions of the ISR for INT1 are:
```
BCF   INTCON3,INT1IF
RETFIE
```
44. Explain the role of INT0IF and INT0IE in the execution of external interrupt 0.
45. Explain the role of INT1IF and INT1IE in the execution of external interrupt 1.
46. Assume that the INT1IE bit for external hardware interrupt INT1 is enabled and is positive edge-triggered. Explain how this interrupt works when it is activated. How can we make sure that a single interrupt is not interpreted as multiple interrupts?
47. True or false. INT0–INT2 are part of the PEIE group.
48. True or false. Upon power-on reset, all of INT0–INT2 are positive edge-triggered.
49. Explain the difference between positive and negative edge-triggered interrupts.

50. How do we make the hardware interrupt negative edge-triggered?
51. True or false. INT0–INT2 must be configured as an input pin for a hardware interrupt to come in.
52. Which register holds the INTEDGx bits?

SECTION 11.4: PROGRAMMING THE SERIAL COMMUNICATION INTERRUPTS
and
SECTION 11.5: PORTB-CHANGE INTERRUPT

53. True or false. Two separate interrupts are assigned to each of the interrupts, TXIF and RCIF.
54. Upon power-on reset, what address in the interrupt vector table is assigned to the serial interrupt? How many bytes are assigned to it?
55. To which register does the TXIF belong? Show how it is enabled.
56. Assume that the TXIE bit for the serial interrupt is enabled. Explain how this interrupt gets activated and also explain its working upon activation.
57. True or false. Upon reset, serial interrupts are blocked.
58. True or false. The last two instructions of the ISR for the transmit interrupt are:

```
BCF   PIR1,TXIF
RETFIE
```

59. State how the RCIF is cleared.
60. Assuming that the TXIE bit is set when TXIF is raised, what happens subsequently?
61. Assuming that the RCIE bit is set when RCIF is raised, what happens subsequently?
62. Write a program using interrupts to get data serially and send it to PORTD while at the same time any changes on PORTB.4 will cause the LED connected to PORTC.7 to toggle.
63. Provide the following information for the PORTB-Change interrupt.
    (a) the flag associated with the PORTB-Change interrupt
    (b) the register to which these flag belong
    (c) the difference between the PORTB-Change and INT0–INT2 interrupts
    (d) the pins that are part of the PORTB-Change interrupt

SECTION 11.6: INTERRUPT PRIORITY IN THE PIC18

64. True or false. Upon reset, all interrupts have high priority.
65. What register enables the interrupt priority in the PIC18 ? Explain its role.
66. Which register has the INT0IP bit? Show how to assign it low priority.
67. Which register has TMR1IP bit? Show how to assign it low priority.
68. Which register has the INT1IP bit? Show how to assign it low priority.
69. Assume that INT1IP and INT2IP are both 0s. Explain what happens if both INT1IF and INT2IF are activated at the same time.
70. Assume that TMR0IP and TMR1IP are both 0s. Explain what happens if both TMR0IF and TMR1IF are activated at the same time.
71. If both TMR0IP and TMR1IP are set to HIGH, what happens if both are activated at the same time?

72. If both INT1IP and INT2IP in the IP are set to HIGH, what happens if both are activated at the same time?
73. Explain what happens if a low-priority interrupt is activated while the PIC18 is serving a high-priority interrupt.
74. Explain what happens if a high-priority interrupt is activated while the PIC18 is serving a low-priority interrupt.
75. Explain the role of the GIEH bit in masking and unmasking the interrupts.
76. True or false. In PIC18, an interrupt inside an interrupt is not allowed.
77. Explain the role of the GIEL bit in masking and unmasking interrupts.
78. Explain the role of RETFIE in enabling the GIEL bit.
79. Explain the difference between the "RETFIE" and "RETFIE 1" instructions.
80. Explain the concept of fast context saving in PIC.

ANSWERS TO REVIEW QUESTIONS

SECTION 11.1: PIC18 INTERRUPTS

1. Interrupts
2. INT0 and TMR0
3. Address locations 0x0008 to 0x00017. No. It is set when the processor is designed.
4. GIE = 0 means that all interrupts are masked, and as a result no interrupts will be responded to by the PIC18.
5. Assuming GIE = 1, we need "`BSF INTCON,TMR0IE`".
6. 0008 for the high-priority interrupts and 0x0018 for the low-priority interrupts.

SECTION 11.2: PROGRAMMING TIMER INTERRUPTS

1. False. There is a single address for all the timers, Timer0, Timer1, and so on.
2. 0008H
3. PIE1 and "`BSF PIE1,TMR1IE`" will enable the Timer1 interrupt.
4. After Timer1 is started, the timer will count up from F5H to FFH on its own while the PIC18 is executing other tasks. Upon rolling over from FFH to 00, the TMR1IF flag is raised, which will interrupt the PIC18 in whatever it is doing and forces it to jump to memory location 0008 to execute the ISR belonging to this interrupt.
5. True

SECTION 11.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

1. True
2. Bits RB0 (PORTB.0), RB1 (PORTB.1), and RB2 (PORTB.2)
3 BSF INTCON3,INT1IE
4. Upon application of a low-to-high pulse to pin RB0, the PIC18 is interrupted in whatever it is doing and jumps to ROM location 0008H to execute the ISR.
5. False
6. When the CPU jumps to ROM location 0008 to execute the ISR, the GIE becomes 0, effectively blocking another interrupt from the same source. The last two instructions of the ISR are "`BCF INCON, INT0IF`" followed by "`RETFIE`". While the first instruction will clear the previous request for interrupt, the second one will make GIE = 1, allowing a new interrupt to come in from the same source. That can happen only if a new low-to-high pulse is applied to the pin.
7. True

SECTION 11.4: PROGRAMMING THE SERIAL COMMUNICATION INTERRUPTS

1. True. There is only one interrupt for all all interrupts including the transfer and receive.
2. 0x0008 for high-priority interrupts and 0x0018 for low-priority interrupts.
3. "`BSF PIE1,TXIE`" will enable the send interrupt and "`BSF PIE1,RCIE`" will enable the receive interrupt.
4. The RCIF (received interrupt flag) is raised when the entire frame of data, including the stop bit, is received. As a result the received byte is delivered to the RCREG register and the PIC18 jumps to memory location 0008H to execute the ISR belonging to this interrupt. In the serial COM interrupt service routine, we must save the RCREG content before it is lost by the incoming data.
5. False
6. True
7.
```
BCF RIR1,TXIF
RETFIE
```

SECTION 11.5: PORTB-CHANGE INTERRUPT

1. False
2. All interrupts, including the PORTB-Change interrupt, go to location 0008 on default.
3. `INTCON,` and we enable it with the instruction "`BSF INTCON, RBIF`"
4.
```
BCF INTCON,RBIF
RETFIE
```
5. False

SECTION 11.6: INTERRUPT PRIORITY IN THE PIC18

1. True
2. IPEN bit of the RCON register. Yes, it is bit-addressable.
3. IPR1 and the instruction "`BCF IPR1,TXIP`" will do it.
4. If both are activated at the same time, INT0 is serviced first because it is checked first. After INT0 is serviced, INT1 is serviced.
5. We have an interrupt inside an interrupt, meaning that the lower-priority interrupt is put on hold and the higher one is serviced. After servicing this higher-priority interrupt, the PIC18 resumes servicing the lower-priority ISR.

CHAPTER 12

LCD AND KEYBOARD INTERFACING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Describe the functions of the pins of a typical LCD**
- **>> List instruction command codes for programming an LCD**
- **>> Interface an LCD to the PIC18**
- **>> Program an LCD in Assembly and C**
- **>> Explain the basic operation of a keyboard**
- **>> Describe the key press and detection mechanisms**
- **>> Interface a 4 × 4 keypad to the PIC18 using C and Assembly**

This chapter explores some real-world applications of the PIC18. We explain how to interface the PIC18 to devices such as an LCD and a keyboard. In Section 12.1, we show LCD interfacing with the PIC18. In Section 12.2, keyboard interfacing with the PIC18 is shown. We use C and Assembly for both sections.

SECTION 12.1: LCD INTERFACING

This section describes the operation modes of LCDs, then describes how to program and interface an LCD to a PIC18 using Assembly and C.

LCD operation

In recent years the LCD has been finding widespread use replacing LEDs (seven-segment LEDs or other multisegment LEDs). This is due to the following reasons:

1. The declining prices of LCDs.
2. The ability to display numbers, characters, and graphics. This is in contrast to LEDs, which are limited to numbers and a few characters.
3. Incorporation of a refreshing controller into the LCD, thereby relieving the CPU of the task of refreshing the LCD. In contrast, the LED must be refreshed by the CPU (or in some other way) to keep displaying the data.
4. Ease of programming for characters and graphics.

LCD pin descriptions

The LCD discussed in this section has 14 pins. The function of each pin is given in Table 12-1. Figure 12-1 shows the pin positions for various LCDs.

Table 12-1: Pin Descriptions for LCD

Pin	Symbol	I/O	Description
1	V_{SS}	--	Ground
2	V_{CC}	--	+5 V power supply
3	V_{EE}	--	Power supply to control contrast
4	RS	I	RS = 0 to select command register, RS = 1 to select data register
5	R/W	I	R/W = 0 for write, R/W = 1 for read
6	E	I/O	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus

V_{CC}, V_{SS}, and V_{EE}

While V_{CC} and V_{SS} provide +5 V and ground, respectively, V_{EE} is used for controlling LCD contrast.

RS, register select

There are two very important registers inside the LCD. The RS pin is used for their selection as follows. If RS = 0, the instruction command code register is selected, allowing the user to send a command such as clear display, cursor at home, and so on. If RS = 1 the data register is selected, allowing the user to send data to be displayed on the LCD.

R/W, read/write

R/W input allows the user to write information to the LCD or read information from it. R/W = 1 when reading; R/W = 0 when writing.

E, enable

The enable pin is used by the LCD to latch information presented to its data pins. When data is supplied to data pins, a high-to-low pulse must be applied to the En pin in order for the LCD to latch in the data present at the data pins. This pulse must be a minimum of 450 ns wide. In this book we call this delay the SDELAY (short delay) to distinguish it from other delays.

D0–D7

The 8-bit data pins, D0–D7, are used to send information to the LCD or read the contents of the LCD's internal registers.

To display letters and numbers, we send ASCII codes for the letters A–Z, a–z, and numbers 0–9 to these pins while making RS = 1.

There are also instruction command codes that can be sent to the LCD to clear the display or force the cursor to the home position or blink the cursor. Table 12-2 lists the instruction command codes. To send any of the commands listed in Table 12-2 to the LCD, make pin RS = 0. For data, make RS = 1. Then send a high-to-low pulse to the E pin to enable the internal latch of the LCD. There are two ways to send characters (command/data) to the LCD: (1) use a delay before sending the next one, (2) use the busy flag to see if the LCD is ready for the next one.

Table 12-2: LCD Command Codes

Code (Hex)	Command to LCD Instruction Register
1	Clear display screen
2	Return home
4	Decrement cursor (shift cursor to left)
6	Increment cursor (shift cursor to right)
5	Shift display right
7	Shift display left
8	Display off, cursor off
A	Display off, cursor on
C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines and 5x7 matrix

Note: This table is extracted from Table 12-4.

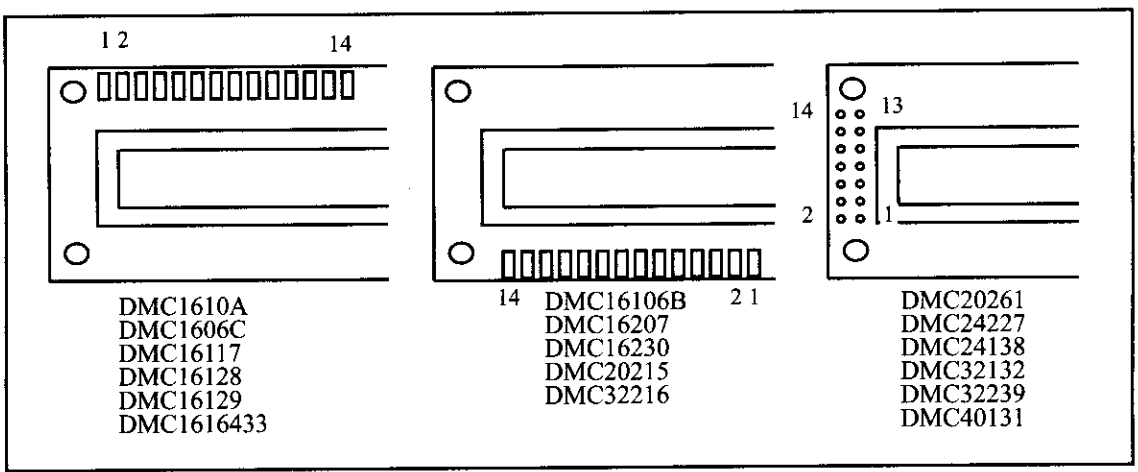


Figure 12-1. Pin Positions for Various LCDs from Optrex

Sending commands and data to LCDs with a time delay

Program 12-1 shows how to send characters (command/data) to the LCD without checking the busy flag. Notice that we need to wait 5–10 ms (DELAY) between issuing each character to the LCD. We call this delay simply DELAY. In programming an LCD, we also need a long delay for the power-up process. We call it LDELAY (long delay). SDELAY (short delay) is used to make the En signal wide enough for the LCD's enable input. See Chapter 3 for delays.

Figure 12-2 shows the LCD connections to the microcontroller.

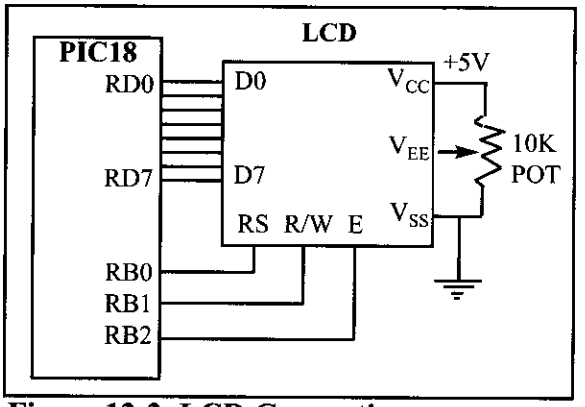


Figure 12-2. LCD Connections

```
;Program 12-1: Using delay before sending data/command
LCD_DATA   EQU PORTD           ;LCD data pins RD0-RD7
LCD_CTRL   EQU PORTB           ;LCD control pins
RS         EQU RB0             ;RS pin of LCD
RW         EQU RB1             ;R/W pin of LCD
EN         EQU RB2             ;E pin of LCD
           CLRF   TRISD        ;PORTD = Output
           CLRF   TRISB        ;PORTB = Output
           BCF    LCD_CTRL,EN  ;enable idle low
           CALL   LDELAY       ;wait for initialization
           MOVLW  0x38         ;init. LCD 2 lines, 5x7 matrix
           CALL   COMNWRT      ;call command subroutine
           CALL   LDELAY       ;initialization hold
           MOVLW  0x0E         ;display on, cursor on
           CALL   COMNWRT      ;call command subroutine
           CALL   DELAY        ;give LCD some time
           MOVLW  0x01         ;clear LCD
           CALL   COMNWRT      ;call command subroutine
           CALL   DELAY        ;give LCD some time
           MOVLW  0x06         ;shift cursor right
           CALL   COMNWRT      ;call command subroutine
           CALL   DELAY        ;give LCD some time
           MOVLW  0x84         ;cursor at line 1, pos. 4
           CALL   COMNWRT      ;call command subroutine
           CALL   DELAY        ;give LCD some time
           MOVLW  A'N'         ;display letter 'N'
           CALL   DATAWRT      ;call display subroutine
           CALL   DELAY        ;give LCD some time
           MOVLW  A'O'         ;display letter 'O'
```

```
            CALL    DATAWRT      ;call display subroutine
AGAIN       BTG     LCD_CTRL,0
            BRA     AGAIN        ;stay here
COMNWRT                          ;send command to LCD
            MOVWF   LCD_DATA     ;copy WREG to LCD DATA pin
            BCF     LCD_CTRL,RS  ;RS = 0 for command
            BCF     LCD_CTRL,RW  ;R/W = 0 for write
            BSF     LCD_CTRL,EN  ;E = 1 for high pulse
            CALL    SDELAY       ;make a wide En pulse
            BCF     LCD_CTRL,EN  ;E = 0 for H-to-L pulse
            RETURN
DATAWRT                          ;write data to LCD
            MOVWF   LCD_DATA     ;copy WREG to LCD DATA pin
            BSF     LCD_CTRL,RS  ;RS = 1 for data
            BCF     LCD_CTRL,RW  ;R/W = 0 for write
            BSF     LCD_CTRL,EN  ;E = 1 for high pulse
            CALL    SDELAY       ;make a wide En pulse
            BCF     LCD_CTRL,EN  ;E = 0 for H-to-L pulse
            RETURN
;look in previous chapters for delay routines
            END
```

Sending command or data to the LCD using busy flag

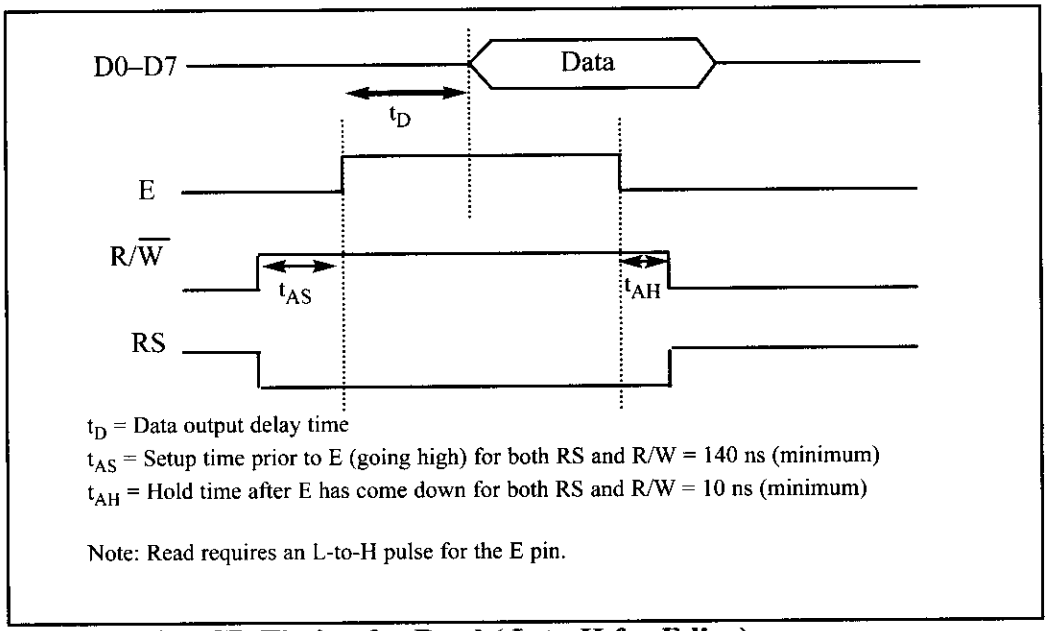
We use RS = 0 to read the busy flag bit to see if the LCD is ready to receive information. The busy flag is D7, and can be read when R/W = 1 and RS = 0, as follows: if R/W = 1, RS = 0. When D7 = 1 (busy flag = 1), the LCD is busy taking care of internal operations and will not accept any new information. When D7 = 0, the LCD is ready to receive new information.

This is shown in Program 12-2.

```
;Program 12-2: Check busy flag before sending
;data or command to LCD (See Fig. 12-2)
LCD_DATA    EQU PORTD            ;LCD data pins RD0-RD7
LCD_CTRL    EQU PORTB            ;LCD control pins
RS          EQU RB0              ;RS pin of LCD
RW          EQU RB1              ;R/W pin of LCD
EN          EQU RB2              ;E pin of LCD
            CLRF    TRISD        ;PORTD = Output
            CLRF    TRISB        ;PORTB = Output
            BCF     LCD_CTRL,EN  ;enable idle low
            CALL    LDELAY       ;long delay (250 ms) for power-up
            MOVLW   0x38         ;init. LCD 2 lines, 5x7 char
            CALL    COMMAND      ;issue command
            CALL    LDELAY       ;initialization hold
            MOVLW   0x0E         ;LCD on, cursor on
            CALL    COMMAND      ;issue command
            CALL    READY        ;Is LCD ready?
            MOVLW   0x01         ;clear LCD command
            CALL    COMMAND      ;issue command
            CALL    READY        ;Is LCD ready?
            MOVLW   0x06         ;shift cursor right
```

```
            CALL    COMMAND      ;issue command
            CALL    READY        ;Is LCD ready?
            MOVLW   0x86         ;cursor: line 1, pos. 6
            CALL    COMMAND      ;command subroutine
            CALL    READY        ;Is LCD ready?
            MOVLW   A'N'         ;display letter 'N'
            CALL    DATA_DISPLAY
            CALL    READY        ;Is LCD ready?
            MOVLW   A'O'         ;display letter 'O'
            CALL    DATA_DISPLAY
HERE        BRA     HERE         ;STAY HERE
;-----------------------------------------------
COMMAND     MOVWF   LCD_DATA     ;issue command code
            BCF     LCD_CTRL,RS  ;RS = 0 for command
            BCF     LCD_CTRL,RW  ;R/W = 0 for write
            BSF     LCD_CTRL,EN  ;E = 1 for high pulse
            CALL    SDELAY       ;make a wide En pulse
            BCF     LCD_CTRL,EN  ;E = 0 for H-to-L pulse
            RETURN
;-----------------------------------------------
DATA_DISPLAY MOVWF LCD_DATA   ;copy WREG to LCD DATA pin
            BSF     LCD_CTRL,RS  ;RS = 1 for data
            BCF     LCD_CTRL,RW  ;R/W = 0 for write
            BSF     LCD_CTRL,EN  ;E = 1 for high pulse
            CALL    SDELAY       ;make a wide En pulse
            BCF     LCD_CTRL,EN  ;E = 0 for H-to-L pulse
            RETURN
;-----------------------------------------------
READY       SETF    TRISD        ;make PORTD input port for LCD data
            BCF     LCD_CTRL,RS  ;RS = 0 access command reg
            BSF     LCD_CTRL,RW  ;R/W = 1 read command reg
;read command reg and check busy flag
BACK        BSF     LCD_CTRL,EN  ;E = 0 for L-to-H pulse
            CALL    SDELAY       ;make a wide En pulse
            BCF     LCD_CTRL,EN  ;E = 1 L-to-H pulse
            BTFSC   LCD_DATA,7   ;stay until busy flag = 0
            BRA     BACK
            CLRF    TRISD ;make PORTD output port for LCD data
            RETURN
;look in previous chapters for delay routines
            END
```

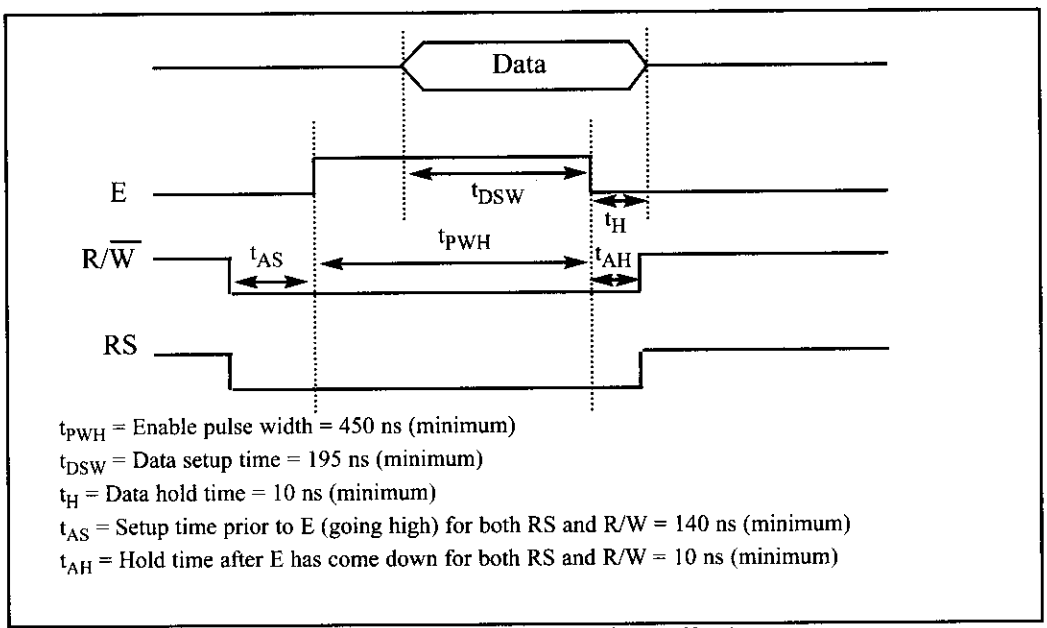
Notice in Program 12-2 that the busy flag is D7 of the command register. To read the command register, we make R/W = 1 and RS = 0, and a L-to-H pulse for the E pin will provide us the command register. After reading the command register, if bit D7 (the busy flag) is HIGH, the LCD is busy and no information (command or data) should be issued to it. Only when D7 = 0 can we send data or commands to the LCD. Notice that no time delays are used in this method because we are checking the busy flag before issuing commands or data to the LCD. Contrast the read and write timing for the LCD in Figures 12-3 and 12-4. Note that the E line is negative edge-triggered for the write while it is positive edge-triggered for the read.



t_D = Data output delay time
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 140 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

Note: Read requires an L-to-H pulse for the E pin.

Figure 12-3. LCD Timing for Read (L-to-H for E line)



t_{PWH} = Enable pulse width = 450 ns (minimum)
t_{DSW} = Data setup time = 195 ns (minimum)
t_H = Data hold time = 10 ns (minimum)
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 140 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

Figure 12-4. LCD Timing for Write (H-to-L for E line)

LCD data sheet

In the LCD, one can put data at any location. The following shows address locations and how they are accessed.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A	A	A	A	A	A	A

where AAAAAAA = 0000000 to 0100111 for line 1 and AAAAAAA = 1000000 to 1100111 for line 2. See Table 12-3.

The upper address range can go as high as 0100111 for the 40-character-wide LCD, while for the 20-character-wide LCD it goes up to 010011 (19 decimal = 10011 binary). Notice that the upper-range 0100111 (binary) = 39 decimal, which corresponds to locations 0 to 39 for the LCDs of 40x2 size.

From the above discussion we can get the addresses of cursor positions for various sizes of LCDs. See Figure 12-5 for the cursor addresses for common types of LCDs. Note that all the addresses are in hex. Table 12-4 provides a detailed list of LCD commands and instructions. (Table 12-2 is extracted from this table.)

Table 12-3: LCD Addressing

	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Line 1 (min)	1	0	0	0	0	0	0	0
Line 1 (max)	1	0	1	0	0	1	1	1
Line 2 (min)	1	1	0	0	0	0	0	0
Line 2 (max)	1	1	1	0	0	1	1	1

16 x 2 LCD	80	81	82	83	84	85	86 through 8F	
	C0	C1	C2	C3	C4	C5	C6 through CF	
20 x 1 LCD	80	81	82	83	through 93			
20 x 2 LCD	80	81	82	83	through 93			
	C0	C1	C2	C3	through D3			
20 x 4 LCD	80	81	82	83	through 93			
	C0	C1	C2	C3	through D3			
	94	95	96	97	through A7			
	D4	D5	D6	D7	through E7			
40 x 2 LCD	80	81	82	83	through A7			
	C0	C1	C2	C3	through E7			

Note: All data is in hex.

Figure 12-5. Cursor Addresses for Some LCDs

Table 12-4: List of LCD Instructions

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (Max)
Clear Display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DD RAM address 0 in address counter	1.64 ms
Return Home	0	0	0	0	0	0	0	0	1	-	Sets DD RAM address 0 as address counter. Also returns display being shifted to original position. DD RAM contents remain unchanged.	1.64 ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies shift of display. These operations are performed during data write and read.	40 µs
Display On/ Off Control	0	0	0	0	0	0	1	D	C	B	Sets On/Off of entire display (D), cursor On/Off (C), and blink of cursor position character (B).	40 µs
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	-	-	Moves cursor and shifts display without changing DD RAM contents.	40 µs
Function Set	0	0	0	0	1	DL	N	F	-	-	Sets interface data length (DL), number of display lines (L), and character font (F).	40 µs
Set CG RAM Address	0	0	0	1	AGC						Sets CG RAM address. CG RAM data is sent and received after this setting.	40 µs
Set DD RAM Address	0	0	1	ADD							Sets DD RAM address. DD RAM data is sent and received after this setting.	40 µs
Read Busy Flag & Address	0	1	BF	AC							Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	40 µs
Write Data CG or DD RAM	1	0	Write Data								Writes data into DD or CG RAM.	40 µs
Read Data CG or DD RAM	1	1	Read Data								Reads data from DD or CG RAM.	40 µs

Notes:

1. Execution times are maximum times when fcp or fosc is 250 kHz.
2. Execution time changes when frequency changes.
   (e.g., when fcp or fosc is 270 kHz: 40 µs × 250 / 270 = 37 µs.)
3. Abbreviations:

DD RAM	Display data RAM		
CG RAM	Character generator RAM		
ACC	CG RAM address		
ADD	DD RAM address, corresponds to cursor address		
AC	Address counter used for both DD and CG RAM addresses.		
I/D = 1	Increment	I/D = 0	Decrement
S = 1	Accompanies display shift		
S/C = 1	Display shift;	S/C = 0	Cursor move
R/L = 1	Shift to the right;	R/L = 0	Shift to the left
DL = 1	8 bits, DL = 0: 4 bits		
N = 1	1 line, N = 0: 1 line		
F = 1	5 × 10 dots, F = 0: 5 × 7 dots		
BF = 1	Internal operation;	BF = 0	Can accept instruction

Optrex is one of the largest manufacturers of LCDs. You can obtain datasheets from their Web site, http://www.optrex.com.

The LCDs can be purchased from the following Web sites:
http://www.digikey.com
http://www.jameco.com
http://www.elexp.com
http://www.bgmicro.com

Sending information to LCD using the TBLRD instruction

Program 12-3 shows how to use the TBLRD instruction to send data and commands to an LCD.

For a PIC18 C version of LCD programming see Program 12-1C and Program 12-2C.

```
;Program 12-3: Using TableRead
;PORTD = D0-D7, RB0 = RS, RB1 = R/W, RB2 = E pins
LCD_DATA    EQU PORTD          ;LCD data pins RD0-RD7
LCD_CTRL    EQU PORTB          ;LCD control pins
RS          EQU RB0            ;RS pin of LCD
RW          EQU RB1            ;R/W pin of LCD
EN          EQU RB2            ;E pin of LCD
            CLRF   TRISD       ;PORTD = Output
            CLRF   TRISB       ;PORTB = Output
            BCF    LCD_CTRL,EN ;enable idle low
            CALL   LDELAY      ;long delay (250 ms) for power-up
            MOVLW  upper(MYCOM)
            MOVWF  TBLPTRU
            MOVLW  high(MYCOM)
            MOVWF  TBLPTRH
            MOVLW  low(MYCOM)
            MOVWF  TBLPTRL
C1          TBLRD*+
            MOVF TABLAT,W      ;give it to WREG
            IORLW 0x0          ;Is it the end of command?
            BZ     SEND_DAT    ;if yes then go to display data
            CALL   COMNWRT     ;call command subroutine
            CALL   DELAY       ;give LCD some time
            BRA    C1
```

```
SEND_DAT    MOVLW   upper(MYDATA)
            MOVWF   TBLPTRU
            MOVLW   high(MYDATA)
            MOVWF   TBLPTRH
            MOVLW   low(MYDATA)
            MOVWF   TBLPTRL
DT1         TBLRD*+
            MOVF TABLAT,W           ;give it to WREG
            IORLW   0x0         ;Is it the end of data string?
            BZ      OVER            ;if yes then exit
            CALL    DATAWRT         ;call DATA subroutine
            CALL    DELAY           ;give LCD some time
            BRA     DT1
OVER        BRA     OVER            ;stay here
COMNWRT                             ;send command to LCD
            MOVWF   LCD_DATA        ;copy WREG to LCD DATA pin
            BCF     LCD_CTRL,RS     ;RS = 0 for command
            BCF     LCD_CTRL,RW     ;R/W = 0 for write
            BSF     LCD_CTRL,EN     ;E = 1 for high pulse
            CALL    SDELAY          ;make a wide En pulse
            BCF     LCD_CTRL,EN     ;E = 0 for H-to-L pulse
            RETURN
DATAWRT                             ;write data to LCD
            MOVWF   LCD_DATA        ;copy WREG to LCD DATA pin
            BSF     LCD_CTRL,RS     ;RS = 1 for data
            BCF     LCD_CTRL,RW     ;R/W = 0 for write
            BSF     LCD_CTRL,EN     ;E = 1 for high pulse
            CALL    SDELAY          ;make a wide En pulse
            BCF     LCD_CTRL,EN     ;E = 0 for H-to-L pulse
            RETURN
            ORG     500H
MYCOM       DB 0x38,0x0E,0x01,0x06,0x84,0;commands and null
MYDATA      DB "HELLO",0            ;data and null
;look in previous chapters for delay routines
            END
```

This C18 program sends letters 'M', 'D', and 'E' to the LCD using delays.

```
//Program 12-1C: This is the C version of Program 12-1.
#include <P18F4580.h>
#define ldata PORTD             //PORTD = LCD data pins (Fig. 12-2)
#define rs PORTBbits.RB0        //rs = PORTB.0
#define rw PORTBbits.RB1        //rw = PORTB.1
#define en PORTBbits.RB2        //en = PORTB.2

void main()
  {
    TRISD = 0;                  //both ports B and D as output
    TRISB = 0;
    en = 0;                     //enable idle low
    MSDelay(250);
    lcdcmd(0x38);               //init. LCD 2 lines, 5x7 matrix
    MSDelay(250);
    lcdcmd(0x0E);               //display on, cursor on
```

```
    MSDelay(15);
    lcdcmd(0x01);              //clear LCD
    MSDelay(15);
    lcdcmd(0x06);              //shift cursor right
    MSDelay(15);
    lcdcmd(0x86);              //line 1, position 6
    MSDelay(15);
    lcddata('M');              //display letter 'M'
    MSDelay(15);
    lcddata('D');              //display letter 'D'
    MSDelay(15);
    lcddata('E');              //display letter 'E'
  }

void lcdcmd(unsigned char value)
  {
    ldata = value;             //put the value on the pins
    rs = 0;
    rw = 0;
    en = 1;                    //strobe the enable pin
    MSDelay(1);
    en = 0;
  }

void lcddata(unsigned char value)
  {
    ldata = value;             //put the value on the pins
    rs = 1;
    rw = 0;
    en = 1;                    //strobe the enable pin
    MSDelay(1);
    en = 0;
  }

void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<135;j++);
  }
```

The following is the C version of Program 12-2, using the busy flag method.

```
//Program 12-2C. C version of Program 12-2
#include <P18F458.h>
#define ldata PORTD   //PORTD = LCD data pins (Fig. 12-2)
#define rs PORTBbits.RB0    //rs = PORTB.0
#define rw PORTBbits.RB1    //rw = PORTB.1
#define en PORTBbits.RB2    //en = PORTB.2
#define busy PORTDbits.RD7 //busy = PORTD.7

void main()
  {
    TRISD = 0;                 //both ports B and D as output
```

```
    TRISB = 0;
    en = 0;                 //enable idle low
    MSDelay(250);           //long delay
    lcdcmd(0x38);
    MSDelay(250);           //long delay
    lcdcmd(0x0E);
    lcdready();             //check the LCD busy flag
    lcdcmd(0x01);
    lcdready();             //check the LCD busy flag
    lcdcmd(0x06);
    lcdready();             //check the LCD busy flag
    lcdcmd(0x86);           //line 1, position 6
    lcdready();             //check the LCD busy flag
    lcddata('M');
    lcdready();             //check the LCD busy flag
    lcddata('D');
    lcdready();             //check the LCD busy flag
    lcddata('E');
  }

void lcdcmd(unsigned char value)
  {
    ldata = value;          //put the value on the pins
    rs = 0;
    rw = 0;
    en = 1;                 //strobe the enable pin
    MSDelay(1);
    en = 0;
  }

void lcddata(unsigned char value)
  {
    ldata = value;          //put the value on the pins
    rs = 1;
    rw = 0;
    en = 1;                 //strobe the enable pin
    MSDelay(1);
    en = 0;
  }

void lcdready()
  {
    TRISD = 0xFF;           //make PORTD an input
    rs = 0;
    rw = 1;
    do                      //wait here for busy flag
     {
       en = 1;              //strobe the enable pin
       MSDelay(1);
       en = 0;
     }while(busy==1);
     TRISD = 0;
   }

void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<135;j++);
  }
```

//Program 12-3C: C version of Program 12-3 Displaying Data in ROM

```
#include <P18F458.h>
#define ldata PORTD     //PORTD = LCD data pins (Fig. 12-2)
#define rs PORTBbits.RB0            //rs = PORTB.0
#define rw PORTBbits.RB1            //rw = PORTB.1
#define en PORTBbits.RB2            //en = PORTB.2

#pragma romdata mycom = 0x300       //command at ROM addr 0x300
far rom const char mycom[] = {0x0E,0x01,0x06,0x84};

#pragma romdata mydata = 0x320      //data at ROM addr 0x320
far rom const char mydata[] = "HELLO";

void main()
  {
    unsigned char z=0;
    TRISD = 0;                  //both ports B and D as output
    TRISB = 0;
    en = 0;                     //enable idle low
    MSDelay(250);
    lcdcmd(0x38);
    MSDelay(250);
    //send out the commands
    for(;z<4;z++)
      {
        lcdcmd(mycom[z]);
        MSDelay(15);
      }
    //send out the data
    for(z=0;z<5;z++)
      {
        lcddata(mydata[z]);
        MSDelay(15);
      }
    while(1); //infinite loop
  }

void lcdcmd(unsigned char value)
  {
    ldata = value;              //put the value on the pins
    rs = 0;
    rw = 0;
    en = 1;                     //strobe the enable pin
    MSDelay(1);
    en = 0;
  }

void lcddata(unsigned char value)
  {
    ldata = value;              //put the value on the pins
    rs = 1;
    rw = 0;
```

```
    en = 1;                    //strobe the enable pin
    MSDelay(1);
    en = 0;
  }
void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<135;j++);
  }
```

Review Questions

1. The RS pin is an _____ (input, output) pin for the LCD.
2. The E pin is an _____ (input, output) pin for the LCD.
3. The E pin requires an _____ (H-to-L, L-to-H) pulse to latch in information at the data pins of the LCD.
4. For the LCD to recognize information at the data pins as data, RS must be set to _____ (HIGH, LOW).
5. Give the command codes for line 1, first character, and line 2, first character.

SECTION 12.2: KEYBOARD INTERFACING

Keyboards and LCDs are the most widely used input/output devices and a basic understanding of them is essential. In this section, we first discuss keyboard fundamentals, along with key press detection and key identification mechanisms. Then we show how a keyboard is interfaced to a PIC18.

Interfacing the keyboard to the PIC18

At the lowest level, keyboards are organized in a matrix of rows and columns. The CPU accesses both rows and columns through ports; therefore, with two 8-bit ports, an 8 × 8 matrix of keys can be connected to a microprocessor. When a key is pressed, a row and a column make a contact; otherwise, there is no connection between rows and columns. In IBM PC keyboards, a single microcontroller takes care of hardware and software interfacing of the keyboard. In such systems, programs stored in the ROM of the microcontroller scan the keys continuously, identify which one has been activated, and present it to the motherboard. See Example 12-3. In programming for keypad interfacing we must have two processes: (a) key press detection, and (b) key identification. There are two ways by which the PIC18 can perform key press detection: (1) the interrupt method, and (2) the scanning method. In the PIC18, the PORTB-Change interrupt can be used to implement the interrupt-based key press detection. Next we explain the interrupt method.

Interrupt method of key press detection

Figure 12-6 shows a 4 × 4 matrix keypad connected to PORTB. The rows are connected to PORTB.Low (RB3–RB0) and the columns are connected to PORTB.High (RB7–RB4), which is the PORTB-Change interrupt. As we dis-

Example 12-3

From Figure 12-6, identify the row and column of the pressed key for the following:
RB3– RB0 = 1110 for the row, RB7–RB4 = 1011 for the column
Solution:

From Figure 12-6, the row and column can be used to identify the key. The row belongs to RB0 and the column belongs to RB6; therefore, key number 2 was pressed.

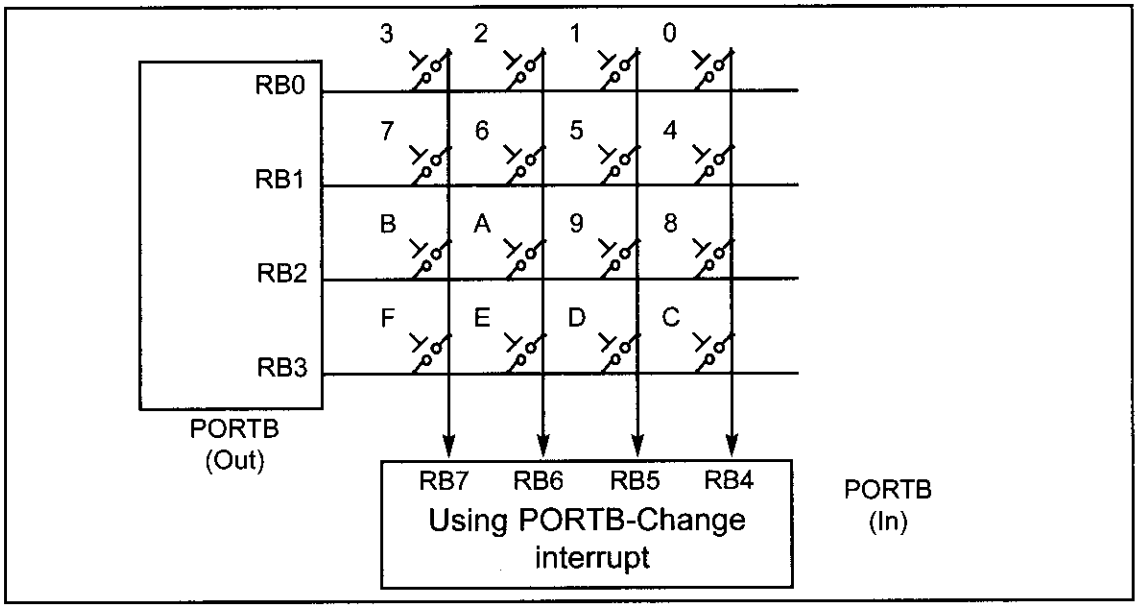


Figure 12-6. Matrix Keyboard Connection to Ports

cussed in Chapter 11, any changes on the RB7–RB4 pins will cause an interrupt indicating a key press. Examine Program 12-4, which goes through the following stages:

1. To make sure that the preceeding key has been released, 0s are output to all rows at once, and the columns are read and checked repeatedly until all the columns are HIGH. When all columns are found to be HIGH, the program waits for a short amount of time before it goes to the next stage of waiting for a key to be pressed.

2. To see if any key is pressed, the columns are connected to the PORTB-Change interrupt. Therefore, any key press will cause an interrupt and the microcontroller will execute the ISR. The ISR must do two things: (a) ensure that the first key press detection was not erroneous due to spike noise, and (b) wait 20 ms to prevent the same key press from being interpreted as multiple key presses. See Figure 12-8 for keyboard debounce.

3. To detect which row the key press belongs to, the microcontroller grounds one row at a time, reading the columns each time. If it finds that all columns are HIGH, this means that the key press cannot belong to that row; therefore, it grounds the next row and continues until it finds the row the key press belongs

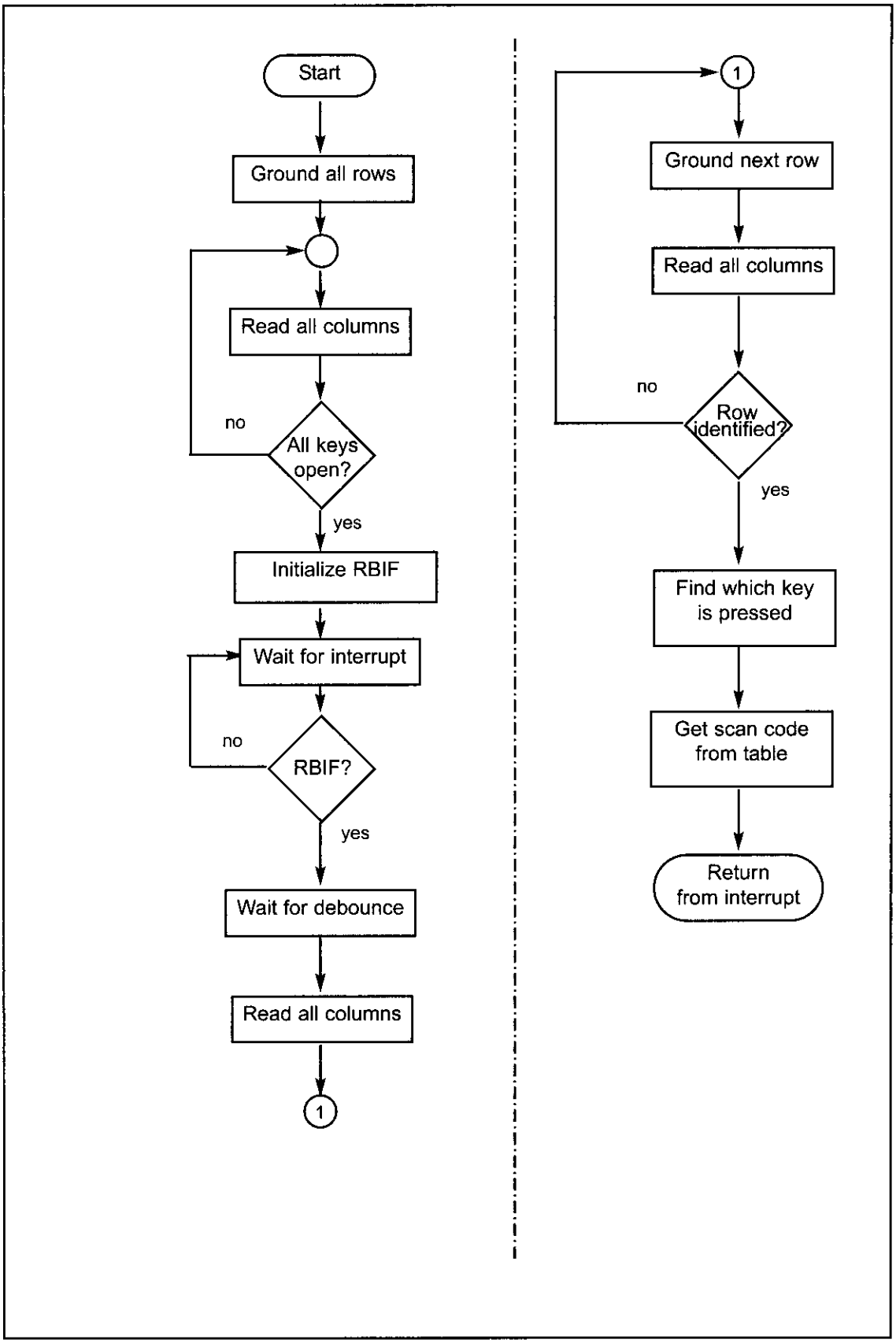


Figure 12-7. Flowchart for Program 12-4

to. Upon finding the row that the key press belongs to, it sets up the starting address for the look-up table holding the scan codes (or the ASCII value) for that row and goes to the next stage to identify the key.

4. To identify the key press, the microcontroller rotates the column bits, one bit at a time, into the carry flag and checks to see if it is LOW. Upon finding the zero, it pulls out the ASCII code for that key from the look-up table; otherwise, it increments the pointer to point to the next element of the look-up table. Figure 12-7 flowcharts this process.

The look-up table method shown in Program 12-4 can be modified to work with any matrix up to 8 × 4. Figure 12-7 provides the flowchart for Program 12-4 for scanning and identifying the pressed key.

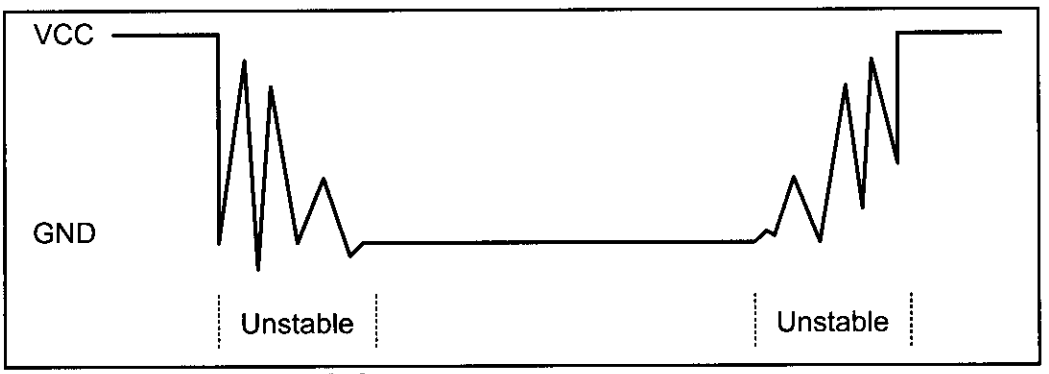


Figure 12-8. Keyboard Debounce

Examine Program 12-4. Notice in that program that the interrupt detects the key press. Then it is the job of the ISR to identify to which key the key press belongs (key identification). Program 12-4C is a C18 version of Program 12-4. In the Assembly version (12-4), the character is placed on PORTD, while in the C18 version (12-4C), it is sent to the serial port to be displayed on the monitor.

Program 12-4: This program waits for a key press on PORTB, then places the character on PORTD. We assume the following for this program:
RB3–RB0 connected to rows
RB7–RB4 connected to columns

```
D15mH       EQU   D'100'       ;15 ms delay high byte of value
D15mL       EQU   D'255'       ;low byte of value
COL         EQU   0x08         ;holds the column found
DR15mH      EQU   0x09         ;registers for 15 ms delay
DR15mL      EQU   0x0A         ;
;-----------------------------------------
            ORG   0x000000
RESET_ISR   GOTO  MAIN         ;jump over interrupt table
            ORG   0x000008
HI_ISR      BTFSC INTCON,RBIF ;Was it a PORTB change?
            BRA   RBIF_ISR     ;yes then go to ISR
            RETFIE             ;else return
```

```
;------------program for initialization
MAIN        CLRF  TRISD        ;make PORTD output port
            BCF   INTCON2,RBPU;enable PORTB pull-up resistors
            MOVLW 0xF0         ;make PORTB high input ports
            MOVWF TRISB        ;make PORTB low output ports
            MOVWF PORTB        ;ground all rows
KEYOPEN     CPFSEQ PORTB       ;are all keys open
            GOTO  KEYOPEN      ;wait until keypad ready
            MOVLW upper(KCODE0)
            MOVWF TBLPTRU      ;load upper byte of TBLPTR
            MOVLW high(KCODE0)
            MOVWF TBLPTRH      ;load high byte of TBLPTR
            BSF   INTCON,RBIE ;enable PORTB change interrupt
            BSF   INTCON,GIE  ;enable all interrupts globally
LOOP        GOTO  LOOP         ;wait for key press
;------------key identification ISR
RBIF_ISR    CALL  DELAY        ;wait for debounce
            MOVFF PORTB,COL    ;get the column of key press
            MOVLW 0xFE
            MOVWF PORTB        ;ground row 0
            CPFSEQ PORTB       ;Did PORTB change?
            BRA   ROW0         ;yes then row 0
            MOVLW 0xFC
            MOVWF PORTB        ;ground row 1
            CPFSEQ PORTB       ;Did PORTB change?
            BRA   ROW1         ;yes then row 1
            MOVLW 0xFB
            MOVWF PORTB        ;ground row 2
            CPFSEQ PORTB       ;Did PORTB change?
            BRA   ROW2         ;yes then row 2
            MOVLW 0xF7
            MOVWF PORTB        ;ground row 3
            CPFSEQ PORTB       ;Did PORTB change?
            BRA   ROW3         ;yes then row 3
            GOTO  BAD_RBIF     ;no then key press too short
ROW0        MOVLW low(KCODE0) ;set TBLPTR = start of row 0
            BRA   FIND         ;find the column
ROW1        MOVLW low(KCODE1) ;set TBLPTR = start of row 1
            BRA   FIND         ;find the column
ROW2        MOVLW low(KCODE2) ;set TBLPTR = start of row 2
            BRA   FIND         ;find the column
ROW3        MOVLW low(KCODE3) ;set TBLPTR = start of row 3
FIND        MOVWF TBLPTRL      ;load low byte of TBLPTR
            MOVLW 0xF0
            XORWF COL          ;invert high nibble
            SWAPF COL,F        ;bring to low nibble
AGAIN       RRCF  COL          ;rotate to find column
            BC    MATCH        ;column found, get the ASCII code
            INCF  TBLPTRL      ;else point to next col. address
```

```
            BRA   AGAIN        ;keep searching
MATCH       TBLRD*+            ;get ASCII code from table
            MOVFF TABLAT,PORTD;display pressed key on PORTD
WAIT        MOVLW 0xF0
            MOVWF PORTB        ;reset PORTB
            CPFSEQ PORTB       ;Did PORTB change?
            BRA   WAIT         ;yes then wait for key release
            BCF   INTCON,RBIF ;clear PORTB, change flag
            RETFIE             ;return and wait for key press
BAD_RBIF    MOVLW 0x00         ;return null
            GOTO  WAIT         ;wait for key release
;-------------delay
DELAY:      MOVLW D15mH        ;high byte of delay
            MOVWF DR15mH       ;store in register
D2:         MOVLW D15mL        ;low byte of delay
            MOVWF DR15mL       ;store in register
D1:         DECF  DR15mL,F     ;stay until DR15mL becomes 0
            BNZ   D1
            DECF  DR15mH,F     ;loop until all DR15m = 0x0000
            BNZ   D2
            RETURN
;-------------key scancode look-up table
            ORG   300H
KCODE0:     DB    '0','1','2','3'   ;ROW 0
KCODE1:     DB    '4','5','6','7'   ;ROW 1
KCODE2:     DB    '8','9','A','B'   ;ROW 2
KCODE3:     DB    'C','D','E','F'   ;ROW 3
            END
```

Program 12-4C shows keypad programming in PIC18 C.

Program 12-4C: This C18 program reads the keypad and sends the result to the serial port. We assume the following for this program.
RB0–RB3 connected to rows
RB4–RB7 connected to columns
Serial port is set for 9600 baud (10 MHz XTAL), 8-bit mode, and 1 stop bit.

```
#include <p18f458.h>
void SerTX(unsigned char x);
void RBIF_ISR(void);
void MSDelay(unsigned int millisecs);
unsigned char keypad[4][4] = {'0','1','2','3',
                              '4','5','6','7',
                              '8','9','A','B',
                              'C','D','E','F'};

#pragma code My_HiPrio_Int =0x0008 //high-priority interrupt
void My_HiPrio_Int (void)
{
  _asm
    GOTO chk_isr
  _endasm
```

```
}
#pragma code

#pragma interrupt chk_isr     //which ISR
void chk_isr (void)
{
  if (INTCONbits.RBIF==1)     //RBIF caused interrupt?
  RBIF_ISR( );                //yes go to RBIF_ISR
}
#pragma code

void main()
{
  TRISD=0;                    //make PORTD output port
  INTCON2bits.RBPU=0;         //enable PORTB pull-up resistors
  TRISB=0xF0;           //PORTB low as output and high as input
  PORTB=0xF0;                 //clear PORTB low
  while(PORTB!=0xF0);         //wait until key not pressed
  TXSTA=0x20;                 //choose low baud rate, 8-bit
  SPBRG=15;                   //9600 baud rate, XTAL = 10 MHz
  TXSTAbits.TXEN=1;           //enable transmit
  RCSTAbits.SPEN=1;           //enable serial port
  INTCONbits.RBIE=1;          //enable PORTB interrupt on change
  INTCONbits.GIE=1;           //enable interrupts globally
  while(1);                   //wait until key press
}
void RBIF_ISR(void)           //finds the key pressed
{
  unsigned char temp,COL=0,ROW=4;
  MSDelay(15);
  temp = PORTB;               //get column
  temp ^= 0xF0;               //invert high nibble
  if(!temp) return;           //if false alarm return
  while(temp<<=1) COL++;      //find the column
  PORTB = 0xFE;               //ground row 0
  if(PORTB != 0xFE)           //Did high nibble change?
    ROW = 0;                  //yes then row 0
  else {                      //try next row
    PORTB = 0xFD;             //ground row 1
    if(PORTB != 0xFD)         //Did high nibble change?
      ROW = 1;                //yes then row 1
    else {                    //try next row
      PORTB = 0xFB;           //ground row 2
      if(PORTB != 0xFB)       //Did high nibble change?
        ROW = 2;              //yes then row 2
      else {                  //try last row
        PORTB = 0xF7;         //ground row 3
          if(PORTB != 0xF7)   //Did high nibble change?
            ROW = 3;          //yes then row 3
      }
    }
  }
  if(ROW<4)                   //Did we find a valid row?
    SerTX(keypad[ROW][COL]);  //then send character
  while(PORTB!=0xF0) PORTB=0xF0;    //wait for release
  INTCONbits.RBIF=0;          //reset flag
}
void SerTX(unsigned char x)   //sends character
{
  while(PIR1bits.TXIF!=1);    //wait until ready
  TXREG=x;                    //send character out serial port
}
```

```
void MSDelay(unsigned int millisecs)
{
  unsigned int i, j;
  for(i=0;i<millisecs;i++)
    for(j=0;j<135;j++);
}
```

Scanning method for key press detection

Another method for key press detection is by scanning. In this method, to detect a pressed key, the microcontroller grounds all rows by providing 0 to the output latch, then it reads the columns. If the data read from the columns are equal to 1111, no key has been pressed and the process continues until a key press is detected. If one of the column bits has a zero, however, this means that a key press has occurred. After a key press is detected, the microcontroller will go through the process of identifying the key. Starting with the top row, the microcontroller grounds it by providing a LOW to the first row only; then it reads the columns. If the data read is all 1s, no key in that row is activated and the process is moved to the next row. It grounds the next row, reads the columns, and checks for any zero. This process continues until the row is identified. After identification of the row in which the key has been pressed, the next task is to find out which column the pressed key belongs to. This should be easy since the microcontroller knows at any time which row and column are being accessed. Figure 12-9 shows the flowchart for this method. The program implementation is left to the reader.

Some IC chips, such as National Semiconductor's MM74C923, incorporate keyboard scanning and decoding all in one chip. Such chips use combinations of counters and logic gates (no microcontroller) to implement the underlying concepts presented in this section.

Review Questions

1. True or false. To see if any key is pressed, all rows are grounded.
2. If RB7–RB4 = 0111 is the data read from the columns, which column does the pressed key belong to?
3. True or false. Key press detection and key identification require two different processes.
4. In Figure 12-6, if the rows are RB3–RB0 = 1110 and the columns are RB7–RB4 = 1110, which key is pressed?
5. True or false. To identify the pressed key, one row at a time is grounded.

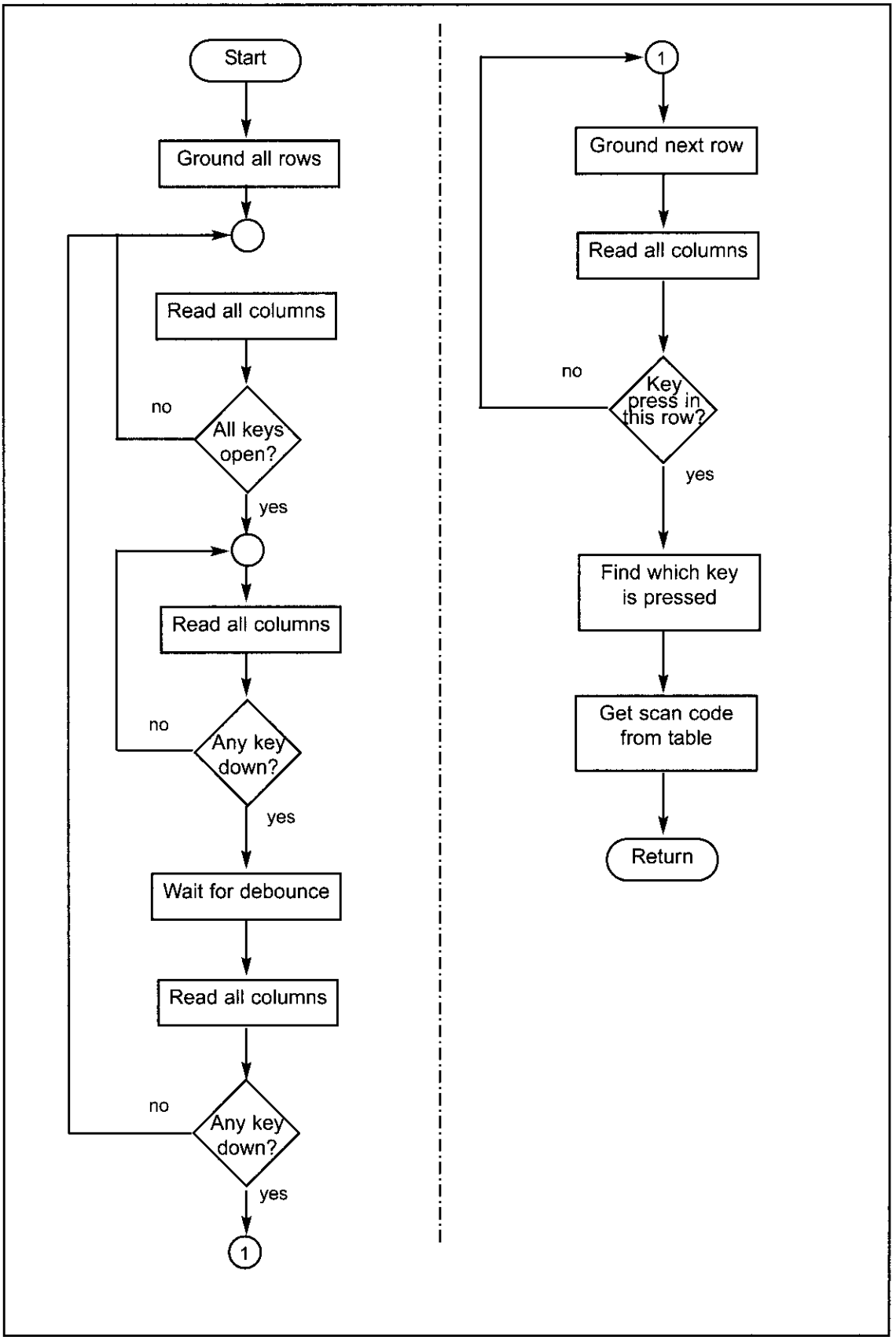


Figure 12-9. Flowchart of Scanning Method for Key Press Detection

SUMMARY

This chapter showed how to interface real-world devices such as LCDs and keypads to the PIC18. First, we described the operation modes of LCDs, then described how to program the LCD by sending data or commands to it via its interface to the PIC18.

Keyboards are one of the most widely used input devices for PIC18 projects. This chapter also described the operation of keyboards, including key press detection and key identification mechanisms. Then the PIC18 was shown interfacing with a keyboard. PIC18 programs were written to return the ASCII code for the pressed key.

PROBLEMS

SECTION 12.1: LCD INTERFACING

1. The LCD discussed in this section has _____ (4, 8) data pins.
2. Describe the function of pins E, R/W, and RS in the LCD.
3. What is the difference between the V_{CC} and V_{EE} pins on the LCD?
4. "Clear LCD" is a _____ (command code, data item) and its value is ___ hex.
5. What is the hex value of the command code for "display on, cursor on"?
6. Give the state of RS, E, and R/W when sending a command code to the LCD.
7. Give the state of RS, E, and R/W when sending data character 'Z' to the LCD.
8. Which of the following is needed on the E pin in order for a command code (or data) to be latched in by the LCD?
   (a) H-to-L pulse (b) L-to-H pulse
9. True or false. For the above to work, the value of the command code (data) must already be at the D0–D7 pins.
10. There are two methods of sending streams of characters to the LCD: (1) checking the busy flag, or (2) putting some time delay between sending each character without checking the busy flag. Explain the difference and the advantages and disadvantages of each method. Also explain how we monitor the busy flag.
11. For a 16 × 2 LCD, the location of the last character of line 1 is 8FH (its command code). Show how this value was calculated.
12. For a 16 × 2 LCD, the location of the first character of line 2 is C0H (its command code). Show how this value was calculated.
13. For a 20 × 2 LCD, the location of the last character of line 2 is 93H (its command code). Show how this value was calculated.
14. For a 20 × 2 LCD, the location of the third character of line 2 is C2H (its command code). Show how this value was calculated.
15. For a 40 × 2 LCD, the location of the last character of line 1 is A7H (its command code). Show how this value was calculated.
16. For a 40 × 2 LCD, the location of the last character of line 2 is E7H (its command code). Show how this value was calculated.

17. Show the value (in hex) for the command code for the 10th location, line 1 on a 20 × 2 LCD. Show calculations.
18. Show the value (in hex) for the command code for the 20th location, line 2 on a 40 × 2 LCD. Show calculations.
19. Rewrite the COMNWRT subroutine. Assume connections RC4 = RS, RC5 = R/W, RC6 = E.
20. Repeat Problem 19 for the data write subroutine. Send the string "Hello" to the LCD by checking the busy flag. Use the instruction TBLRD.

SECTION 12.2: KEYBOARD INTERFACING

21. In reading the columns of a keyboard matrix, if no key is pressed we should get all _____ (1s, 0s).
22. In Program 12-4, to detect the key press, which of the following is performed?
    (a) PORTB-Change interrupt (b) grounding one row at time
23. In Figure 12-6, to identify the key pressed, which of the following is grounded?
    (a) all rows (b) one row at time (c) both (a) and (b)
24. For Figure 12-6, indicate the key press for RB7–RB4 = 0111, RB3–RB0 = 1110.
25. Indicate an advantage and a disadvantage of using an IC chip instead of a microcontroller for keyboard scanning and decoding.
26. What is the best compromise for the answer to Problem 25?

ANSWERS TO REVIEW QUESTIONS

SECTION 12.1: LCD INTERFACING

1. Input
2. Input
3. H-to-L
4. HIGH
5. 80H and C0H

SECTION 12.2: KEYBOARD INTERFACING

1. True
2. Column 3
3. True
4. 0
5. True

CHAPTER 13

ADC, DAC, AND SENSOR INTERFACING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Discuss the ADC (analog-to-digital converter) section of the PIC18 chip**
- **>> Interface temperature sensors to the PIC18**
- **>> Explain the process of data acquisition using ADC**
- **>> Describe factors to consider in selecting an ADC chip**
- **>> Program the PIC18's ADC in C and Assembly**
- **>> Describe the basic operation of a DAC (digital-to-analog converter) chip**
- **>> Interface a DAC chip to the PIC18**
- **>> Program a DAC chip to produce a sine wave on an oscilloscope**
- **>> Program DAC chips in PIC18 C and Assembly**
- **>> Explain the function of precision IC temperature sensors**
- **>> Describe signal conditioning and its role in data acquisition**

This chapter explores more real-world devices such as ADCs (analog-to-digital converters), DACs (digital-to-analog converters), and sensors. We will also explain how to interface the PIC18 to these devices. In Section 13.1, we describe analog-to-digital converter (ADC) chips. We will program the ADC portion of the PIC18 chip in Section 13.2. The characteristics of DAC chips are discussed in Section 13.3. In Section 13.4, we show the interfacing of sensors and discuss the issue of signal conditioning.

SECTION 13.1: ADC CHARACTERISTICS

This section will explore ADC programming in PIC18 chips. First, we describe some general aspects of the ADC itself, then show how to program the ADC portion of the PIC18 in both Assembly and C.

ADC devices

Analog-to-digital converters are among the most widely used devices for data acquisition. Digital computers use binary (discrete) values, but in the physical world everything is analog (continuous). Temperature, pressure (wind or liquid), humidity, and velocity are a few examples of physical quantities that we deal with every day. A physical quantity is converted to electrical (voltage, current) signals using a device called a *transducer*. Transducers are also referred to as *sensors*. Sensors for temperature, velocity, pressure, light, and many other natural quantities produce an output that is voltage (or current). Therefore, we need an analog-to-digital converter to translate the analog signals to digital numbers so that the microcontroller can read and process them. See Figures 13-1 and 13-2.

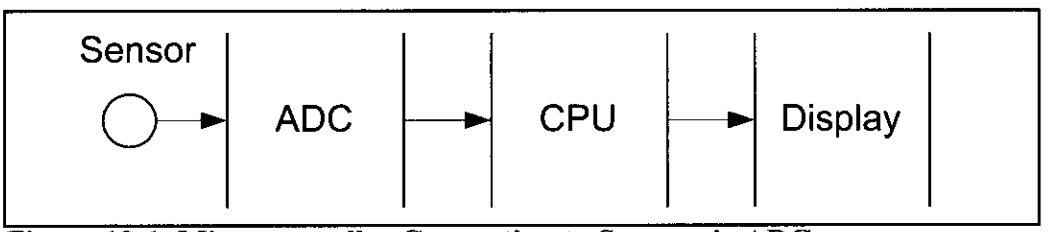


Figure 13-1. Microcontroller Connection to Sensor via ADC

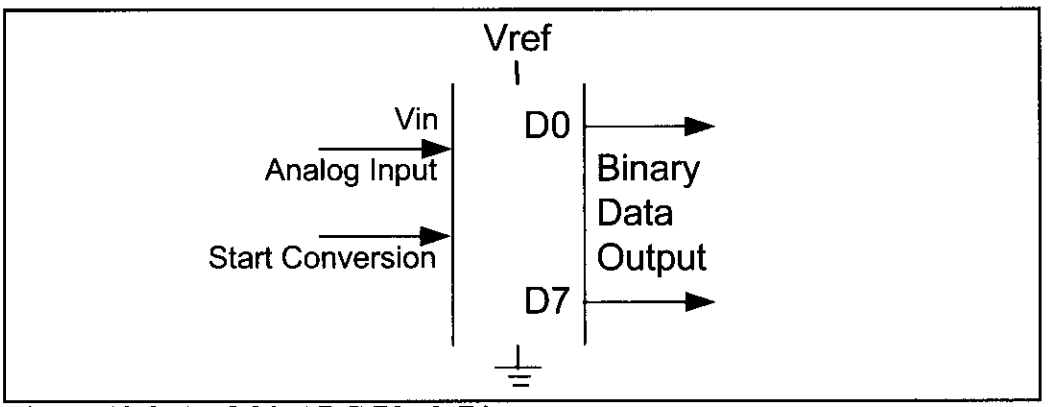


Figure 13-2. An 8-bit ADC Block Diagram

Table 13-1: Resolution versus Step Size for ADC (Vref = 5 V)

n-bit	Number of steps	Step size (mV)
8	256	5/256 = 19.53
10	1,024	5/1,024 = 4.88
12	4,096	5/4,096 = 1.2
16	65,536	5/65,536 = 0.076

Notes: V_{CC} = 5 V
Step size (resolution) is the smallest change that can be discerned by an ADC.

Some of the major characteristics of the ADC are as follows:

Resolution

ADC has *n*-bit resolution, where *n* can be 8, 10, 12, 16, or even 24 bits. The higher-resolution ADC provides a smaller step size, where *step size* is the smallest change that can be discerned by an ADC. Some widely used resolutions for ADCs are shown in Table 13-1. Although the resolution of an ADC chip is decided at the time of its design and cannot be changed, we can control the step size with the help of what is called Vref. This is discussed below.

Conversion time

In addition to resolution, conversion time is another major factor in judging an ADC. *Conversion time* is defined as the time it takes the ADC to convert the analog input to a digital (binary) number. The conversion time is dictated by the clock source connected to the ADC in addition to the method used for data conversion and technology used in the fabrication of the ADC chip such as MOS or TTL technology.

V_{ref}

V_{ref} is an input voltage used for the reference voltage. The voltage connected to this pin, along with the resolution of the ADC chip, dictate the step size. For an 8-bit ADC, the step size is $V_{ref}/256$ because it is an 8-bit ADC, and 2 to the power of 8 gives us 256 steps. See Table 13-1. For example, if the analog input range needs to be 0 to 4 volts, V_{ref} is connected to 4 volts. That gives 4 V/256 = 15.62 mV for the step size of an 8-bit ADC. In another case, if we need a step size of 10 mV for an 8-bit ADC, then V_{ref} = 2.56 V, because 2.56 V/256 = 10 mV. For

Table 13-2: V_{ref} Relation to V_{in} Range for an 8-bit ADC

V_{ref} (V)	V_{in} (V)	Step Size (mV)
5.00	0 to 5	5/256 = 19.53
4.0	0 to 4	4/256 = 15.62
3.0	0 to 3	3/256 = 11.71
2.56	0 to 2.56	2.56/256 = 10
2.0	0 to 2	2/256 = 7.81
1.28	0 to 1.28	1.28/256 = 5
1	0 to 1	1/256 = 3.90

Table 13-3: V_{ref} Relation to V_{in} Range for an 10-bit ADC

V_{ref} (V)	V_{in} (V)	Step Size (mV)
5.00	0 to 5	5/1,024 = 4.88
4.096	0 to 4.096	4.096/1,024 = 4
3.0	0 to 3	3/1,024 = 2.93
2.56	0 to 2.56	2.56/1,024 = 2.5
2.048	0 to 2.048	2.048/1,024 = 2
1.28	0 to 1.28	1/1,024 = 1.25
1.024	0 to 1.024	1.024/1,024 = 1

the 10-bit ADC, if the Vref = 5V, then the step size is 4.88 mV as shown in Table 13-1. Tables 13-2 and 13-3 show the relationship between the Vref and step size for the 8- and 10-bit ADCs, respectively. In some applications, we need the differential reference voltage where Vref = Vref (+) – Vref (–). Often the Vref (–) pin is connected to ground and the Vref (+) pin is used as the Vref.

Digital data output

In an 8-bit ADC we have an 8-bit digital data output of D0–D7 while in the 10-bit ADC the data output is D0–D9. To calculate the output voltage, we use the following formula:

$$D_{out} = \frac{V_{in}}{step\ size}$$

where Dout = digital data output (in decimal), Vin = analog input voltage, and step size (resolution) is the smallest change, which is Vref/256 for an 8-bit ADC. See Example 13-1. This data is brought out of the ADC chip either one bit at a time (serially), or in one chunk, using a parallel line of outputs. This is discussed next.

Example 13-1

For an 8-bit ADC, we have V_{ref} = 2.56 V. Calculate the D0–D7 output if the analog input is: (a) 1.7 V, and (b) 2.1 V.

Solution:

Because the step size is 2.56/256 = 10 mV, we have the following:

(a) D_{out} = 1.7 V/10 mV = 170 in decimal, which gives us 10101011 in binary for D7–D0.

(b) D_{out} = 2.1 V/10 mV = 210 in decimal, which gives us 11010010 in binary for D7–D0.

Parallel versus serial ADC

The ADC chips are either parallel or serial. In parallel ADC, we have 8 or more pins dedicated to bringing out the binary data, but in serial ADC we have only one pin for data out. That means that inside the serial ADC, there is a paral-

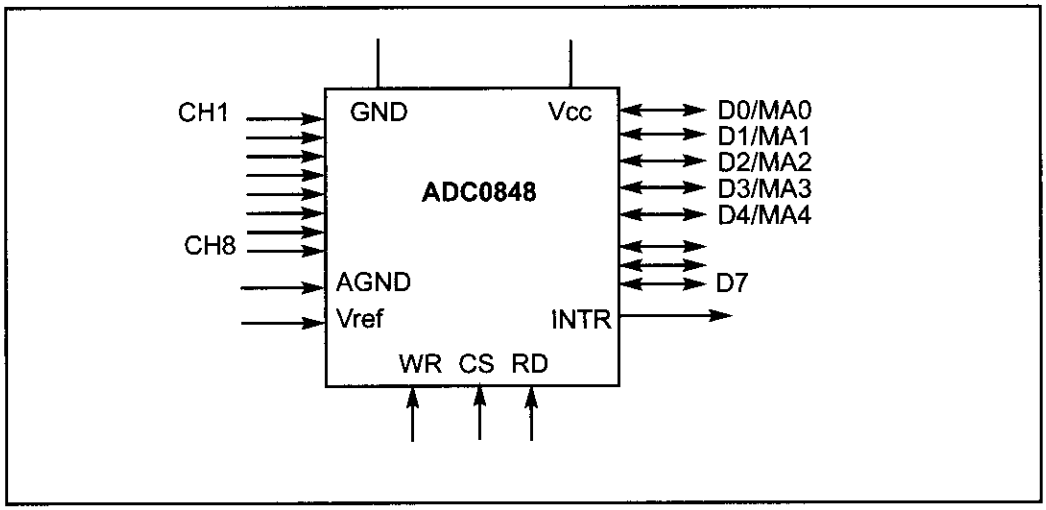


Figure 13-3. ADC0848 Parallel ADC Block Diagram

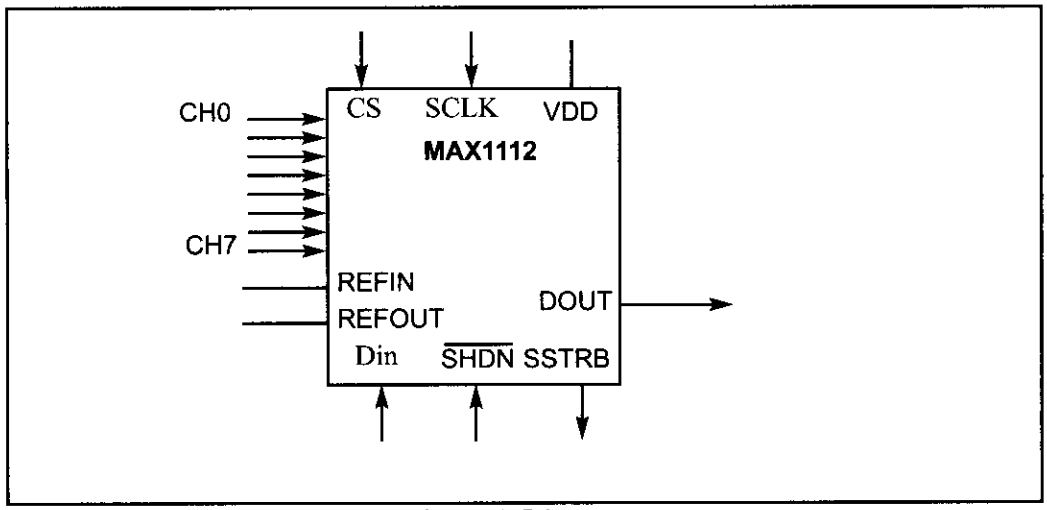


Figure 13-4. MAX1112 Serial ADC Block Diagram

lel-in-serial-out shift register responsible for sending out the binary data one bit at a time. The D0–D7 data pins of the 8-bit ADC provide an 8-bit parallel data path between the ADC chip and the CPU. In the case of the 16-bit parallel ADC chip, we need 16 pins for the data path. In order to save pins, many 12- and 16-bit ADCs use pins D0–D7 to send out the upper and lower bytes of the binary data. In recent years, for many applications where space is a critical issue, using such a large number of pins for data is not feasible. For this reason, serial devices such as the serial ADC are becoming widely used. While the serial ADCs use fewer pins and their smaller packages take much less space on the printed circuit board, more CPU time is needed to get the converted data from the ADC because the CPU must get data one bit at a time, instead of in one single read operation as with the parallel ADC. ADC848 is an example of a parallel ADC with 8 pins for the data output, while the MAX1112 is an example of a serial ADC with a single pin for Dout. Figures 13-3 and 13-4 show the block diagram for ADC848 and MAX1112, respectively.

Analog input channels

Many data acquisition applications need more than one ADC. For this reason, we see ADC chips with 2, 4, 8, or even 16 channels on a single chip. Multiplexing of analog inputs is widely used as shown in the ADC848 and MAX1112. In these chips, we have 8 channels of analog inputs, allowing us to monitor multiple quantities such as temperature, pressure, heat, and so on. PIC18 microcontroller chips come with 5 to 15 ADC channels, depending on the family member. The PIC18 ADC feature is discussed in the next section.

Start conversion and end-of-conversion signals

The fact that we have multiple analog input channels and a single digital output register makes it necessary for start conversion (SC) and end-of-conversion (EOC) signals. When SC is activated, the ADC starts converting the analog input value of Vin to an *n*-bit digital number. The amount of time it takes to convert varies depending on the conversion method as was explained earlier. When the data conversion is complete, the end-of-conversion signal notifies the CPU that the converted data is ready to be picked up.

From the discussion we conclude that the following steps must be followed for data conversion by an ADC chip:

1. Select a channel.
2. Activate the start conversion (SC) signal to start the conversion of analog input.
3. Keep monitoring the end-of-conversion (EOC) signal.
4. After the EOC has been activated, we read data out of the ADC chip.

Review Questions

1. Give two factors that affect the step size calculation.
2. The ADC0848 is a(n) _____-bit converter.
3. True or false. While the ADC0848 has 8 pins for D_{OUT}, the MAX1112 has only one D_{OUT} pin.
4. Indicate the number of analog input channels for each of the following ADC chips.
   (a) ADC0848 (b) MAX1112
5. Find the step size for an 8-bit ADC, if Vref = 1.28 V
6. For question 5, calculate the D0–D7 output if the analog input is: (a) 0.7 V, and (b) 1 V.

SECTION 13.2: ADC PROGRAMMING IN THE PIC18

Because the ADC is widely used in data acquisition, in recent years an increasing number of microcontrollers have an on-chip ADC peripheral, just like timers and USART. An on-chip ADC eliminates the need for an external ADC connection, which leaves more pins for other I/O activities. The vast majority of the PIC18 chips come with 8 channels of ADC, and some PIC18s have as many as 16 channels of ADCs. In this section we discuss the ADC feature of the PIC18452/458 and show how it is programmed in both Assembly and C.

PIC18F452/458 ADC features programming

The ADC peripheral of the PIC18 has the following characteristics:

(a) It is a 10-bit ADC.

(b) It can have 5 to 15 channels of analog input channels, depending on the family member. In PIC18452/458, pins RA0–RA7 of PORTA are used for the 8 analog channels. See Figures 13-5A and 13-5B.

(c) The converted output binary data is held by two special function registers called ADRESL (A/D Result Low) and ADRESH (A/D Result High).

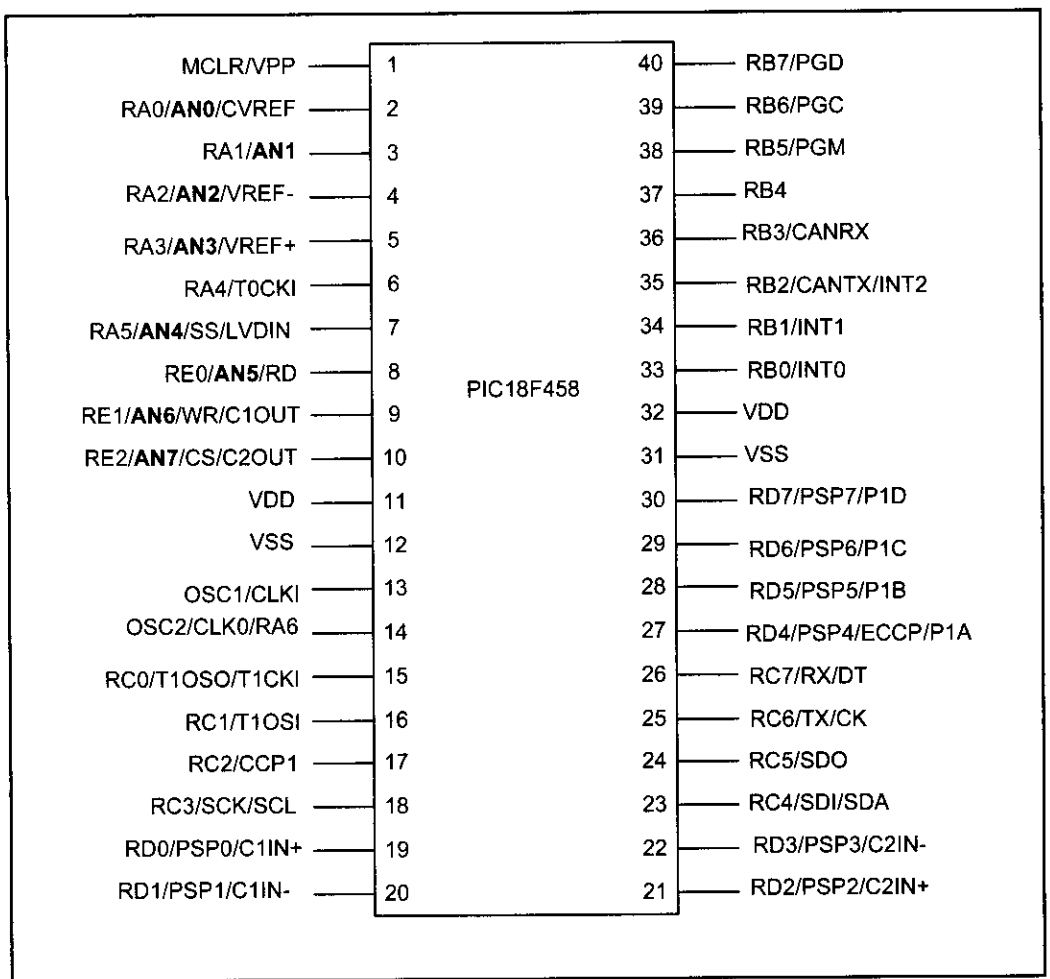


Figure 13-5A: PIC18F458 Pins with ADC Channels Shown in Bold

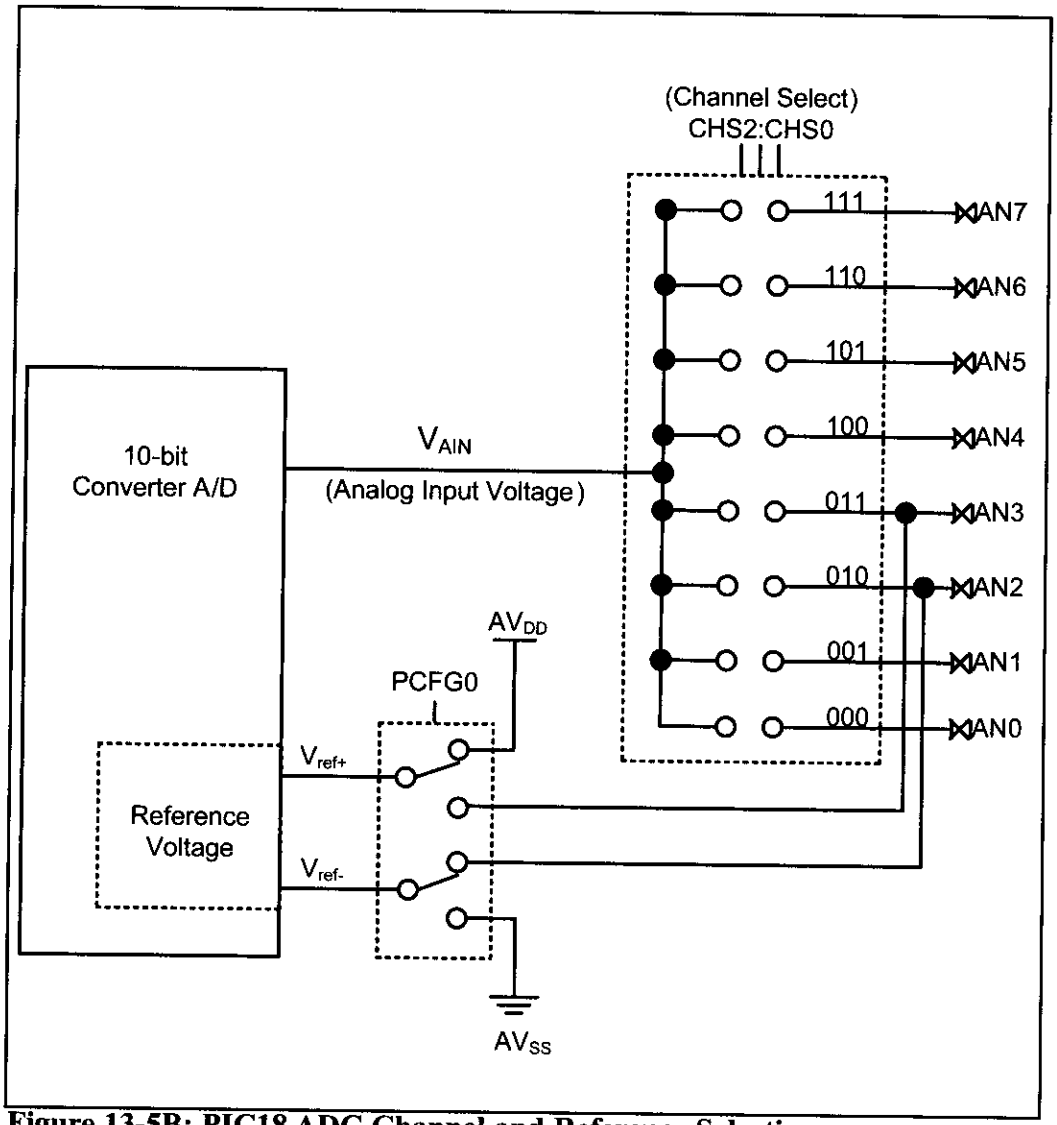


Figure 13-5B: PIC18 ADC Channel and Reference Selection

(d) Because the ADRESH:ADRESL registers give us 16 bits and the ADC data out is only 10 bits wide, 6 bits of the 16 are unused. We have the option of making either the upper 6 bits or the lower 6 bits unused.

(e) We have the option of using Vdd (Vcc), the voltage source of the PIC18 chip itself, as the Vref or connecting it to an external voltage source for the Vref.

(f) The conversion time is dictated by the Fosc of crystal frequency connected to the OSCs pins. While the Fosc for PIC18 can be as high as 40 MHz, the conversion time can not be shorter than 1.6 ms.

(g) It allows the implementation of the differential Vref voltage using the Vref(+) and Vref(–) pins, where Vref = Vref (+) – Vref (-).

Many of the above features can be programmed by way of ADCON0 (A/D control register 0) and ADCON1 (A/D control register 1), as we will see next.

ADCON0 register

The ADCON0 register is used to set the conversion time and select the analog input channel among other things. Figure 13-6 shows the ADCON0 register. In order to reduce the power consumption of the PIC18, the ADC feature is turned off when the microcontroller is powered up. We turn on the ADC with the ADON bit of the ADCON0 register, as shown in Figure 13-6. The other important bit is the GO/DONE bit. We use this bit to start conversion and monitor it to see if conversion has ended. Notice in ADCCON0 that not all family members have all the 8 analog input channels. The conversion time is set with the ADCS bits. While ADCS1 and ADCS0 are held by the ADCON0 register, ADCS2 is part of the ADCON1 register. This is discussed next.

ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	--	ADON

ADCS2 (from ADCON1)	ADCS1	ADCS0	Conversion Clock Source
0	0	0	Fosc/2
0	0	1	Fosc/8
0	1	0	Fosc/32
0	1	1	Internal RC used for clock source
1	0	0	Fosc/4
1	0	1	Fosc/16
1	1	0	Fosc/64
1	1	1	Internal RC used for clock source

CHS2	CHS1	CHS0	CHANNEL SELECTION	
0	0	0	CHAN0 (AN0)	
0	0	1	CHAN1 (AN1)	
0	1	0	CHAN2 (AN2)	
0	1	1	CHAN3 (AN3)	
1	0	0	CHAN4 (AN4)	
1	0	1	CHAN5 (AN5)	not implemented on 28-pin PIC18
1	1	0	CHAN6 (AN6)	not implemented on 28-pin PIC18
1	1	1	CHAN7 (AN7)	not implemented on 28-pin PIC18

GO/DONE A/D conversion status bit.
1 = A/D conversion is in progress. This is used as start conversion, which means after the conversion is complete, it will go LOW to indicate the end-of-conversion.
0 = A/D conversion is complete and digital data is available in registers ADRESH and ADRESL.

ADON A/D on bit
0 = A/D part of the PIC18 is off and consumes no power. This is the default and we should leave it off for applications in which ADC is not used.
1 = A/D feature is powered up.

Figure 13-6. ADCON0 (A/D Control Register 0)

ADCON1 register

Another major register of the PIC18's ADC feature is ADCON1. The ADCON1 register is used to select the Vref voltage among other things. It is shown in Figure 13-7. After the A/D conversion is complete, the result sits in registers ADRESL (A/D Result Low Byte) and ADRESH (A/D Result High Byte). The ADFM bit of the ADCON1 is used for making it right-justified or left-justified because we need only 10 bits of the 16. See Figure 13-8.

ADFM	ADCS2	--	--	PCFG3	PCFG2	PCFG1	PCFG0

ADFM A/D Result format select bit

1 = Right justified: The 10-bit result is in the ADRESL register and the lower 2 bits of ADRESH. That means the 6 most significant bits of the ADRESH register are all 0s.

0 = Left justified: The 10-bit result is in the ADRESL register and the upper 2 bits of ADRESL. That means the 6 least significant bits of the ADRESL register are all 0s.

ADCS2 A/D Clock Select bit 2. This bit along with the ADCS1 and ADCS0 bits of the ADCON0 register decide the conversion clock for the ADC. The default value for ADCS2 is 0, which means setting the ADCS0 and ADCS1 values of ADCON0 can give us clock conversion of Fosc/2, Fosc/8, and Fosc/32. See the ADCON0 register.

PCFGs: A/D Port Configuration Control bits:

PCFGs	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	Vref+	Vref-	C/R
0 0 0 0	A	A	A	A	A	A	A	A	Vdd	Vss	8/0
0 0 0 1	A	A	A	A	Vref+	A	A	A	AN3	Vss	7/1
0 0 1 0	D	D	D	A	A	A	A	A	Vdd	Vss	5/0
0 0 1 1	D	D	D	A	Vref+	A	A	A	AN3	Vss	4/1
0 1 0 0	D	D	D	D	A	D	A	A	Vdd	Vss	3/0
0 1 0 1	D	D	D	D	Vref+	D	A	A	AN3	Vss	2/1
01 1 x	D	D	D	D	D	D	D	D	-	-	0/0
1 0 0 0	A	A	A	A	Vref+	Vref–	A	A	AN3	AN2	6/2
1 0 0 1	D	D	A	A	A	A	A	A	Vdd	Vss	6/0
1 0 1 0	D	D	A	A	Vref+	A	A	A	AN3	Vss	5/1
1 0 1 1	D	D	A	A	Vref+	Vref–	A	A	AN3	AN2	4/2
1 1 0 0	D	D	D	A	Vref+	Vref–	A	A	AN3	AN2	3/2
1 1 0 1	D	D	D	D	Vref+	Vref–	A	A	AN3	AN2	2/2
1 1 1 0	D	D	D	D	D	D	D	A	Vdd	Vss	1/0
1 1 1 1	D	D	D	D	Vref+	Vref–	D	A	AN3	AN2	1 /2

A = Analog input, D = Digital I/O

C/R = # of analog input channels / # of pins used for A/D voltage reference

The default is option 0000, which gives us 8 channels of analog input and uses the Vdd of PIC18 as Vref.

Figure 13-7. ADCON1 (A/D Control Register 1)

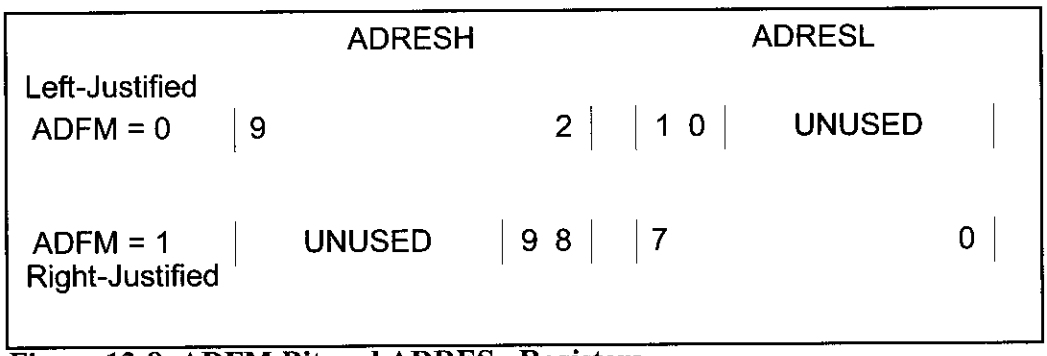


Figure 13-8. ADFM Bit and ADRESx Registers

The port configuration for A/D channels is handled by the PCFG (A/D port configuration) bits. While in chips such as the PIC18452/458, we can have up to 8 channels of analog input, not all applications need that many ADC inputs. The PORTA pins of RA0–RA3 and RA5 and RE0–RE2 of PORTE are used for the analog input channels. With PCFG = 0110, we can use all the pins of the PORTA as the digital I/O. The default is PCFG = 0000, which allows us to use all 8 pins for analog inputs. In that case Vref = Vdd, the same voltage source used by the PIC18 chip itself. In many applications we need Vref other than Vdd. The AN3 pin can be used as an external source of voltage for Vref. For example, option PCFG = 0101 allows us to use two channels for analog inputs, AN3 = Vref, and the other 5 pins for the digital I/O. In this case the Vss (Gnd) pin of the PIC18 is used for the Vref (–). See Examples 13-2 and 13-3.

Example 13-2

For a PIC18-based system, we have V_{ref} = Vdd = 5 V. Find (a) the step size, and (b) the ADCON1 value if we need 3 channels. Assume that the ADRESH:ADRESL registers are right justified.

Solution:

(a) The step size is 5/1,024 = 4.8 mV.
(b) ADCON1 = 1x000100 because option 100 gives us 3 analog input channels. The x = ADCS2 is decided by the conversion speed.

Example 13-3

For a PIC18-based system, we have V_{ref} = 2.56 V. Find (a) the step size, and (b) the ADCON1 value if we need 3 channels. Assume that the ADRESH:ADRESL registers are right justified.

Solution:

(a) The step size is 2.56/1,024 = 2.5 mV.
(b) ADCON1 = 1x000011 because option 0011 gives us 3 analog input channels where x = ADCS2 is decided by the conversion speed.

Calculating A/D conversion time

By using the ADCS (A/D clock source) bits of both the ADCON0 and ADCON1 registers we can set the A/D conversion time. The conversion time is defined in terms of Tad, where Tad is the conversion time per bit. To calculate the Tad, we can select a conversion clock source of Fosc/2, Fosc/4, Fosc/8, Fosc/16, Fosc/32, or Fosc/64, where Fosc is the speed of the crystal frequency connected to the PIC18 chip. For the PIC18, the conversion time is 12 times the Tad. Notice that the Tad cannot be faster than 1.6 ms. Look at Examples 13-4 and 13-5 for clarification.

We can also use the the internal RC oscillator for the conversion clock source, instead of the Fosc of the external crystal oscillator. In that case the Tad is typically 4–6 μs and conversion time is 12 × 6 μs = 72 μs.

Another timing factor that we must pay attention to is the acquisition time (Tacq). After an A/D channel is selected, we must allow some time for the sample -and-hold capacitor (C hold) to charge fully to the input voltage level present at the channel. It is only after the elapsing of this acquisition time that the A/D conversion can be started. Although many factors (e.g., Vdd and temperature) affect the duration of Tacq, we can use a typical value of 15 μs. In some newer generations of the PIC18, we have the option of controlling the exact time of Tacq by programming the internal register ADCON2. In the PIC18F452/458, we have only the ADCON0 and ADCON1 registers. See Example 13-6.

Example 13-4

A PIC18 is connected to the 10 MHz crystal oscillator. Calculate the conversion time for all options of ADCS bits in both the ADCON0 and ADCON1 registers.

Solution:

The options for the conversion clock source for both ADCON0 and ADCON1 are as follows:
(a) For Fosc/2, we have 10 MHz / 2 = 5 MHz.
Tad = 1 / 5 MHz = 200 ns. Invalid because it is faster than 1.6 μs.
(b) For Fosc/4, we have 10 MHz / 4 = 2.5 MHz.
Tad = 1 / 2.5 MHz = 400 ns. Invalid because it is faster than 1.6 μs.
(c) For Fosc/8, we have 10 MHz / 8 = 1.25 MHz.
Tad = 1 / 2.5 MHz = 800 ns. Invalid because it is faster than 1.6 μs.
(d) For Fosc/16, we have 10 MHz / 16 = 625 kHz.
Tad =1 / 625 kHz = 1.6 μs. The conversion time = 12 × 1.6 μs = 19.2 μs
(e) For Fosc/32, we have 10 MHz / 32 = 312.5 kHz.
Tad = 1 / 312.5 kHz = 3.2 μs. The conversion time = 12 × 3.2 μs = 38.4 μs
(f) For Fosc/64, we have 10 MHz / 64 = 156.25 kHz.
Tad = 1 / 156.25 kHz = 6.4 μs. The conversion time = 12 × 6.4 μs = 76.8 μs
Notice that for the Fosc/4, Fosc/16, and Fosc/64 selections, we must use the ADSC2 bit in the ADCON1 register, in addition to the ADCS bits in the ADCON0 register.

Example 13-5

A PIC18 is connected to the 4 MHz crystal oscillator. Calculate the conversion time if we want to use only the ADCS bits of the ADCON0 register.

Solution:

The options for the conversion clock source available in the ADCON0 register are as follows:
(a) For Fosc/2, we have 4 MHz / 2 = 2 MHz.
Tad = 1 / 2 MHz = 400 ns. Invalid because it is faster than 1.6 μs.
(b) For Fosc/8, we have 4 MHz / 8 = 500 kHz.
Tad = 1 / 500 kHz = 2 μs. The conversion time = 12 × 2 μs = 24 μs
(c) For Fosc/32, we have 4 MHz / 32 = 125 kHz.
Tad = 1 / 125 kHz = 8 μs. The conversion time = 12 × 8 μs = 96 μs

Example 13-6

Find the values for the ADCON0 and ADCON1 registers for the following options: (a) channel AN0 as analog input, (b) Vref+ = Vdd, Vref– = Vss, (c) Fosc/64, (d) A/D result is right justified, and (e) A/D module is on.

Solution:

From Figure 13-6, we have ADCON0 = 10000x1. With x = 0 we have 10000001.
From Figure 13-7, we have ADCON1 = 11xx1110. With x = 0 we have 11001110.

Steps in programming the A/D converter using polling

To program the A/D converter of the PIC18, the following steps must be taken:

1. Turn on the ADC module of the PIC18 because it is disabled upon power-on reset to save power. We can use the "`BSF ADCON0,ADON`" instruction.
2. Make the pin for the selected ADC channel an input pin. We use "`BSF TRISA,x`." or "`BSF TRISE,x`" where x is the channel number.
3. Select voltage reference and A/C input channels. We use registers ADCON0 and ADCON1.
4. Select the conversion speed. We use registers ADCON0 and ADCON1.
5. Wait for the required acquisition time.
6. Activate the start conversion bit of GO/DONE.
7. Wait for the conversion to be completed by polling the end-of-conversion (GO/DONE) bit.
8. After the GO/DONE bit has gone LOW, read the ADRESL and ADRESH registers to get the digital data output.
9. Go back to step 5.

Programming PIC18F458 ADC in Assembly

The Assembly language Program 13-1 illustrates the steps for ADC conversion shown above. The C version of the program is shown in Program 13-1C.

```
Program 13-1: This program gets data from channel 0 (RA0)
of ADC and displays the result on PORTC and PORTD. This is
done every quarter of second.
;Program 13-1
      ORG   0000H
      CLRF   TRISC           ;make PORTC an output
      CLRF   TRISD           ;make PORTD an output
      BSF    TRISA,0 ;make RA0 an input for analog input
      MOVLW  0x81            ;Fosc/64, channel 0, A/D is on
      MOVWF  ADCON0
      MOVLW  0xCE  ;right justified, Fosc/64, AN0 = analog
      MOVWF  ADCON1
OVER  CALL   DELAY ;wait for Tacq (sample and hold time)
      BSF   ADCON0,GO          ;start conversion
BACK  BTFSC ADCON0,DONE        ;keep polling end-of-conversion
      BRA   BACK               ;wait for end of conversion
      MOVFF  ADRESL,PORTC      ;give the low byte to PORTC
      MOVFF  ADRESH,PORTD      ;give the high byte to PORTD
      CALL   QSEC_DELAY
      BRA   OVER               ;keep repeating it
      END
```

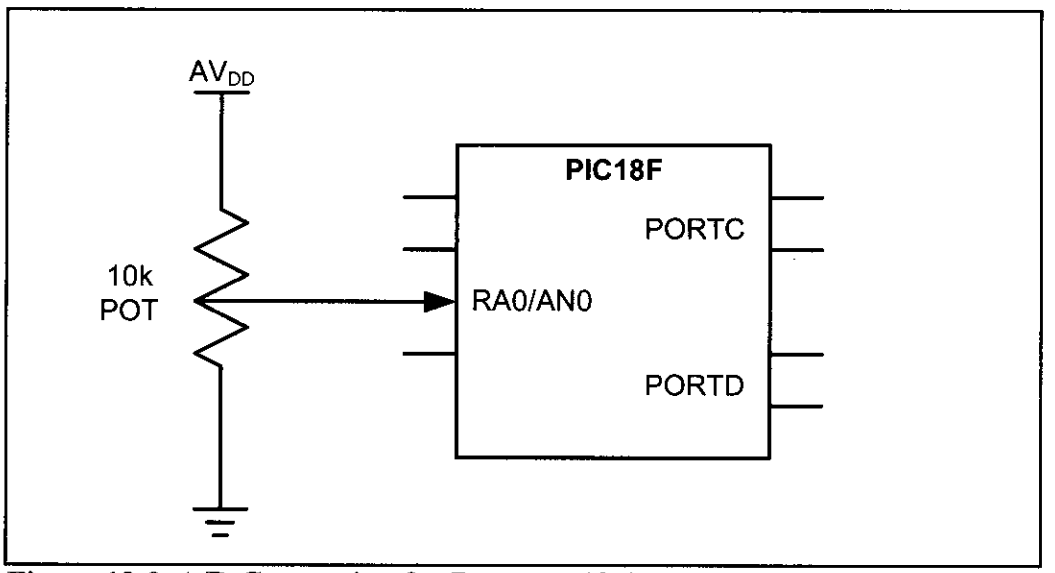


Figure 13-9. A/D Connection for Program 13-1

Programming PIC18F458 A/D in C

Program 13-1C is the C version of the ADC conversion for Program 13-1.

```
Program 13-1C: This program gets data from channel 0 (RA0)
of ADC and displays the result on PORTC and PORTD. This is
done every quarter of second. This is the C version of
Program 13-1.
//Program 13-1C

void main(void)
  {
     TRISC=0;                 //make PORTC output port
     TRISD=0;                 //make PORTD output port
     TRISAbits.TRISA0=0;      //RA0 = INPUT for analog input
     ADCON0 = 0x81;   //Fosc/64, channel 0, A/D is on
     ADCON1 = 0xCE;           //right justified, Fosc/64,
                              //AN0 = analog
     while(1)
     {
      DELAY(1); //give A/D channel time to sample
      ADCON0bits.GO = 1;      //start converting
      while(ADCON0bits.DONE == 1);
      PORTC=ADRESL;           //display low byte on PORTC
      PORTD=ADRESH;           //display high byte on PORTD
      DELAY(250);             //wait for one quarter of a
                              //second before trying again
     }
  }
```

Programming A/D converter using interrupts

In Chapter 11, we showed how to use interrupts instead of polling to avoid tying down the microcontroller. To program the A/D using the interrupt method, we need to set HIGH the ADIE (A/D interrupt enable) flag. If ADIE = 1, then upon the completion of the conversion, the ADIF (A/D interrupt flag) becomes HIGH, which will force the CPU to jump to read binary outputs. Table 13-4 shows to which registers these two flags belong.

Table 13-4: A/D Converter Interrupt Flag Bits and their Registers

Interrupt	Flag bit	Register	Enable bit	Register
ADIF (ADC)	ADIF	PIR1	ADIE	PIE1

Note: Upon power-on reset, the A/D is assigned to high-priority interrupt (vector address of 0008). We can use the ADIP bit of the IPR1 register to assign low priority to it, which will land at vector address 00018H. See Chapter 11.

```
;Program 13-2
      ORG   0000H
      GOTO  MAIN        ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
      ORG   0008H       ;interrupt vector table
      BTFSS PIR1,ADIF   ;Did we get here due to A/D int?
      RETFIE            ;No. Then return to main
      GOTO  AD_ISR      ;Yes. Then go INT0 ISR
;--the main program for initialization
      ORG   00100H
MAIN  CLRF  TRISC       ;make PORTC an output
      CLRF   TRISD      ;make PORTD an output
      BSF   TRISA,0 ;make RA0 an input pin for analog input
      MOVLW 0x81        ;Fosc/64, channel 0, A/D is on
      MOVWF ADCON0
      MOVLW 0xCE  ;right justified, Fosc/64, AN0 = analog
      MOVWF ADCON1
      BCF PIR1,ADIF     ;clear ADIF for the first round
      BSF PIE1,ADIE     ;enable A/D interrupt
      BSF INTCON,PEIE ;enable peripheral interrupts
      BSF INTCON,GIE    ;enable interrupts globally
OVER  CALL DELAY        ;wait for Tacq (sample and hold time)
      BSF   ADCON0,GO   ;start conversion
      BRA OVER          ;stay in this loop forever
;-----A/D Converter ISR
AD_ISR
      ORG 200H
      MOVFF ADRESL,PORTC    ;give the low byte to PORTC
      MOVFF ADRESH,PORTD    ;give the high byte to PORTD
      CALL QSEC_DELAY
      BCF PIR1,ADIF     ;clear ADIF interrupt flag bit
      RETFIE
      END

//Program 13-2C (This is the C version of Program 13-2)

#include <PIC18F458.h>

#pragma code My_HiPrio_Int=0x0008 //high-priority interrupt
void My_HiPrio_Int (void)
{
  chk_isr();
}
#pragma code                //end high-priority interrupt
#pragma interrupt chk_isr  //Which interrupt?
void chk_isr (void)
{
  if (PIR1bits.ADIF==1)     //A/D caused interrupt?
  AD_ISR( );                //Yes. Execute INT0 program
}
```

```
void main(void)
{
  TRISC=0;                    //make PORTC output port
  TRISD=0;                    //make PORTD output port
  TRISAbits.TRISA0=0;         //RA0 = INPUT for analog input
  ADCON0 = 0x81;              //Fosc/64, channel 0, A/D is on
  ADCON1 = 0xCE; //right justified, Fosc/64, AN0 = analog
  PIR1bits.ADIF=0;            //clear A/D interrupt flag
  PIE1bits.ADIE=1;            //enable A/D interrupt
  INTCONbits.PEIE=1;          //enable peripheral interrupts
  INTCONbits.GIE=1;           //enable all interrupts globally
  while(1)              //keep looping until interrupt comes
  {
    DELAY(1);
    ADCON0bits.GO = 1;        //start converting
  }
}
//----------A/D ISR
void AD_ISR(void)
  {
    PORTC=ADRESL;             //display low byte on PORTC
    PORTD=ADRESH;             //display high byte on
    DELAY(250);               //wait for one quarter of a
                              //second before trying again
    PIR1bits.ADIF=0;          //clear A/D interrupt flag
  }
```

Review Questions

1. Give the main factor affecting the step size of A/D in PIC18.
2. The A/D of PIC18 is a(n) _____-bit converter.
3. True or false. The A/D of PIC18 has pins for D_{OUT}.
4. True or false. A/D in the PIC18 is an off-chip module.
5. Find the step size for an PIC18 ADC, if Vref = 1.024 V.
6. For problem 5, calculate the D0–D9 output if the analog input is: (a) 0.7 V, and (b) 1 V.
7. Indicate the number of available analog input channels for each of the following options in the ADCON0 register:
   (a) PCFG = 0100 (b) PCFG = 1001
8. True or false. The conversion time is equal to 12 × Tad.
9. The minimum Tad allowed is _____ μs.
10. Which bit is used to poll for the end of conversion?

SECTION 13.3: DAC INTERFACING

This section will show how to interface a DAC (digital-to-analog converter) to the PIC18. Then we demonstrate how to generate a sine wave on the scope using the DAC.

Digital-to-analog converter (DAC)

The digital-to-analog converter (DAC) is a device widely used to convert digital pulses to analog signals. In this section we discuss the basics of interfacing a DAC to the PIC18.

Recall from your digital electronics course the two methods of creating a DAC: binary weighted and R/2R ladder. The vast majority of integrated circuit DACs, including the MC1408 (DAC0808) used in this section, use the R/2R method because it can achieve a much higher degree of precision. The first criterion for judging a DAC is its resolution, which is a function of the number of binary inputs. The common ones are 8, 10, and 12 bits. The number of data bit inputs decides the resolution of the DAC because the number of analog output levels is equal to 2^n, where n is the number of data bit inputs. Therefore, an 8-input DAC such as the DAC0808 provides 256 discrete voltage (or current) levels of output. See Figure 13-10. Similarly, the 12-bit DAC provides 4,096 discrete voltage levels. There are also 16-bit DACs, but they are more expensive.

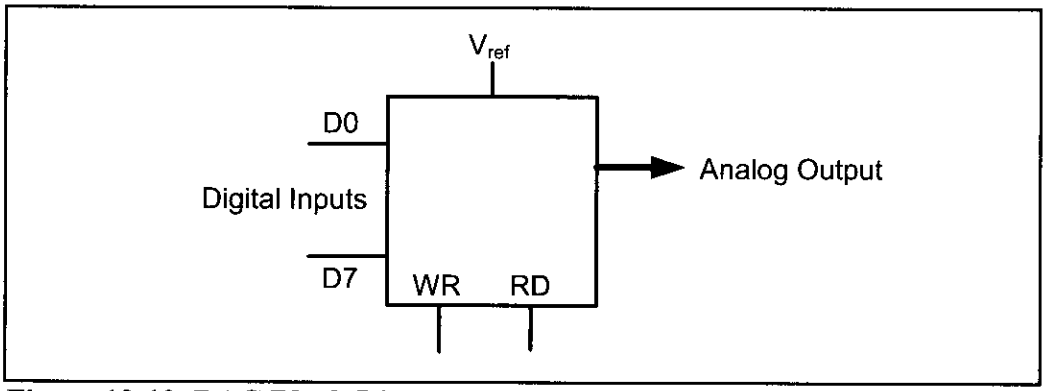


Figure 13-10. DAC Block Diagram

MC1408 DAC (or DAC0808)

In the MC1408 (DAC0808), the digital inputs are converted to current (I_{out}), and by connecting a resistor to the I_{out} pin, we convert the result to voltage. The total current provided by the I_{out} pin is a function of the binary numbers at the D0–D7 inputs of the DAC0808 and the reference current (I_{ref}), and is as follows:

$$I_{out} = I_{ref} \left(\frac{D7}{2} + \frac{D6}{4} + \frac{D5}{8} + \frac{D4}{16} + \frac{D3}{32} + \frac{D2}{64} + \frac{D1}{128} + \frac{D0}{256} \right)$$

where D0 is the LSB, D7 is the MSB for the inputs, and I_{ref} is the input current that must be applied to pin 14. The I_{ref} current is generally set to 2.0 mA. Figure 13-11 shows the generation of current reference (setting $I_{ref} = 2$ mA) by using the

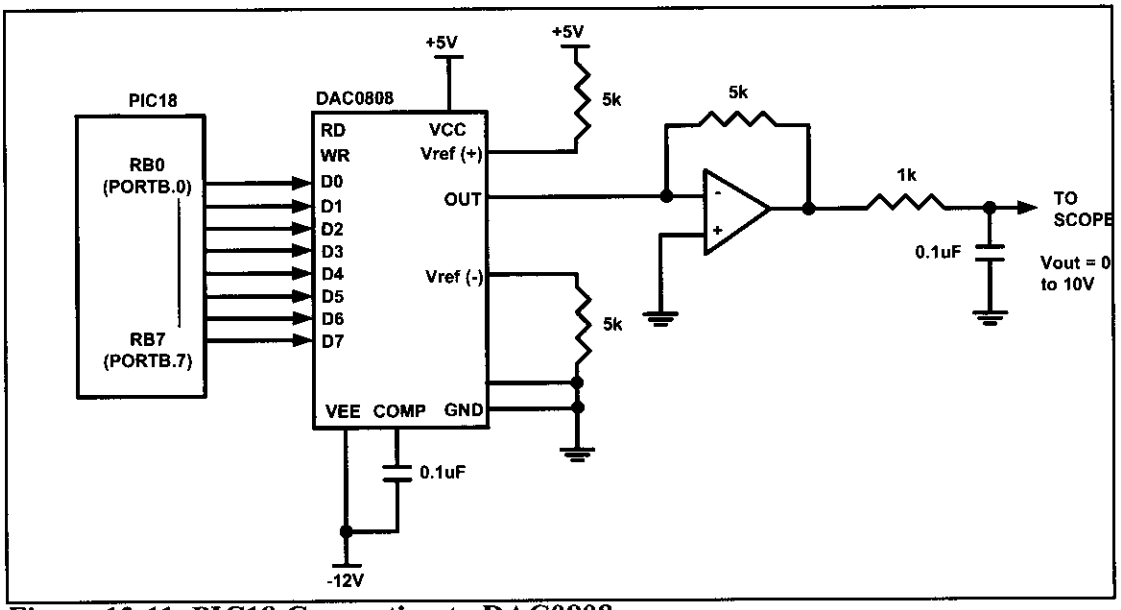


Figure 13-11. PIC18 Connection to DAC0808

standard 5 V power supply. Now assuming that I_{ref} = 2 mA, if all the inputs to the DAC are high, the maximum output current is 1.99 mA (verify this for yourself).

Converting I_{out} to voltage in DAC0808

Ideally we connect the output pin I_{out} to a resistor, convert this current to voltage, and monitor the output on the scope. In real life, however, this can cause inaccuracy because the input resistance of the load where it is connected will also affect the output voltage. For this reason, the I_{ref} current output is isolated by connecting it to an op-amp such as the 741 with R_f = 5 kOhms for the feedback resistor. Assuming that R = 5 kOhms, by changing the binary input, the output voltage changes as shown in Example 13-7.

Example 13-7

Assuming that R = 5 kOhms and I_{ref} = 2 mA, calculate V_{out} for the following binary inputs:
(a) 10011001 binary (99H) (b) 11001000 (C8H)

Solution:

(a) I_{out} = 2 mA (153/256) = 1.195 mA and V_{out} = 1.195 mA × 5K = 5.975 V
(b) I_{out} = 2 mA (200/256) = 1.562 mA and V_{out} = 1.562 mA × 5K = 7.8125 V

Generating a sine wave

Example 13-8 shows how to generate a stair-step ramp. To generate a sine wave, we first need a table whose values represent the magnitude of the sine of angles between 0 and 360 degrees. The values for the sine function vary from -1.0 to +1.0 for 0- to 360-degree angles. Therefore, the table values are integer num-

Example 13-8

In order to generate a stair-step ramp, set up the circuit in Figure 13-11 and connect the output to an oscilloscope. Then write a program to send data to the DAC to generate a stair-step ramp.

Solution:

```
            CLRF  TRISB       ;PORTB as output
            CLRF  PORTB       ;clear PORTB
AGAIN:      INCF  PORTB,F     ;count from 0 to FFH send it to DAC
            RCALL DELAY       ;let DAC recover
            BRA   AGAIN
```

bers representing the voltage magnitude for the sine of theta. This method ensures that only integer numbers are output to the DAC by the PIC18 microcontroller. Table 13-5 shows the angles, the sine values, the voltage magnitudes, and the integer values representing the voltage magnitude for each angle (with 30-degree increments). To generate Table 13-5, we assumed a full-scale voltage of 10 V for DAC output (as designed in Figure 13-11). Full-scale output of the DAC is achieved when all the data inputs of the DAC are HIGH. Therefore, to achieve the full-scale 10 V output, we use the following equation.

$$V_{out} = 5\ V + (5 \times \sin \theta)$$

V_{out} of DAC for various angles is calculated and shown in Table 13-5. See Example 13-9 for verification of the calculations.

Table 13-5: Angle versus Voltage Magnitude for Sine Wave

Angle θ (degrees)	Sin θ	V_{out} (Voltage Magnitude) 5 V + (5 V × sin θ)	Values Sent to DAC (decimal) (Voltage Mag. × 25.6)
0	0	5	128
30	0.5	7.5	192
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192
180	0	5	128
210	–0.5	2.5	64
240	–0.866	0.669	17
270	–1.0	0	0
300	–0.866	0.669	17
330	–0.5	2.5	64
360	0	5	128

Example 13-9

Verify the values given for the following angles: (a) 30° (b) 60°.

Solution:

(a) $V_{out} = 5\ V + (5\ V \times \sin \theta) = 5\ V + 5 \times \sin 30° = 5\ V + 5 \times 0.5 = 7.5\ V$
DAC input value = 7.5 V × 25.6 = 192 (decimal)

(b) $V_{out} = 5\ V + (5\ V \times \sin \theta) = 5\ V + 5 \times \sin 60° = 5\ V + 5 \times 0.866 = 9.33\ V$
DAC input value = 9.33 V × 25.6 = 238 (decimal)

To find the value sent to the DAC for various angles, we simply multiply the V_{out} voltage by 25.60 because there are 256 steps and full-scale V_{out} is 10 volts. Therefore, 256 steps / 10 V = 25.6 steps per volt. To further clarify this, look at the following code. This program sends the values to the DAC continuously (in an infinite loop) to produce a crude sine wave. See Figure 13-12.

```
;Program 13-3
OVER  MOVLW upper(TABLE)
      MOVWF TBLPTRU
      MOVLW high(TABLE)
      MOVWF TBLPTRH
      MOVLW low(TABLE)
      MOVWF TBLPTRL
      CLRF  TRISB
AGAIN TBLRD*
      MOVF  TABLAT,W
      XORLW 0x0
      BZ    OVER
      MOVWF PORTB
      INCF  TBLPTRL,F
      BRA   AGAIN
      ORG   0x250
TABLE:      DB D'128',D'192',D'238',D'255',D'238',D'192'
      DB D'128',D'64',D'17',D'1',D'17',D'64',D'0'
      END

      ;to get a better looking sine wave, regenerate
      ;Table 13-5 for 2-degree angles
```

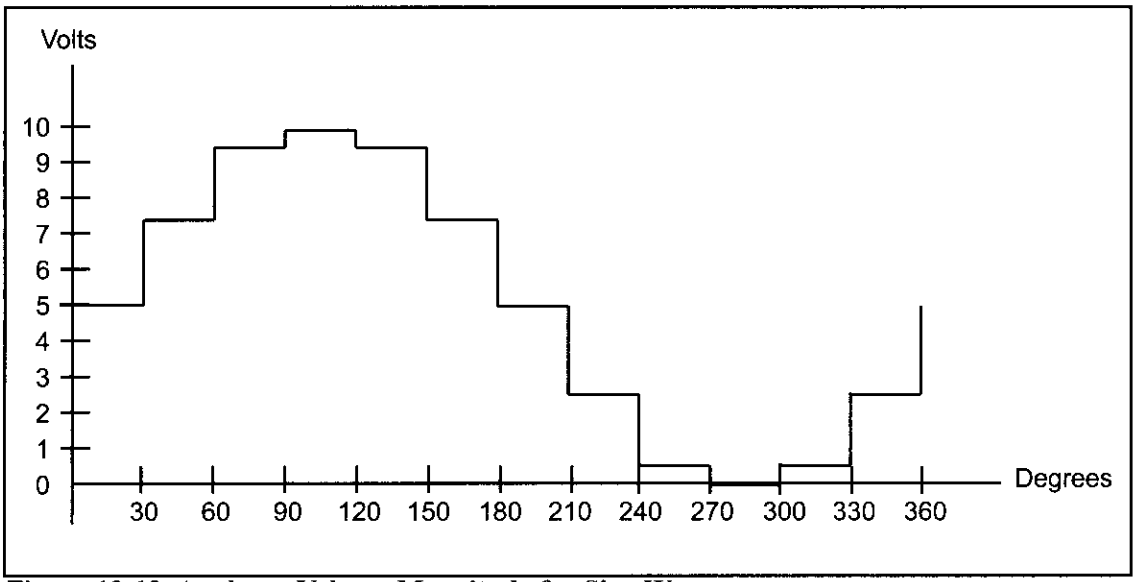


Figure 13-12. Angle vs. Voltage Magnitude for Sine Wave

Programming DAC in C

```
//Program 13-3C. This is the C version of Program 13-3.
#include <p18F458.h>
rom const unsigned char WAVEVALUE[12] ={128,192,238,255,
                  238,192,128,64,
                  17,0,17,64};
void main()
  {
    unsigned char x;
    TRISB=0;
    while(1)
      {
      for(x=0;x<12;x++)
       PORTB = WAVEVALUE[x];
      }
  }
```

Review Questions

1. In a DAC, input is _____ (digital, analog) and output is _____ (digital, analog).
2. In an ADC, input is _____ (digital, analog) and output is _____ (digital, analog).
3. DAC0808 is a(n) ____-bit D-to-A converter.
4. (a) The output of DAC0808 is in _____ (current, voltage).
   (b) True or false. The output of DAC0808 is ideal to drive a motor.

SECTION 13.4: SENSOR INTERFACING AND SIGNAL CONDITIONING

This section will show how to interface sensors to the microcontroller. We examine some popular temperature sensors and then discuss the issue of signal conditioning. Although we concentrate on temperature sensors, the principles discussed in this section are the same for other types of sensors such as light and pressure sensors.

Temperature sensors

Transducers convert physical data such as temperature, light intensity, flow, and speed to electrical signals. Depending on the transducer, the output produced is in the form of voltage, current, resistance, or capacitance. For example, temperature is converted to electrical signals using a transducer called a *thermistor*. A thermistor responds to temperature change by changing resistance, but its response is not linear, as seen in Table 13-6.

Table 13-6: Thermistor Resistance vs. Temperature

Temperature (C)	Tf (K ohms)
0	29.490
25	10.000
50	3.893
75	1.700
100	0.817

From William Kleitz, *Digital Electronics*

The complexity associated with writing software for such nonlinear devices has led many manufacturers to market a linear temperature sensor. Simple and widely used linear temperature sensors include the LM34 and LM35 series from National Semiconductor Corp. They are discussed next.

LM34 and LM35 temperature sensors

The sensors of the LM34 series are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Fahrenheit temperature. See Table 13-7. The LM34 requires no external calibration because it is internally calibrated. It outputs 10 mV for each degree of Fahrenheit temperature. Table 13-7 is a selection guide for the LM34.

The LM35 series sensors are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Celsius (centigrade) tem-

Table 13-7: LM34 Temperature Sensor Series Selection Guide

Part	Temperature Range	Accuracy	Output Scale
LM34A	–50 F to +300 F	+2.0 F	10 mV/F
LM34	–50 F to +300 F	+3.0 F	10 mV/F
LM34CA	–40 F to +230 F	+2.0 F	10 mV/F
LM34C	–40 F to +230 F	+3.0 F	10 mV/F
LM34D	–32 F to +212 F	+4.0 F	10 mV/F

Note: Temperature range is in degrees Fahrenheit.

Table 13-8: LM35 Temperature Sensor Series Selection Guide

Part	Temperature Range	Accuracy	Output Scale
LM35A	–55 C to +150 C	+1.0 C	10 mV/C
LM35	–55 C to +150 C	+1.5 C	10 mV/C
LM35CA	–40 C to +110 C	+1.0 C	10 mV/C
LM35C	–40 C to +110 C	+1.5 C	10 mV/C
LM35D	0 C to +100 C	+2.0 C	10 mV/C

Note: Temperature range is in degrees Celsius.

perature. The LM35 requires no external calibration because it is internally calibrated. It outputs 10 mV for each degree of centigrade temperature. Table 13-8 is the selection guide for the LM35. (For further information see http://www.national.com.)

Signal conditioning and interfacing the LM35 to the PIC18

Signal conditioning is widely used in the world of data acquisition. The most common transducers produce an output in the form of voltage, current, charge, capacitance, and resistance. We need to convert these signals to voltage, however, in order to send input to an A-to-D converter. This conversion (modification) is commonly called *signal conditioning*. See Figure 13-13. Signal conditioning can be a current-to-voltage conversion or a signal amplification. For example, the thermistor changes resistance with temperature. The change of resistance must be translated into voltages in order to be of any use to an ADC. Look at the case of connecting an LM34 to an ADC of the PIC18F458. The A/D has 10-bit resolution with a maximum of 1,024 steps and the LM34 (or LM35) produces 10 mV for every degree of temperature change. Now, if we use the step size of 10 mV, the V_{out} will be 10,240 mV (10.24 V) for full-scale output. This is not acceptable even though the maximum temperature sensed by the LM34 is 300 degrees F, and the highest output for the A/D we will get is 3,000 mV (3.00 V). Now, if we use the step size of 2.5 mV, the V_{out} will be 1,024 × 2.5 mV = 2,560 mV (2.56 V) for full-scale output. That means we must set V_{ref} = 2.56 V. This makes the binary output number for the A/D 4 times the real temperature (10 mV/2.5 mV = 4). We can scale it by dividing it by 4 to get the real number for temperature. See Table 13-9.

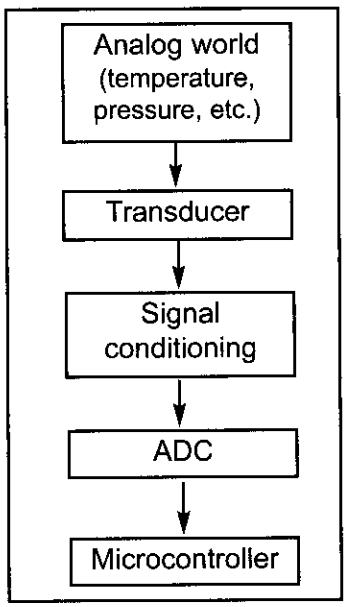


Figure 13-13. Getting Data From the Analog World

Figure 13-14 shows the connection of a temperature sensor to the PIC18F458. Notice that we use the LM336-2.5 zener diode to fix the voltage across the 10K pot at 2.5 volts. The use of the LM336-2.5 should overcome any fluctuations in the power supply.

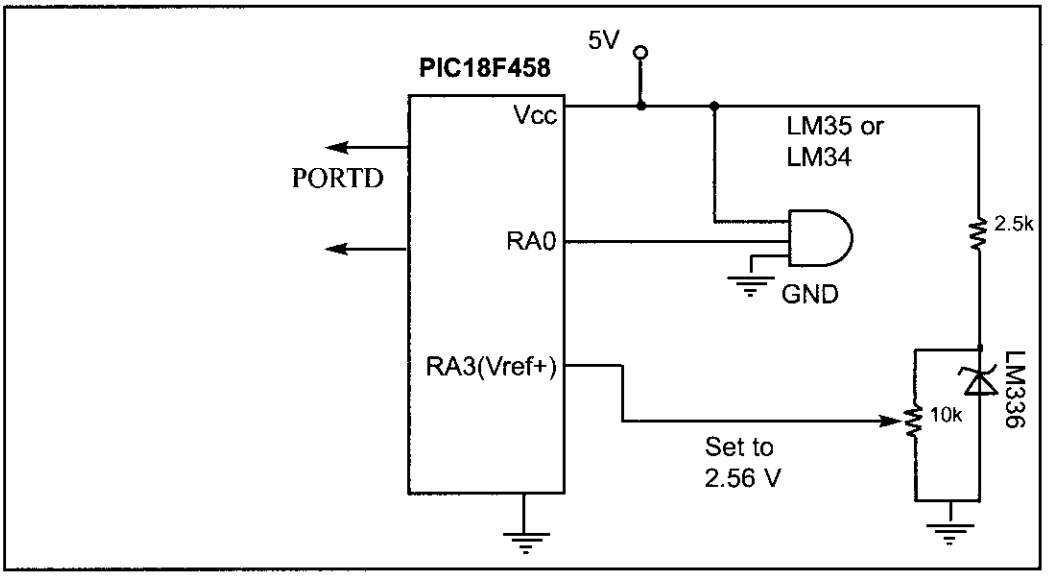


Figure 13-14. PIC18F458 Connection to Temperature Sensor

Table 13-9: Temperature vs. V_{out} for PIC18 with Vref = 2.56 V (SS = 2.5 mV)

Temp. (F)	V_{in} (mV)	#of steps	Binary V_{out} (b9–b0)	Temp. in Binary
0	0	0	00 00000000	00000000
1	10	4	00 00000100	00000100
2	20	8	00 00001000	00000010
3	30	12	00 00001100	00000011
10	100	20	00 00101000	00001010
20	200	80	00 01010000	00010100
30	300	120	00 01111000	00011110
40	400	160	00 10100000	00101000
50	500	200	00 11001000	00110010
60	600	240	00 11110000	00111100
70	700	300	01 00101100	01001011
80	800	320	01 01000000	01010000
90	900	360	01 01101000	01011010
100	1000	400	01 10010000	01100100

Example 13-10

In Table 13-9, verify the PIC output for a temperature of 70 degrees. Find values in the PIC18 A/D registers of ADRESL and ADRESH.

Solution:

The step size is 2.56/1,024 = 2.5 mV because Vref = 2.56 V.
For the 70 degrees temperature we have 700 mV output because the LM34 provides 10 mV output for every degree. Now, the number of steps are 700 mV/2.5 mV = 300 in decimal. Now 300 = 0101000000 in binary and the PIC18 A/D output registers have ADRESL = 0100000 and ADRESH = 00000001.

Reading and displaying temperature

Programs 13-4 and 13-4C show code for reading and displaying temperature in both Assembly and C respectively.

The programs correspond to Figure 13-14. Regarding these two programs, the following points must be noted:

(1) The LM34 (or LM35) is connected to channel 0 (RA0 pin).

(2) The channel AN3 (RA3 pin) is connected to the Vref of 2.56 V. That makes PCFG = 0010 for the ADCON1 register.

(3) The 10-bit output of the A/D is divided by 4 to get the real temperature.

The algorithm is as follows: (a) Shift right the ADRESL 2 bits, (b) rotate the ADRESH 2 bits, and (c) OR the ADRESH with ADRESL together to get the 8-bit output for temperature.

```
;Program 13-4
;this program reads the sensor and displays it on PORTD
L_Byte     SET 0x20    ;set a location 0x20 for L_Byte
H_Byte     SET 0x21    ;set a location 0x21 for H_Byte
BIN_TEMP   SET 0x22    ;set a location 0x22 for BIN_TEMP
      CLRF  TRISD      ;make PORTD an output
      BSF   TRISA,0 ;make RA0 an input pin for analog volt
      BSF   TRISA,3 ;make RA3 an input pin for Vref volt
      MOVLW 0x81             ;Fosc/64, channel 0, A/D is on
      MOVWF ADCON0
      MOVLW 0xC5             ;right justified, Fosc/64,
      MOVWF ADCON1           ;AN0 = analog, AN3 = Vref+
OVER  CALL  DELAY       ;wait for Tacq (sample and hold time)
      BSF   ADCON0,GO          ;start conversion
BACK  BTFSC ADCON0,DONE;keep polling end-of-conversion(EOC)
      BRA   BACK               ;wait for end-of-conversion
      MOVFF ADRESL,L_Byte      ;save the low byte
      MOVFF ADRESH,H_Byte      ;save the high byte
      CALL  ALGO_10_to_8       ;make it an 8-bit value
      MOVFF BIN_TEMP,PORTD     ;display the temp on PORTD
      BRA   OVER               ;keep repeating it
;----------------
ALGO_10_to_8
      RRNCF L_Byte,F           ;rotate right twice
      RRNCF L_Byte,W
      ANDLW 0x3F               ;mask the upper 2 bits
      MOVWF L_Byte
      RRNCF H_Byte,F    ;rotate right through carry twice
      RRNCF H_Byte,W
      ANDLW 0xC0               ;mask the lower 6 bits
      IORWF L_Byte,W           ;combine low and high
      MOVWF BIN_TEMP
      RETURN
;-----------
```

```
//Program 13-4C
void main(void)
{
  unsigned char L_Byte,H_Byte,Bin_Temp;
  TRISD=0;              //make PORTD output port
  TRISAbits.TRISA0=1; //RA0 = INPUT for analog input
```

```
    TRISAbits.TRISA2=1; //RA2 = INPUT for vref input
    ADCON0 = 0x81;      //Fosc/64, channel 0, A/D is on
    ADCON1 = 0xC5;      //right justified, Fosc/64,
                        //AN0 = analog, AN3 = Vref+
    while(1)
    {
      MSDelay(1);       //give A/D channel time to sample
      ADCON0bits.GO = 1;     //start converting
      while(ADCON0bits.DONE == 1);      //wait for EOC
      L_Byte=ADRESL;    //save the low byte
      H_Byte=ADRESH;    //save the high byte
      L_Byte>>=2;       //shift right
      L_Byte&=0x3F;     //mask the upper 2 bits
      H_Byte<<=6;       //shift left 6 times
      H_Byte&=0xC0;     //mask the lower 6 bits
      Bin_Temp= L_Byte|H_Byte;
      PORTD=Bin_Temp;
    }
}
```

Review Questions

1. True or false. The transducer must be connected to signal conditioning circuitry before it is sent to the ADC.
2. The LM35 provides _____ mV for each degree of _____ (Fahrenheit, Celsius) temperature.
3. The LM34 provides ____ mV for each degree of _____ (Fahrenheit, Celsius) temperature.
4. Why do we set the V_{ref} of the PIC to 2.56 V if the analog input is connected to the LM35?
5. In Question 4, what is the temperature if the ADC output is 0011 1001?

SUMMARY

This chapter showed how to interface real-world devices such as DAC chips, ADC chips, and sensors to the PIC. First, we discussed both parallel and serial ADC chips, then described how the ADC module inside the PIC18 works and explained how to program it in both Assembly and C. Next we explored the DAC chip, and showed how to interface it to the PIC. In the last section we studied sensors. We also discussed the relation between the analog world and a digital device, and described signal conditioning, an essential feature of data acquisition systems.

PROBLEMS

SECTION 13.1: ADC CHARACTERISTICS

1. True or false. Sensor output is in analog.
2. True or false. A 10-bit ADC has 10-bit digital output.
3. True or false. ADC0848 is an 8-bit ADC.

4. True or false. MAX1112 is a 10-bit ADC.
5. True or false. An ADC with 8 channels of analog input must have 8 pins, one for each analog input.
6. True or false. For a serial ADC, it takes a longer time to get the converted digital data out of the chip.
7. True or false. ADC0848 has 4 channels of analog input.
8. True or false. MAX1112 has 8 channels of analog input.
9. True or false. ADC0848 is a serial ADC.
10. True or false. MAX1112 is a parallel ADC.
11. Which of the following ADC sizes provides the best resolution?
    (a) 8-bit (b) 10-bit (c) 12-bit (d) 16-bit (e) They are all the same.
12. In Question 11, which provides the smallest step size?
13. Calculate the step size for the following ADCs, if V_{ref} is 5 V:
    (a) 8-bit (b) 10-bit (c) 12-bit (d) 16-bit
14. With V_{ref} = 1.28 V, find the V_{in} for the following outputs:
    (a) D7–D0 = 11111111 (b) D7–D0 = 10011001 (c) D7–D0 = 1101100
15. In the ADC0848, what should be the Vref value if we want a step size of 5 mV?
16. With Vref+ = 2.56 V and Vref– = Gnd, find the V_{in} for the following outputs:
    (a) D7–D0 = 11111111 (b) D7–D0 = 10011001 (c) D7–D0 = 01101100

SECTION 13.2: ADC PROGRAMMING IN THE PIC18

17. True or false. The PIC18F452/458 has an on-chip A/D converter.
18. True or false. A/D of the PIC18 is an 8-bit ADC.
19. True or false. PIC18F452/458 has 8 channels of analog input.
20. True or false. The unused analog pins of the PIC18F452/458 can be used for I/O pins.
21. True or false. The A/D conversion speed in the PIC18F452/458 depends on the crystal frequency.
22. True or false. Upon power-on reset, the A/D module of the PIC18F452/458 is turned on and ready to go.
23. True or false. The A/D module of the PIC18F452/458 has an external pin for the start-conversion signal.
24. True or false. The A/D module of the PIC18F452/458 can convert only one channel at a time.
25. True or false. The A/D module of the PIC18F452/458 can have multiple external Vref+ at any given time.
26. True or false. The A/D module of the PIC18F452/458 can use the Vdd for Vref+.
27. In the A/D of PIC18 what happens to the converted analog data? How do we know that the ADC is ready to provide us the data?
28. In the A/D of PIC18 what happens to the old data if we start conversion again before we pick up the last data?
29. Assume Vref– = Gnd. For the A/D of PIC18, find the step size for each of the following V_{ref+}:
    (a) V_{ref} = 1.024 V (b) V_{ref} = 2.048 V (c) V_{ref} = 2.56 V

30. In the PIC18, what should be the Vref value if we want a step size of 2 mV?
31. In the PIC18, what should be the Vref value if we want a step size of 3 mV?
32. With a step size of 1 mV, what is the analog input voltage if all outputs are 1?
33. With V_{ref} = 1.024 V, find the V_{in} for the following outputs:
    (a) D9–D0 = 0011111111 (b) D9–D0 = 0010011000 (c) D9–D0 = 0011010000
34. In the A/D of PIC18, what should be the Vref value if we want a step size of 4 mV?
35. With Vref+ = 2.56 V and Vref– = Gnd, find the V_{in} for the following outputs:.
    (a) D9–D0 = 1111111111 (b) D9–D0 = 1000000001 (c) D9–D0 = 1100110000
36. Find the conversion time for the following cases if XTAL = 8 MHz:
    (a) Fosc/2 (b) Fosc/4 (c) Fosc/8 (d) Fosc/16 (e) Fosc/32
37. Find the conversion time for the following cases if XTAL = 12 MHz:
    (a) Fosc/8 (b) Fosc/16 (c) Fosc/32 (d) Fosc/64
38. How do we start conversion in the PIC18?
39. How do we recognize the end of conversion in the PIC18?
40. The PIC18F452/458 can have a minimum of _____channels of analog input.
41. In the PIC18F452/458, what ports are used for the analog channels?
42. Which register of the PIC18 is used to designate the number of A/D channels?
43. Which register of the PIC18 is used to select the A/D's conversion speed?
44. Which register of the PIC18 is used to select the analog channel to be converted?
45. Find the value for the ADCON0 register if we want Fosc/8, channel 0, and ADON on.
46. Find the value for the ADCON1 register if we want Fosc/64, 3 channels of analog input, and right-justified output.
47. Find the value for the ADCON0 register if we want Fosc/2, channel 2, and ADON off.
48. Find the value for the ADCON1 register if we want Fosc/32, 2 channels of analog input with external source for Vref+, and left-justified output.
49. Give the name of the interrupt flags for the A/D of the PIC18F452/458. State to which register they belong.
50. Upon power-on reset, the A/D of the PIC18F452/458 is given (low, high) priority.

SECTION 13.3: DAC INTERFACING

51. True or false. DAC0808 is the same as DAC1408.
52. Find the number of discrete voltages provided by the *n*-bit DAC for the following:
    (a) $n = 8$ (b) $n = 10$ (c) $n = 12$
53. For DAC1408, if I_{ref} = 2 mA, show how to get the I_{out} of 1.99 when all inputs are HIGH.
54. Find the I_{out} for the following inputs. Assume I_{ref} = 2 mA for DAC0808.
    (a) 10011001 (b) 11001100 (c) 11101110
    (d) 00100010 (e) 00001001 (f) 10001000

55. To get a smaller step, we need a DAC with _____ (more, fewer) digital inputs.
56. To get full-scale output, what should be the inputs for DAC?

SECTION 13.4: SENSOR INTERFACING AND SIGNAL CONDITIONING

57. What does it mean when a given sensor is said to have a linear output?
58. The LM34 sensor produces _____ mV for each degree of temperature.
59. What is signal conditioning?
60. What is the purpose of the LM336 Zener diode around the pot setting the V_{ref} in Figure 13-14?

ANSWERS TO REVIEW QUESTIONS

SECTION 13.1: ADC CHARACTERISTICS

1. Number of steps and Vref voltage
2. 8
3. True
4. (a) 8 (b) 8
5. 1.28 V/256 = 5 mV
6. (a) 0.7 V/ 5 mV= 140 in decimal and D7–D0 = 10001100 in binary.
   (a) 1 V/ 5 mV= 200 in decimal and D7–D0 = 11001000 in binary.

SECTION 13.2: ADC PROGRAMMING IN THE PIC18

1. Vref
2. 10
3. False
4. False
5. 1 mV
6. (a) 700 mV (1010111100), (b) 1000 mV (1111101000)
7 (a) 2 channels (b) 6 channels
8. True
9. 1.6
10. DONE bit of the ADCON0 register

SECTION 13.3: DAC INTERFACING

1. Digital, analog
2. Analog, digital
3. 8
4. (a) current (b) true

SECTION 13.3: SENSOR INTERFACING AND SIGNAL CONDITIONING

1. True
2. 10, Celsius
3. 10, Fahrenheit
4. Using the 8-bit part of the 10-bit ADC, it gives us 256 steps, and 2.56 V/256 = 10 mV. The LM35 produces 10 mV for each degree of temperature, which matches the ADC's step size.
5. 00111001 = 57, which indicates it is 57 degrees.

CHAPTER 15

CCP AND ECCP PROGRAMMING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Understand the compare and capture features of the PIC18**
- **>> Examine the use of timers in CCP and ECCP modules**
- **>> Explain how the compare feature of CCP and ECCP modules works**
- **>> Explain how the capture feature of CCP and ECCP modules works**
- **>> Code programs for compare and capture features in Assembly and C**
- **>> Explain how the PWM (pulse width modulation) works in both CCP and ECCP**
- **>> Code programs to create PWM in Assembly and C**

This chapter discusses the capture/compare/pulse width modulation (CCP) features of the PIC18. In Section 15.1, we show the difference between standard and enhanced CCP modules. In Section 15.2, we describe the compare feature while Section 15.3 deals with the capture feature of the PIC18. The pulse width modulation (PWM) of the PIC18 is shown in Section 15.4. An overview of ECCP is given in Section 15.5. In all these sections we use both Assembly and C language programs to show these important features of the PIC18.

SECTION 15.1: STANDARD AND ENHANCED CCP MODULES

Depending on the family member, the PIC18 has anywhere between 0 and 5 CCP modules inside it. The multiple CCP modules are designated as CCP1, CCP2, CCP3, and so on (CCPx). In recent years, the PWM feature of the CCP has been enhanced greatly for better DC motor control, producng what is called enhanced CCP (ECCP). Therefore, a given family member can have two standard CCP modules and one or more ECCP modules, all on a single chip. See Table 15-1. The ECCP modules are discussed in Chapter 17.

CCP and timers

Table 15-1: PIC18 CCP and ECCP Modules

Chip	# of CCP	# of ECCP
PIC18F2220	2	0
PIC18F4220	1	1
PIC18F452/4520	1	1
PIC18F458/4580	1	1
PIC18F65J10	2	3

To program these CCP modules, we must understand PIC18 timers. Review timers in Chapter 9 before you embark on this chapter. Depending on the CCP feature used, the timer usage is different. The allocation of the timers among the CCP features is shown in Table 15-2.

Table 15-2: PIC18 Usage of Timers

CCP mode	Timer
Capture	Timer1 or Timer3
Compare	Timer1 or Timer3
PWM	Timer2

Note: The T3CON register is used to choose the timer for the compare and capture modes.

The CCP registers

Each CCP module has three registers associated with it. They are as follows:

(a) CCPxCON is an 8-bit control register. We select one of the compare, capture, and PWM modes using this register. See Figure 15-1.

(b) and (c) CCPRxL and CCPxH form the low byte and the high byte of the 16-bit register. This 16-bit register can be used either as a 16-bit compare register, or a 16-bit capture register, or an 8-bit duty cycle register by the PWM, but not all at the same time. See Figures 15-1 and 5-2. The CCP1CON register selects the mode of operation.

-	-	DC1B1	DC1B2	CCP1M3	CCP1M2	CCP1M1	CCP1M0

DC1B1 Duty Cycle Bit 1. Used only in PWM mode.
Bit 1 of the 10-bit duty cycle register used in PWM

DC1B0 Duty Cycle Bit 0. Used only in PWM mode.
The least-significant bit (bit 0) of the 10-bit duty cycle register. Used in PWM. The CCPxL register is used as bit 2 to bit 9 of the 10-bit duty cycle register.

CCP1M3–CC1M0 CCP1 Mode Select

0 0 0 0	CCP1 is off
0 0 0 1	Reserved
0 0 1 0	Compare mode. Toggle CCP1 output pin on match. (CCP1IF bit is set.)
0 0 1 1	Reserved
0 1 0 0	Capture mode, every falling edge
0 1 0 1	Capture mode, every rising edge
0 1 1 0	Capture mode, every 4th rising edge
0 1 1 1	Capture mode, every 16th rising edge
1 0 0 0	Compare mode. Initialize CCP1 pin LOW, on compare match force CCP1 pin HIGH. (CCP1IF is set.)
1 0 0 1	Compare mode. Initialize CCP1 pin HIGH, on compare match force CCP1 pin LOW. (CCP1IF is set.)
1 0 1 0	Compare mode. Generate software interrupt on compare match. (CCP1IF bit is set, CCP1 pin is unaffected.)
1 0 1 1	Compare mode. Trigger special event. (CCP1IF bit is set, and Timer1 or Timer3 is reset to zero.)
1 1 x x	PWM mode

Figure 15-1. CCP1 Control Register. (This register selects one of the operation modes of Capture, Compare, or PWM.)

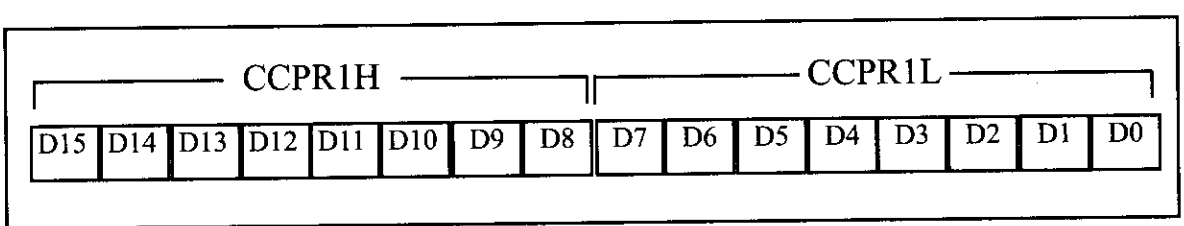


Figure 15-2. CCP High and Low Registers

CCP pins

Each CCP module has a single pin assigned to it. That means that a PIC18 family member with two standard CCP modules (e.g., PIC18F65J10) has two pins, one assigned to each of the CCPs. See Figure 15-3. In the case of the enhanced CCP (ECCP), although it has a single pin, we can program up to four pins to be used by the PWM feature of the ECCP, as we will see at the end of this chapter and in Chapter 17.

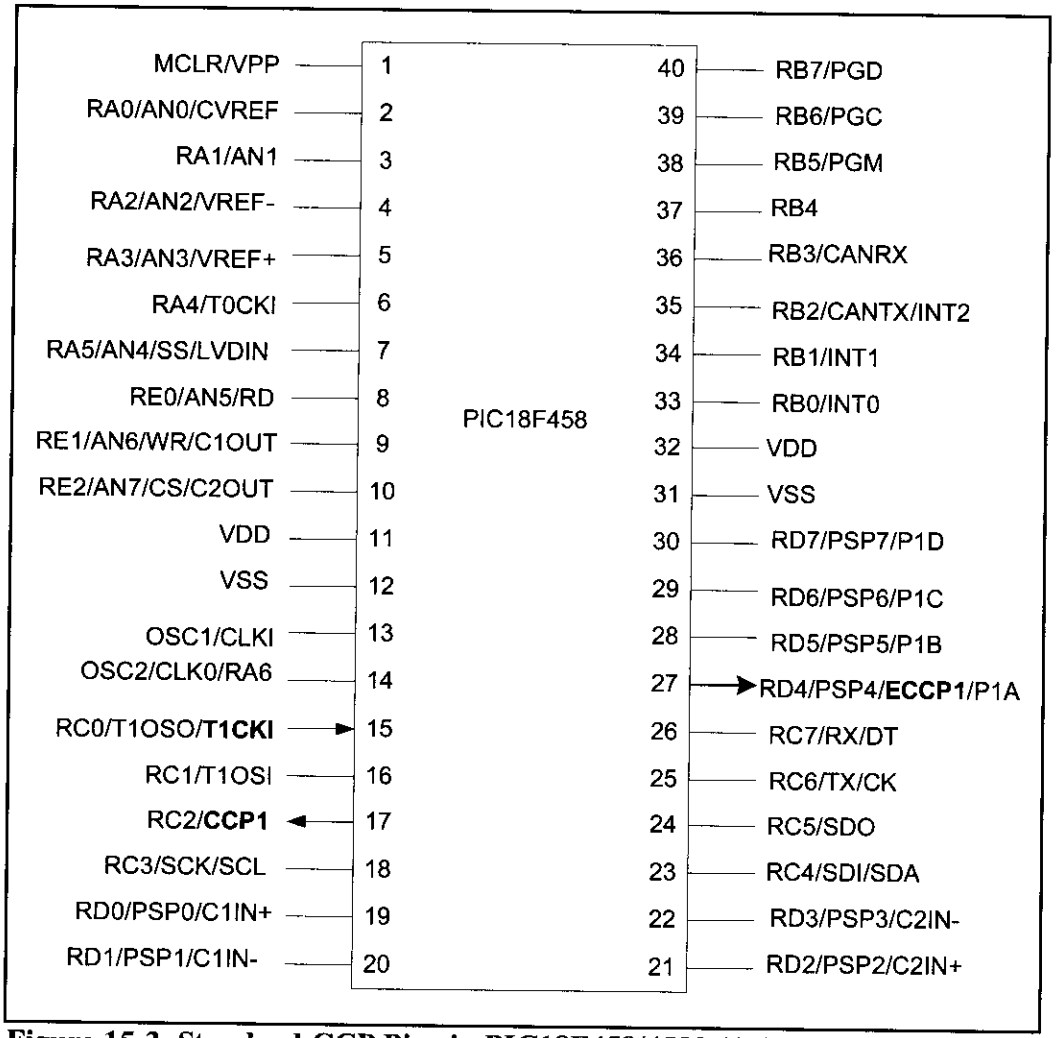


Figure 15-3. Standard CCP Pins in PIC18F458/4580 (452/4520)

Review Questions

1. True or false. The PIC18 chip can have multiple CCP modules inside a single chip.
2. True or false. The CCP1 register is a 16-bit register.
3. True or false. A single pin is associated with each of the standard CCP modules.
4. Give the pin number used for the standard CCP1 in the PIC18F452/458 (or PIC18F4520/4580) chip.

SECTION 15.2: COMPARE MODE PROGRAMMING

The Compare mode of the CCP module is selected using the select bits in the CCPxCON register. The Compare mode can cause an event outside the microcontroller. This event can be simply turning on a device connected to the CCP pin, or the start of an ADC conversion. This event is caused when the content of the Timer1 (or Timer3) register is equal to the 16-bit CCPR1H:CCPR1L register. To use the compare mode of the CCP, we must load both the 16-bit (CCP1H:CCP1L) and the Timer1 (or Timer3) register with some initial values. As Timer1 (or Timer3) counts up, its value is constantly compared with the CCPR1H:CCPR1L register and when a match occurs, the CCP1 pin can perform one of the following actions:

(a) Drive high the CCP1 pin
(b) Drive low the CCP1 pin
(c) Toggle the CCP1 pin
(d) Remain unaffected
(e) Trigger a special event with a hardware interrupt and clear the timer

We use the CCP1CON register to select one of the above actions. See Example 15-1. Note that upon match, the CCP1IF will also go HIGH. See Figure 15-5. Notice that for the above options of (a), (b), and (c) to work, the CCP pin must be configured as an output pin. From Figure 15-4 we use the T3CON register to select Timer1 or Timer3 for the Compare mode. In PIC18F452/458 (or their newer version, PIC18F4520/4580) chips with both CCP1 and ECCP1 modules on the chip, we can assign Timer1 to CCP1 and Timer3 to ECCP1, therefore making them work independently of each other. Also note that only option (e), the special event trigger, will cause Timer1 (or Timer3) to clear, while in other cases we must clear the timer.

Example 15-1

Using Figures 15-1 and 15-4, find the following:
(a) The CCP1CON register value for Compare mode if we want to toggle the CCP1 pin upon match
(b) The T3CON register value if we want to to use Timer3 for the Compare mode of CCP1 with no prescaler

Solution:

(a) From Figure 15-1 we have 00000010 (binary) or 0x20 for the CCP1CON register.

(b) From Figure 15-4 we have 01000010 (binary) or 0x42 for the T3CON register.

There are many applications for the compare feature. One application can be to count the number of people going through a door and closing the door when it reaches a certain number.

RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	T3SYNC	TMR3CS	TMR3ON

RD16 D7 16-bit read/write enable bit
1 = Timer3 16-bit is accessible in one 16-bit operation.
0 = Timer3 16-bit is accessible in two 8-bit operations.

T3CCP2:T3CCP1 D6 D3 assigns Timer3 or Timer1 to CPP1 and CCP2 modules

	CCP1	ECCP1 (or CCP2)	
0 0 =	Timer1	Timer1	(clock source for compare/capture)
0 1 =	Timer1	Timer3	(clock source for compare/capture)
1 x =	Timer3	Timer3	(clock source for compare/capture)

T3CKPS1:T3CKPS0 D5 D4 Timer3 Input Clock Prescaler Selector

0 0 = 1:1 Prescale value
0 1 = 1:2 Prescale value
1 0 = 1:4 Prescale value
1 1 = 1:8 Prescale value

T1SYNC D2 Timer3 External Clock Input Synchronization Control bit
Used only when TMR3CS = 1 and clock comes from an external source. If TMR3CS = 0, this bit is not used.
1 = Do not synchronize external clock input.
0 = Synchronize external clock input.

TMR3CS D1 Timer 3 Clock Source Select bit
1 = External clock from pin RC0 (T1OSI or T1CKI)
0 = Internal clock (Fosc/4)

TMR3ON D0 Timer3 ON and OFF Control bit
1 = Enable (start) Timer3
0 = Stop Timer3

Figure 15-4. T3CON (Timer 3 Control) Register

CCP1IF CCP1IF Interrupt Flag bit
Compare Mode
0 = Timer1 (or Timer3) match did not occur.
1 = Timer1 (or Timer3) match occurred (must be cleared by software).
Capture Mode
0 = Timer1 (or Timer3) register capture did not occur.
1 = Timer1 (or Timer3) register capture occurred (must be cleared by software).

Figure 15-5. PIR1 (Peripheral Interrupt flag register 1) Contains the CCP1IF Flag

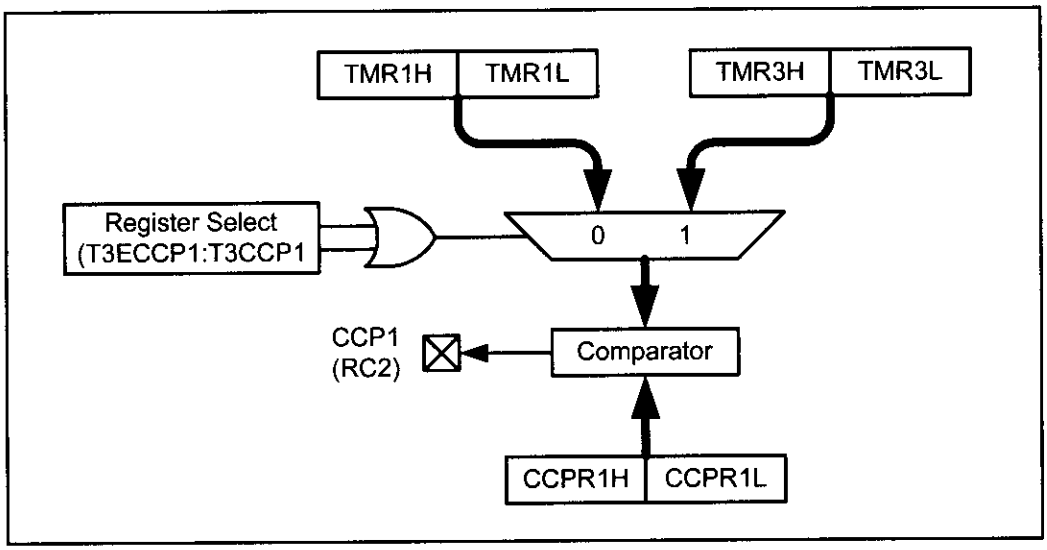


Figure 15-6. Compare Mode Operation

Steps for programming the Compare mode

The following steps are taken in programming the Compare mode:

1. Initialize the CCP1CON register for the compare option.
2. Initialize the T3CON register for Timer1 (or Timer3).
3. Initialize the CCPR1H:CCPR1L registers.
4. Make the CCP1 pin an output pin.
5. Initialize Timer1 (or Timer3) register values.
6. Start Timer1 (or Timer3).
7. Monitor the CCP1IF flag (or use an interrupt).

Program 15-1 shows an example of the Compare mode. It uses Timer3 as a counter and counts the number of pulses fed to Timer3. The pulses could be the number of people going into an elevator. When the count reaches 10, it toggles the LED connected to the CCP1 pin.

For Program 15-1 assume that a 1-Hz pulse is connected to the Timer3 pin and an LED is connected to the CCP1 pin. Timer3 is being used as a counter. Using the Compare mode, this Assembly language program will toggle the LED every 10 pulses.

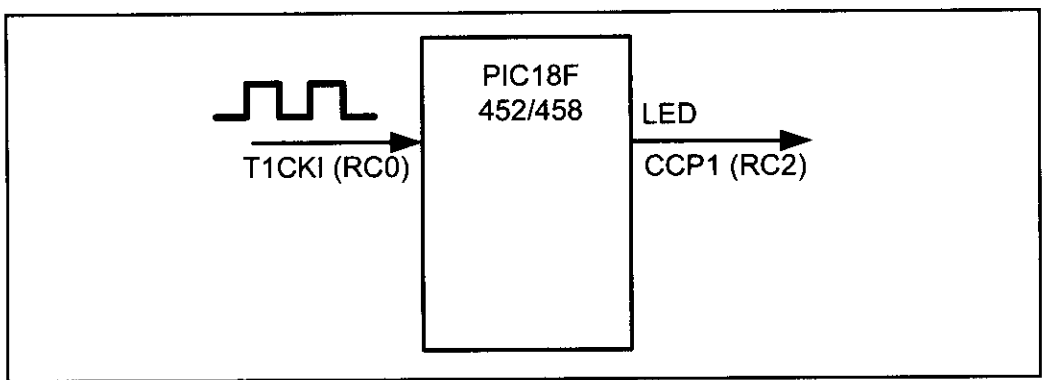


Figure 15-7. Drawing for Programs 15-1 and 15-1C

```
;Program 15-1
      MOVLW  0x02
      MOVWF  CCP1CON          ;Compare mode, toggle upon match
      MOVLW  0x42
      MOVWF  T3CON      ;Timer3 for Compare, 1:1 prescaler
      BCF    TRISC,CCP1       ;CCP1 pin as output
      BSF    TRISC,T1CKI      ;T3CLK pin as input pin
      MOVLW  D'10'
      MOVWF  CCPR1L           ;CCPR1L = 10
      MOVLW  0x0              ;CCPR1H = 0
      MOVWF  CCPR1H
 OVER CLRF   TMR3H            ;clear TMR3H
      CLRF   TMR3L            ;clear TMR3L
      BCF    PIR1,CCP1IF      ;clear CCP1IF
      BSF    T3CON,TMR3ON     ;start Timer3
 B1   BTFSS  PIR1,CCP1IF
      BRA    B1
 ;---------CCP toggle CCP pin upon match
 B2   BCF    T3CON,TMR3ON     ;stop Timer3
      GOTO   OVER             ;keep doing it

 //Program 15-1C is a C version of Program 15-1
    CCP1CON=0x02;             //Compare mode, toggle upon match
    T3CON=0x42;       //Timer3 for Compare, 1:1 prescaler
    TRISCbits.TRISC2=0; //CCP1 pin an output
    TRISCbits.TRISC0=1;     //T3CLK pin an input
    CCPR1L=10;              //load CCPR1L
    CCPR1H=0;               //load CCPR1H
    while(1)
    {
      TMR3H=0;
      TMR3L=0;
      PIR1bits.CCP1IF=0;    //clear CCP1IF flag
      T3CONbits.TMR3ON=1;  //turn on Timer3
      while(PIR1bits.CCP1IF==0);//wait for CCP1IF
      //CCP toggles CCP pin upon match
      T3CONbits.TMR3ON=0;  //stop Timer3
    }
```

Examine Program 15-2: For this program we assume that the PIC18452/458 has Fosc = 10 MHz. It programs the CCP1 module in Compare mode to create a square wave with a period of 40 ms on the CCP1 pin continuously. The square wave has a 50% duty cycle, which means it is high 50% of each period. This is an example of how Timer1 is used in compare mode. See Figure 15-8. Because the timer uses the Fosc/4, we have 1/2.5 MHz = 0.4 μs for the clock. A 40 ms period gives us 20 ms for high and low portions of the square wave. Now 20 ms / 0.4 μs = 50,000 or C350 in hex. This is the value we load into CCPR1H:CCPR1L for the Compare mode.

RD16	---	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON

RD16 D7 16-bit read/write enable bit
1 = Timer1 16-bit is accessible in one 16-bit operation.
0 = Timer1 16-bit is accessible in two 8-bit operations.

D6 Not used

T1CKPS2:T1CKPS0 D5 D4 Timer1 prescaler selector
0 0 = 1:1 Prescale value
0 1 = 1:2 Prescale value
1 0 = 1:4 Prescale value
1 1 = 1:8 Prescale value

T1OSCEN D3 Timer1 oscillator enable bit
1 = Timer1 oscillator is enabled
0 = Timer1 oscillator is shut off

T1SYNC D2 Timer1 synchronization (used only when TMR1CS = 1 for counter mode to synchronize external clock input)
If TMR1CS = 0, this bit is not used.

TMR1CS D1 Timer1 clock source select bit
1 = External clock from pin RC0/T1CKI
0 = Internal clock (Fosc/4 from XTAL)

TMR1ON D0 Timer1 ON and OFF control bit
1 = Enable (start) Timer 1
0 = Stop Timer 1

Figure 15-8. T1CON (Timer 1 Control) Register

Program 15-2 creates a square wave with a 40 ms period and 50% duty cycle on CCP1 pin using the Compare mode.

```
;Program 15-2
     MOVLW 0x02
     MOVWF CCP1CON  ;Compare mode, toggle upon match
     MOVLW 0x0
     MOVWF T3CON    ;use Timer1 for Compare mode
     MOVLW 0x0
     MOVWF T1CON    ;Timer1, internal CLK, 1:1 prescale
     BCF   TRISC,CCP1    ;CCP1 pin as output
     MOVLW 0xC3
     MOVWF CCPR1H   ;CCPR1H = 0xC3
     MOVLW 0x50
```

```
      MOVWF CCPR1L          ;CCPR1L = 0x50
OVER  CLRF  TMR1H           ;clear TMR1H
      CLRF  TMR1L           ;clear TMR1L
      BCF   PIR1,CCP1IF     ;clear CCP1IF
      BSF   T1CON,TMR1ON    ;start Timer1
B1    BTFSS PIR1,CCP1IF
      BRA   B1
      ;CCP toggles CCP1 pin upon match
      BCF   T1CON,TMR1ON    ;stop Timer1
      GOTO  OVER            ;keep doing it
```

```
//Program 15-2C is the C version of Program 15-2.
 CCP1CON=0x02;        //Compare mode, toggle upon match
 T3CON=0x0;           //Timer1 for Compare, 1:1 prescaler
 T1CON=0x0;           //Timer1 internal clk, 1:1 prescaler
 TRISCbits.TRISC2=0; //make CCP1 pin an output
 TRISCbits.TRISC0=1; //make T1CLK pin an input
 CCPR1H=0xC3;         //load CCPR1H
 CCPR1L=0x50;         //load CCPR1L
 while(1)
 {
   TMR1H=0;                    //clear Timer1
   TMR1L=0;
   PIR1bits.CCP1IF=0;          //clear CCP1IF flag
   T1CONbits.TMR1ON=1;         //turn on Timer1
   while(PIR1bits.CCP1IF==0);  //wait for CCP1IF
   //CCP toggles CCP1 pin upon match
   T1CONbits.TMR1ON=0;         //stop Timer1
 }
```

Review Questions

1. True or false. We can use any timers we want for the Compare mode.
2. True or false. There is a single pin associated with the Compare mode.
3. True or false. In Compare mode, the CCP pin must be configured as an input pin.
4. Which register is used to choose the timer for the Compare mode?

SECTION 15.3: CAPTURE MODE PROGRAMMING

We select Capture mode with the bit selection in the CCP1CON register. In Capture mode, an event at the CCP pin will cause the contents of the Timer1 (or Timer3) register to be loaded into the 16-bit CCPR1H:CCPR1L register. That means, for the Capture mode to work, the CCP pin must be configured as an input pin. The event that causes the contents of Timer1 (or Timer 3) to be captured into the CCPR1H:CCPR1L register can be a High-to-Low (falling-edge) pulse or Low-to-High (rising-edge) pulse. As far as the edge-triggering pulse is concerned, we have the following four options to choose from:

(a) every falling-edge pulse
(b) every rising-edge pulse
(c) every fourth rising-edge pulse
(d every 16th rising-edge pulse

One of the above options can be chosen by selection bits in the CCP1CON register. See Example 15-2. Notice that for any of the above options to work, the CCP pin must be configured as an input pin.

Example 15-2

Using Figures 15-1 and 15-4, find the following:
(a) The CCP1CON register value for Capture mode if we want to capture on the rising edge of every pulse
(b) The T3CON register value if we want to to use Timer3 for Capture mode of the CCP1 with no prescaler

Solution:

(a) From Figure 15-1, we have 00000101 (binary) or 0x05 for the CCP1CON register.
(b) From Figure 15-4, we have 01000010 (binary) or 0x42 for the T3CON register.

One application of Capture mode is measuring the frequency of an incoming pulse. See Program 15-3.

Steps for programming Capture mode

The following steps are used in programming Capture mode for measuring the period of a pulse:

1. Initialize the CCP1CON register for capture.
2. Make the CCP1 pin an input pin.
3. Initialize the T3CON register to select Timer1 or Timer3.
4. Read the Timer1 (or Timer3) register value on the first rising edge and save it.
5. Read the Timer1 (or Timer3) register value on the second rising edge and save it.
6. Subtract the value in step 4 from the value in step 5.

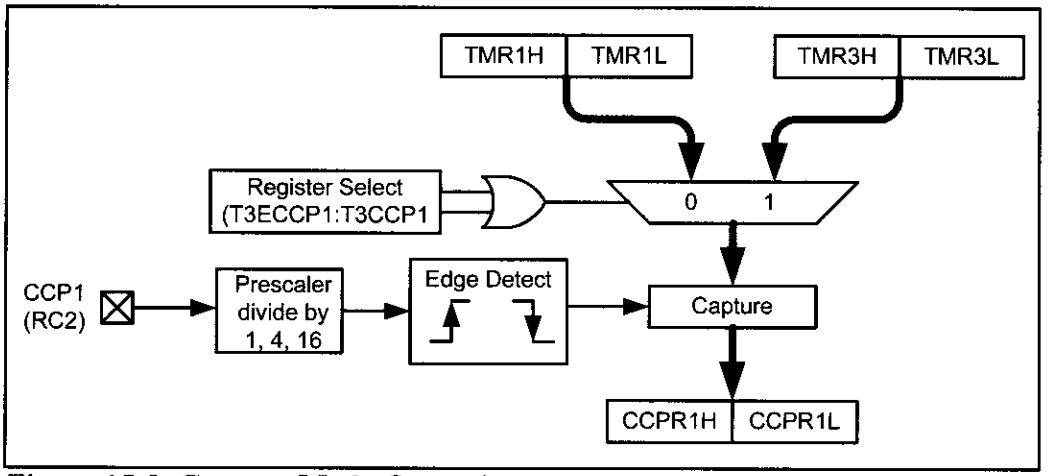


Figure 15-9. Capture Mode Operation

Measuring the period of a pulse

Program 15-3 shows an example of capture mode. See Figure 15-9. It measures the period of the pulse fed to the CCP pin. The measurement is in terms of the number of clock cycles, Tsclk (1/(Fosc/4)). See Figure 15-10.

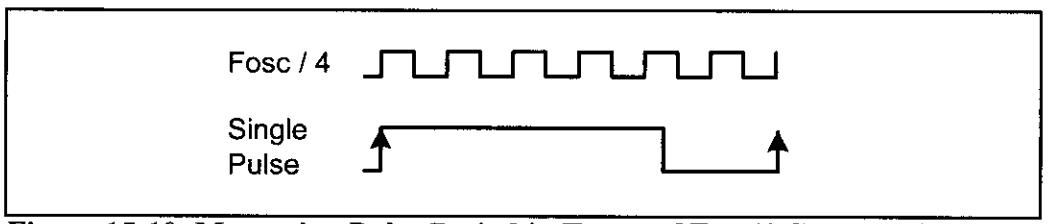


Figure 15-10. Measuring Pulse Period in Terms of Fosc/4 Clock Period

For Program 15-3 assume a pulse is being fed to the CCP1 pin. Using Capture mode, this Assembly language program measures the period of the pulse and puts the results on PORTB and PORTD. The measurement is in terms of the Fosc/4 clock period. See Figure 15-11.

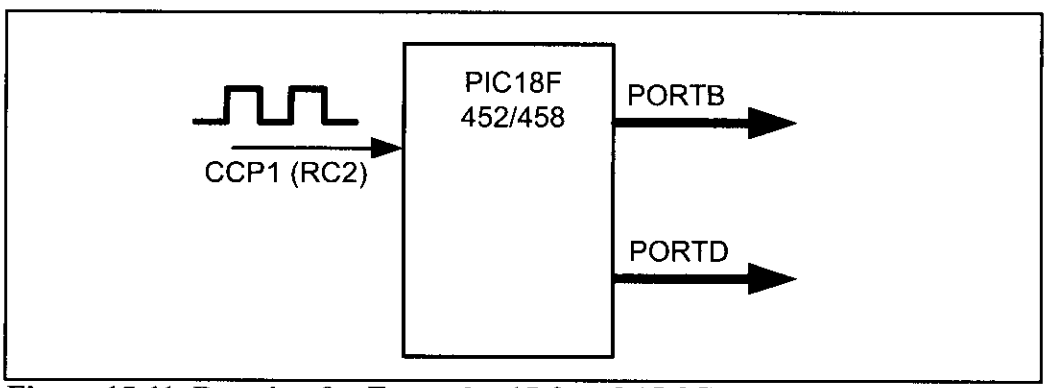


Figure 15-11. Drawing for Examples 15-3 and 15-3C.

```
;Program 15-3
      MOVLW  0x05
      MOVWF  CCP1CON          ;Capture mode rising edge
      MOVLW  0x0
      MOVWF  T3CON            ;Timer1 for Capture
      MOVLW  0x0
      MOVWF  T1CON      ;Timer1, internal CLK, 1:1 prescale
      CLRF   TRISB            ;make PORTB output port
      CLRF   TRISD            ;make PORTD output port
      BSF    TRISC,CCP1       ;make CCP1 pin an input
      MOVLW  0x0
      MOVWF  CCPR1H           ;CCPR1H = 0
      MOVWF  CCPR1L           ;CCPR1L = 0
OVER  CLRF   TMR1H            ;clear TMR1H
      CLRF   TMR1L            ;clear TMR1L
      BCF    PIR1,CCP1IF      ;clear CCP1IF
RE_1  BTFSS  PIR1,CCP1IF
      BRA    RE_1             ;stay here for 1st rising edge
      BSF    T1CON,TMR1ON     ;start Timer1
      BCF    PIR1,CCP1IF      ;clear CCP1IF for next
RE_2  BTFSS  PIR1,CCP1IF
      BRA    RE_2             ;stay here for 2nd rising edge
      BCF    T1CON,TMR1ON     ;stop Timer1
      MOVFF  TMR1L,PORTB      ;put low byte on PORTB
      MOVFF  TMR1H,PORTD      ;put high byte on PORTD
      GOTO   OVER             ;keep doing it
```

```
//Program 15-3C is the C version of Program 15-3.
    CCP1CON=0x05;     //Capture mode on every rising edge
    T3CON=0x0;              //Timer1 for capture
    T1CON=0x0;        //Timer1 internal clk, 1:1 prescaler
    TRISB=0;                //make PORTB output port
    TRISD=0;                //make PORTD output port
    TRISCbits.TRISC2=1;     //make CCP1 pin an input
    CCPR1L=0;               //CCPR1L = 0
    CCPR1H=0;               //CCPR1H = 0
  while(1)
    {
    TMR1H=0;                //clear Timer1
    TMR1L=0;
    PIR1bits.CCP1IF=0;      //clear CCP1IF flag
    while(PIR1bits.CCP1IF==0);  //wait for 1st rising edge
    T1CONbits.TMR1ON=1;     //start Timer1
    PIR1bits.CCP1IF=0;      //clear CCPIF for next edge
    while(PIR1bits.CCP1IF==0);  //wait for 2nd rising edge
    T1CONbits.TMR1ON=0;     //stop Timer1
    PORTB=CCPR1L;
    PORTD=CCPR1H;           //display the clock count
    }
```

One problem in measuring the period in the above program is the rate of error due to overhead associated with the program. One way to reduce the effect of the overhead is to use every fourth or every sixteenth rising edge.

Measuring pulse width

One of the most widely used applications of Capture mode is measuring the pulse width. A large number of devices measure things such as distance, temperature, and so on, in which the quantity is provided in terms of the pulse width instead of traditional voltage or current. In these devices, the output is provided in pulse-width-modulated (PWM) form. In a device with PWM output, the output has a fixed frequency and the variable duty cycle provides the quantity we are measuring. For example, the MAX6666/6667 temperature sensors from Maxim Corp. "convert the ambient temperature into a ratiometric PWM output with temperature information contained in the duty cycle of the output square wave." According to their data sheets the output is a square wave with a nominal frequency of 35 Hz at +25°C. The output format is decoded as follows:

$$\text{Temperature (°C)} = 235 - (400 \times t1) / t2 \qquad \text{(Equation 15-1)}$$

where t1 is fixed with a typical value of 10 ms and t2 is modulated by the temperature. In the above formula, T = t1 + t2 where T is the period of the pulse, t1 is the high portion of the pulse, and t2 is the low portion, as shown in Figure 15-12. With t1 = 10 and t2 = 20 ms, we get temperature = 235 – (400 × 10 ms) / 20 ms = 235 – 200 = 35°C. Program 15-4 shows how to measure the duty cycle using Capture mode.

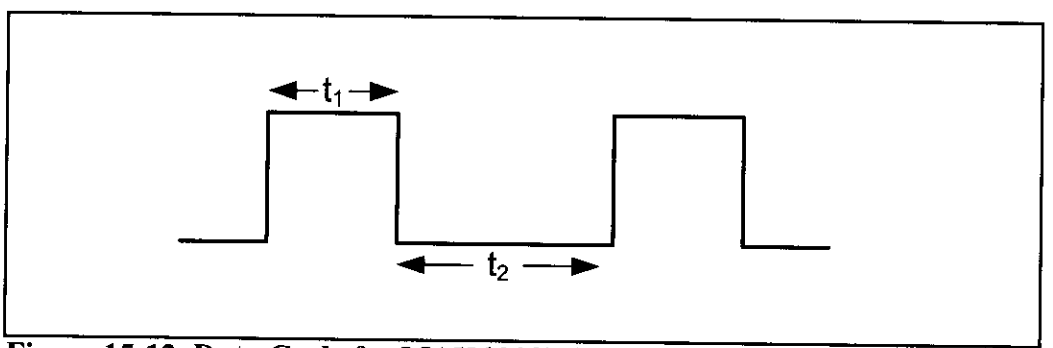


Figure 15-12. Duty Cycle for MAX6666/76 Temperature Sensor (Maxim Corp.)

```
;Program 15-4
FLAG EQU 0x10    ;flag register for steps in detection
DISP EQU 0x0     ;flag for capture complete
RF   EQU 0x1     ;flag for rising or falling edge
     ORG 0x0000
     GOTO MAIN
     ORG 0x0008
     BTFSC PIR1,CCP1IF      ;Is it CCP1?
     GOTO CCP_ISR           ;service CCP1
     RETFIE                 ;else return
MAIN MOVLW 0x05
     MOVWF CCP1CON          ;Capture mode rising edge
     MOVLW 0x0
     MOVWF T3CON            ;Timer1 for Capture
     MOVLW 0x0
     MOVWF T1CON ;Timer1, internal CLK, 1:1 prescale
     CLRF  TRISB            ;make PORTB output port
     CLRF  TRISD            ;make PORTD output port
     BSF   TRISC,CCP1       ;make CCP1 pin an input
     CLRF CCPR1H            ;CCPR1H = 0
     CLRF CCPR1L            ;CCPR1L = 0
     BSF   PIE1,CCP1IE      ;enable CCP1 interrupt
     BSF   INTCON,PEIE      ;enable peripheral interrupt
     BSF   INTCON,GIE       ;enable all interrupts
OVER CLRF  TMR1H            ;clear TMR1H
     CLRF  TMR1L            ;clear TMR1L
WAIT BTFSS FLAG,DISP        ;Is capture complete?
     BRA WAIT               ;else wait
     BCF FLAG,DISP          ;clear flag for next capture
     MOVLW 0x03
     SUBWF TMR1L,F          ;subtract the overhead
     MOVFF TMR1L,PORTB      ;put low byte on PORTB
     MOVFF TMR1H,PORTD      ;put high byte on PORTD
     GOTO OVER              ;keep doing it
CCP_ISR    BTFSS FLAG,RF    ;Is it rising edge?
     GOTO RISE_ISR          ;service rising edge
     GOTO FALL_ISR          ;else service falling edge
RISE_ISR   BSF  T1CON,TMR1ON    ;start Timer1
     BSF FLAG,RF            ;ready for falling edge
     BCF CCP1CON,0          ;detect falling edge
     BCF PIR1,CCP1IF ;clear interrupt
     RETFIE          ;return and wait for falling edge
FALL_ISR   BCF  T1CON,TMR1ON    ;stop Timer1
     BSF FLAG,DISP          ;capture complete
     BCF FLAG,RF            ;ready for rising edge
     BSF CCP1CON,0          ;detect rising edge
     BCF PIR1,CCP1IF ;clear interrupt
     RETFIE          ;return capture complete
     END
```

```
//Program 15-4C
#include "p18f458.h"

void CCP1_ISR(void);
void rising(void);
void falling(void);
unsigned char disp = 0;
unsigned char rf = 0;

#pragma interrupt chk_isr
void chk_isr (void)
{
  if (PIR1bits.CCP1IF==1)
    CCP1_ISR();
}

#pragma code My_HiPrio_Int=0x0008
void My_HiPrio_Int (void)
{
  _asm
  GOTO chk_isr
  _endasm
}

#pragma code
void main()
  {
    CCP1CON=0x05;          //Capture mode rising edge
    T3CON=0x0;             //Timer1 for Capture
    T1CON=0x0;        //Timer1, internal CLK, 1:1 prescale
    TRISB=0x0;             //make PORTB output port
    TRISD=0x0;             //make PORTD output port
    TRISCbits.TRISC2=1;    //make CCP1 pin an input
    CCPR1H=0x0;            //CCPR1H = 0
    CCPR1L=0x0;            //CCPR1L = 0
       PIE1bits.CCP1IE=1; //enable CCP1 interrupt
       INTCONbits.PEIE=1; //enable peripheral interrupt
       INTCONbits.GIE=1;  //enable all interrupts
       while(1)
       {
      TMR1H=0x0;           //clear TMR1H
         TMR1L=0x0;        //clear TMR1L
      while(disp==0);      //Is data ready to display?
         disp=0;           //clear the flag
         TMR1L-=15;        //subtract the overhead
         PORTB=TMR1L;      //put low byte on PORTB
         PORTD=TMR1H;      //put high byte on PORTD
       }
     }
```

```
void CCP1_ISR()
  {
    if(rf==0) rising();
       else falling();
     }

void rising()
  {
     T1CONbits.TMR1ON=1;          //start Timer1
     rf=1;                        //ready for falling edge
     CCP1CONbits.CCP1M0=0;        //detect falling edge
     PIR1bits.CCP1IF=0;           //clear interrupt
  }

void falling()
  {
     T1CONbits.TMR1ON=0;          //stop Timer1
     disp=1;                      //capture complete
     rf=0;                        //ready for rising edge
     CCP1CONbits.CCP1M0=1;        //detect rising edge
     PIR1bits.CCP1IF=0;           //clear interrupt
     }
```

Notice that in the company's web site for data sheets the output for a given device is identified as analog (voltage or current) or PWM.

Review Questions

1. True or false. In Capture mode, the CCP pin must be configured as an input pin.
2. True or false. We can use only Timers 1 and 3 for Capture mode.
3. True or false. The timer's register values are transferred to CCPR1H:CCPR1L every time the CPU is reset.
4. True or false. After the timer's register values are transferred to CCPR1H:CCPR1L, the timer's registers are cleared.
5. Which register is used to choose the timer for Capture mode?

SECTION 15.4: PWM PROGRAMMING

Another feature of CCP is pulse width modulation (PWM). The PWM feature allows us to create pulses with variable widths. Although we can program timers to create PWM, the CCP module makes the programming of PWM much easier and less tedious. PWM is widely used in industrial controls such as DC motor controls, as we will see in Chapter 17. Indeed the PWM is so widely used that Microchip has enhanced the PWM capabilities of the newer generation of the PIC18 family members and has designated them as ECCP (enhanced CCP). We will study ECCP in the next section. The main difference between ECCP and standard CCP is the PWM capability. In creating pulses with variable widths for the PWM, two factors are important: The period of the pulse and its duty cycle. The duty cycle (DC) is the portion of the pulse that stays HIGH relative to the entire period. Very often the DC is stated in the form of percentages. For example, a pulse with a 4 ms period that stays HIGH for 1 ms has DC of 25% (1 ms / 4 ms = 25%), as shown in Figure 15-13.

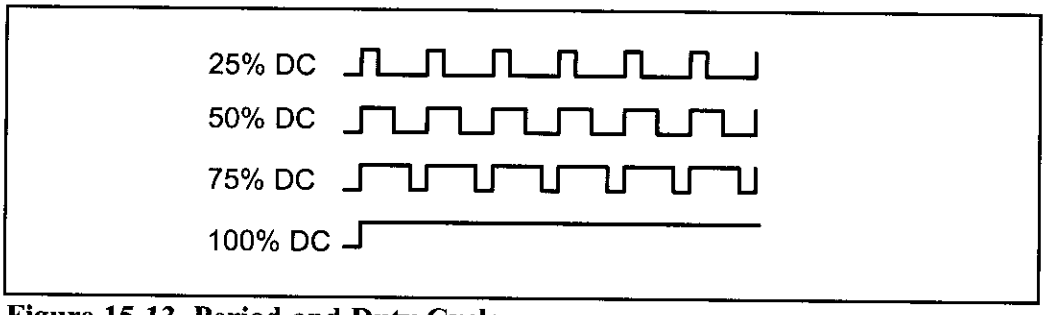


Figure 15-13. Period and Duty Cycle

The period of PWM

The CCP module uses Timer2 and its associated register, PR2, for the PWM time-base, which means that the frequency of the PWM is a fraction of the Fosc, the crystal frequency. It uses the PR2 register to set the PWM period as follows:

$$Tpwm = [(PR2) + 1\ ]\ 4 \times N \times Tosc \qquad \text{(Equ. 15-2)}$$

where Tosc is the inverse of 1/Fosc, the crystal frequency, Tpwm is the desired PWM period, and N is the prescaler of 1, 4, or 16 set by the Timer2 control register (T2CON). Therefore, we can get the value for the PR2 register as follows:

$$PR2 = [Fosc / (Fpwm \times 4 \times N)] - 1 \qquad \text{(Equ. 15-3)}$$

From Equation 15-2, we can conclude that the maximum value for Tpwm can be achieved when N = 16 and PR2 = 255. Therefore, we have:

$$Tpwm = [(255) + 1\ ] \times 4 \times 16 \times Tosc = 16{,}382\ Tosc$$

which means that the minimum allowed Fpwm = Fosc/16,382.

Examine Examples 15-3 to 15-5 to see the calculation of the PWM period.

Example 15-3

Find the PR2 value and the prescaler needed to get the following PWM frequencies. Assume XTAL = 20 MHz.
(a) 1.22 kHz, (b) 4.88 kHz, (c) 78.125 kHz

Solution:

(a) PR2 value = [(20 MHz / (4 × 1.22 kHz)] – 1 = 4,097, which is larger than 255, the maximum value allowed for the PR2. Now choosing the prescaler of 16 we get
PR2 value = [(20 MHz / (4 × 1.22 kHz × 16)] – 1 = 255

(b) PR2 value = [(20 MHz / (4 × 4.88 kHz)] – 1 = 1,023, which is larger than 255, the maximum value allowed for the PR2. Now choosing the prescaler of 4 we get
PR2 value = [(20 MHz / (4 × 4.88 kHz × 4)] – 1 = 255

(c) PR2 value = [(20 MHz / (4 × 78.125 kHz)] – 1 = 63

Example 15-4

Find the PR2 value for the following PWM frequencies. Assume XTAL = 10 MHz and prescaler = 1.
(a) 10 kHz, (b) 25 kHz

Solution:

(b) PR2 value = [(10 MHz / (4 × 10 kHz × 1)] – 1 = 250 – 1 = 249
(c) PR2 value = [(10 MHz / 4 × 25 kHz × 1) – 1 = 100 – 1 = 99

Example 15-5

Find the minimum and maximum Fpwm frequency allowed for XTAL = 10 MHz. State the PR2 and prescaler values for the minimum and maximum Fpwm.

Solution:

We get the minimum Fpwm by making PR2 = 255 and prescaler = 16, which gives us 10 MHz / (4 × 16 × 256) = 610 Hz.

We get the maximum Fpwm by making PR2 = 1 and prescaler = 1, which gives us 10 MHz / (4 × 1 × 1) = 2.5 MHz.

The duty cycle of PWM

As stated earlier, the duty cycle of PWM is the portion of the pulse that stays HIGH relative to the entire period. To set the duty cycle, the CCP module uses the 10-bit register of DC1B9:DC1B0. The 10-bit register of DC1B9:DC1B0 is formed from 8 bits of CCPR1L and 2 bits from the CCP1CON register, where CCPR1L is the upper 8 bits and DC1B2:DC1B1 of the CCP1CON are the lower 2 bits of the 10-bit register. In reality, CCPRL1 is the main register for the duty cycle and the lower 2 bits of DC1B2:DC1B1 are for the decimal point portion of the duty cycle and are set as follows:

DC1B2	DC1B1	Decimal points
0	0	0
0	1	0.25
1	0	0.5
1	1	075

It must be noted that the value for the duty cycle register of the CCPR1L is always some percentage of the PR2 register. For example, if PR2 = 50, and we need a 20% duty cycle, then CCPRL1 = 10 because 20% × 50 = 10. In this case, DC1DB2:DC1B1 = 00. Now assume that we want a 25% duty cycle for the same PR2. Because 50 × 25% = 12.5, we make CCPRL1 = 12 and DC1B2:DC1B1 = 10 to take care of the 0.5 part. See Example 15-6 for further clarification.

Example 15-6

Find the values of registers PR2, CCP1RL, and DC1B2:DC1B1 for the following PWM frequencies if we want a 75% duty cycle. Assume XTAL = 10 MHz.
(a) 1 kHz b) 2.5 kHz

Solution:
(a)
Using the PR2 = Fosc / (4 × Fpwm × N) equation, we must set N = 16 for prescale. Therefore, we have
PR2 = [(10 MHz / (4 × 1 kHz × 16)] – 1 = 156 – 1 = 155 and because 155 × 75% = 116.25 we have CCPR1L = 116 and DC1B2:DC1B1 = 01 for the 0.25 portion.
(b)
Using the PR2 = Fosc / (4 × Fpwm × N) equation, we can set N = 4 for prescale. Therefore, we have
PR2 = [(10 MHz / (4 × 2.5 kHz × 4)] – 1 = 250 – 1 = 249 and because 249 × 75% = 186.75 we have CCPR1L = 186 and DC1B2:DC1B1 = 11 for the 0.75 portion.

TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0

D6 D0

D7 Not used

TOUTPS3:TOUTPS0 D6-D3 Timer 2 Output Postscale Select bits
00 0 0 = 1:1 Postscale value
00 0 1 = 1:2 Postscale value
00 1 0 = 1:3 Postscale value
00 1 1 = 1:4 Postscale value

11 1 0 = 1:15 Postscale value
11 1 1 = 1:16 Postscale value

TMR2ON D2 Timer 2 ON and OFF Control bit
1 = Enable (start) Timer2
0 = Stop Timer2

T2CKPS1:T2CKPS0 D1–D0 Timer2 Clock Prescale Select bits
0 0 = Prescale is 1.
0 1 = Prescale is 4.
1 x = Prescale is 16.

Figure 15-14. T2CON (Timer2 Control) Register

						TMR2IF	TMR1IF

TMR2IF Timer 2 Interrupt overflow Flag bit
0 = TMR2 value is not equal to PR2 register.
1 = TMR2 value is equal to PR2 register.

The location of the TMRxIF in the PIR register can vary in future products.

Figure 15-15. PIR1 (Peripheral Interrupt Flag Register 1) Has the TMR2IF Flag

Steps in programming PWM

The following steps are taken to program the PWM feature of the CCP module:

1. Set the PWM period by writing to the PR2 register.
2. Set the PWM duty cycle by writing to CCPR1L for the higher 8 bits.
3. Set the CCP pin as an output.
4. Using the T2CON register, set the prescale value. See Figure 15-14.
5. Clear the TMR2 register.
6. Configure the CCP1CON register for PWM and set DC1B2:DC1B1 bits for the decimal portion of the duty cycle.
7. Start Timer2.

Examine Programs 15-5 and 15-5C to see how the PWM feature is programmed. These programs use the TMR2IF flag. See Figure 15-15.

Using data from Example 15-6, Program 15-5 will create a 2.5 kHz PWM frequency with a 75% duty cycle on the CCP1 pin.

```
;Program 15-5
     CLRF   CCP1CON          ;clear CCP1CON reg
     MOVLW  D'249'
     MOVWF  PR2
     MOVLW  D'186'           ;75% duty cycle
     MOVWF  CCPR1L
     BCF    TRISC,CCP1       ;make PWM pin an output
     MOVLW  0x01       ;Timer2, 4 prescale, no postscaler
     MOVWF  T2CON
     MOVLW  0x3C             ;PWM mode, 11 for DC1B1:B0
     MOVWF  CCP1CON
     CLRF   TMR2             ;clear Timer2
     BSF    T2CON,TMR2ON     ;turn on Timer2
AGAIN BCF   PIR1,TMR2IF      ;clear Timer2 flag
OVER BTFSS  PIR1,TMR2IF      ;wait for end of period
     BRA    OVER
     GOTO   AGAIN            ;continue

//Program 15-5C is the C version of Program 15-5.
 CCP1CON=0;                  //clear CCP1CON reg
 PR2=249;
 CCPR1L=186;                 //75% duty cycle
 TRISCbits.TRISC2=0;         //make PWM pin an output
 T2CON=0x01;           //Timer2, 4 prescale, no postscaler
 CCP1CON=0x3C;               //PWM mode, 11 for DC1B1:B0
 TMR2=0;                     //clear Timer2
 T2CONbits.TMR2ON=1;         //turn on Timer2
 while(1)
 {
   PIR1bits.TMR2IF=0;        //clear Timer2 flag
   while(PIR1bits.TMR2IF==0); //wait for end of period
 }
```

The role of CCPR1H in the process of creating the duty cycle must be noted. A copy of the duty cycle value in register CCPR1L is given to CCPR1H as soon as we start Timer2. Timer2 goes through the following stages in creating the PWM:

(a) The CCPR1L is loaded into CCPR1H and the CCP1 pin goes HIGH to start the beginning of the period.

(b) As TMR2 counts up, the TMR2 value is compared with both the CCPR1H and PR2 registers.

(c) When the TMR2 and CCPR1H (which is the same as CCPR1L) values are equal, the CCP pin is forced low. That ends the duty cycle portion of the period.

(d) The TMR2 keeps counting up until its value matches the PR2. At that point, the CCP pin goes high, indicating the end of one period and the beginning

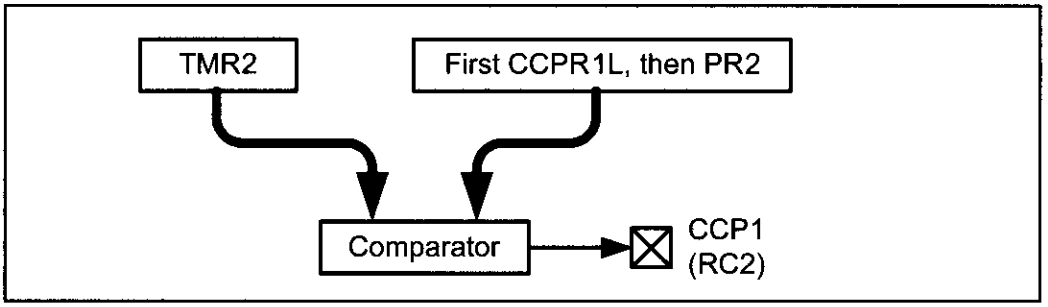


Figure 15-16. TMR2 and PR2 Role in Creating the Duty Cycle

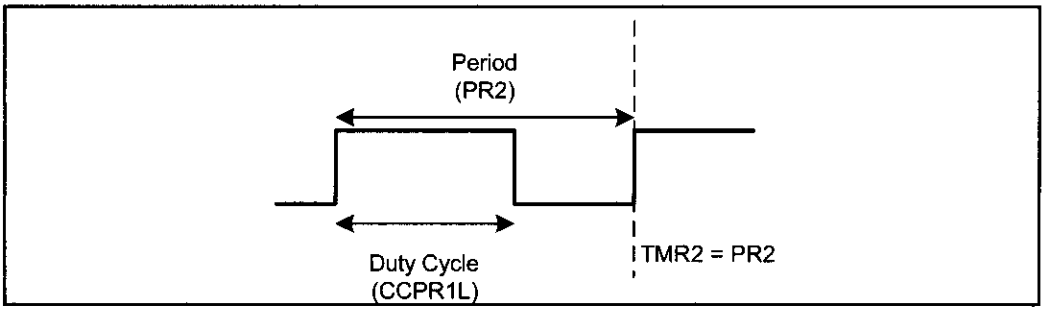


Figure 15-17. TMR2 Relation to CCPR1L and PR2 in PWM

of the next one. It also clears Timer2 for the next round. The CCPR1L is loaded into CCPR1H, and the process continues. See Figures 15-16 and 15-17.

Notice that because the CCPR1L is a fraction of PR2, Timer2 matches CCPR1L first before it matches PR2, unless we have a 100% duty cycle. In that case, Timer2 matches both CCPR1L and PR2 at same time because they have equal values for the 100% duty cycle.

Duty cycle and Fosc

The PIC18 datasheet gives the relation between the Fosc and duty cycle period as follows:

$$Tdutycycle = (DC1B9{:}DC1B0\ value) \times Tosc \times N \qquad \text{(Equ. 15-4)}$$

where Tosc = 1 / Fosc and N is the prescaler of 1, 4, or 16 set by the Timer2 control register. To get the value for the DC1B9:DC1B0 register, we can rearrange the above equation as follows:

$$DC1B9{:}DC1B0 = [(Fosc / (Fdutycycle \times N)] \qquad \text{(Equ. 15-5)}$$

To calculate the the maximum resolution (number of bits) that can be used for the PWM, the PIC manual gives the following equation:

Maximum PWM Resolution (bits) = log(Fosc / Fpwm) / log(2) bits.

Notice that the maximum resolution is 10 bits.

Review Questions

1. True or false. Every standard CCP module has only one PWM pin.
2. How many standard CCP modules do we have in the PIC18F458/4580?
3. True or false. For CCP1, we must use PR2 to set the PWM period.
4. True or false. For CCP1, we must use CCPR1L to set the PWM duty cycle.
5. Which pin of the PIC18F458/4580 is used for PWM?
6. True or false. The duty cycle is always a fraction of the period, unless we want a 100% duty cycle.

SECTION 15.5: ECCP PROGRAMMING

A large number of the PIC18F family members come with ECCP (enhanced CCP) in addition to the standard CCP. While the standard CCP modules are called CCP1, CCP2, and so on, the ECCP modules are designated as ECCP1, ECCP2, and so on. Just like standard CCP, the ECCP has its own pins and registers. The PIC18F452/458 chip uses pin RD4 (PORTD.4) for the ECCP1 pin, while pin RC2 (PORTC.2) is used by the standard CCP1. See Figure 15-18. Figure 15-19 shows the ECCP1 control register.

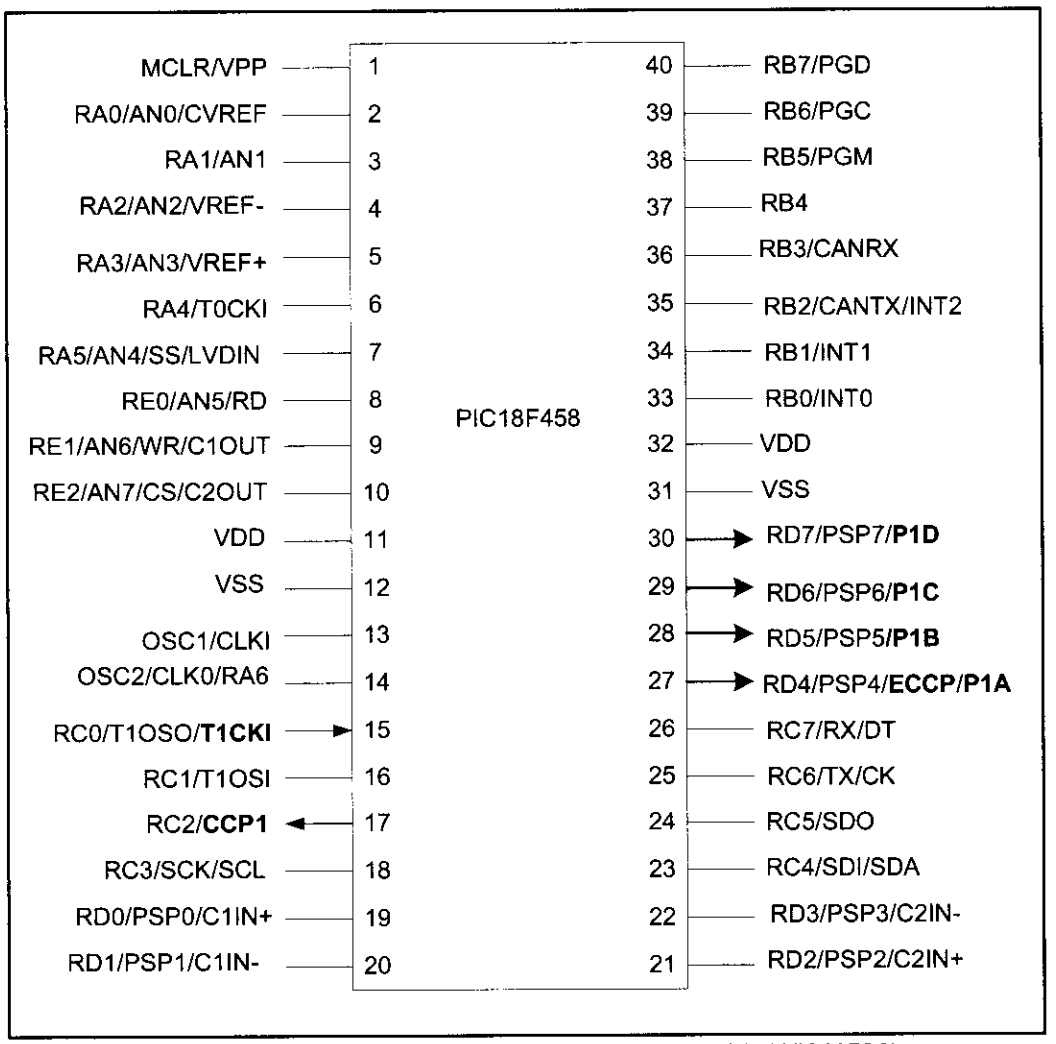


Figure 15-18. ECCP Pins for PWM in PIC18F458/4580 (452/4520)

EPWM1M1	EPWM1M0	EDC1B1	EDC1B0	ECCP1M3	ECCP1M2	ECCP1M1	ECCP1M0
D7							D0

EPWM1M1:EPWM1M0 PWM output pin configuration. It allows the use of a single pin for the capture/compare mode, or four pins for the PWM.
In compare/capture mode, only pin P1A (RD4) is used. In that case, there is no selection for these two bits.
In PWM mode, the options for these two bits are as follows:

00 P1A is used for a modulated output. P1B, P1C, and P1D are used as I/O.
01 Full-Bridge output forward. P1D modulated, P1A active. P1B and P1C inactive.
10 Half-Bridge output. P1A and P1D modulated with deadband control, P1C and P1D used as I/O.
01 Full-Bridge output reverse. P1B modulated, P1C active. P1A and P1D inactive.

EDC1B10:EDC1B1 PWM Duty Cycle least-significant bits. Used in PWM only. The least-significant bits (Bit 1 and Bit 0) of the 10-bit duty cycle register are used in PWM. The ECCPR1L register is used as Bit 2 to Bit 9 of the 10-bit duty cycle register.

ECCP1M3–ECC1M0 ECCP1 Mode Select

0 0 0 0	ECCP1 is off
0 0 0 1	Reserved
0 0 1 0	Compare Mode. Toggle ECCP1 output pin on match. (ECCP1IF bit is set.)
0 0 1 1	Reserved
0 1 0 0	Capture Mode, every falling edge
0 1 0 1	Capture Mode, every rising edge
0 1 1 0	Capture Mode, every 4th rising edge
0 1 1 1	Capture Mode, every 16th rising edge
1 0 0 0	Compare Mode. Initialize ECCP1 pin LOW, on compare match, force CCP1 pin HIGH. (ECCP1IF is set.)
1 0 0 1	Compare Mode. Initialize CCP1 pin HIGH, on compare match, force CCP1 pin LOW. (ECCP1IF is set.)
1 0 1 0	Compare Mode. Generate software interrupt on compare match. (ECCP1IF bit is set, ECCP1 pin is unaffected.)
1 0 1 1	Compare Mode. Trigger special event (ECCP1IF bit is set, and Timer1 or Timer3 is reset to zero.)
1 1 0 0	PWM Mode; P1A, P1C active-HIGH; P1B and P1D active-HIGH
1 1 0 1	PWM Mode; P1A, P1C active-HIGH; P1B and P1D active-LOW
1 1 1 0	PWM Mode; P1A, P1C active-LOW; P1B and P1D active-HIGH
1 1 1 1	PWM Mode; P1A, P1C active-LOW; P1B and P1D active-LOW

Figure 15-19. ECCP1 Control Register. (This register selects one of the operation modes of Capture, Compare, or PWM of EECP1.)

The ECCP1 also has the registers of ECCPR1L, ECCPR1H, and ECCP-CON1. Register PIR2 has the ECCP1IF flag. See Figures 15-20 and 15-21. Just like the standard CCP, it uses Timer1, Timer2, and Timer3 to program the features of compare-capture and PWM. See Table 15-3.

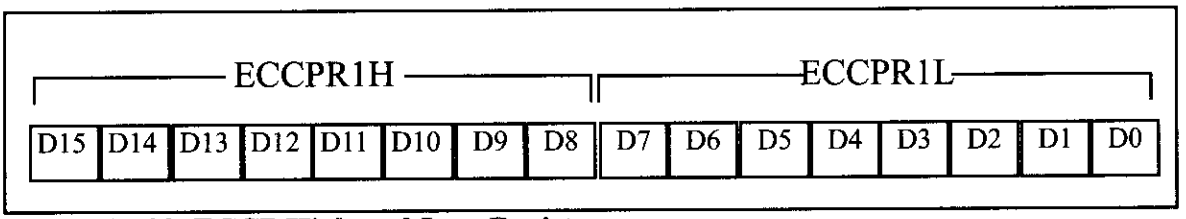


Figure 15-20. ECCP High and Low Registers

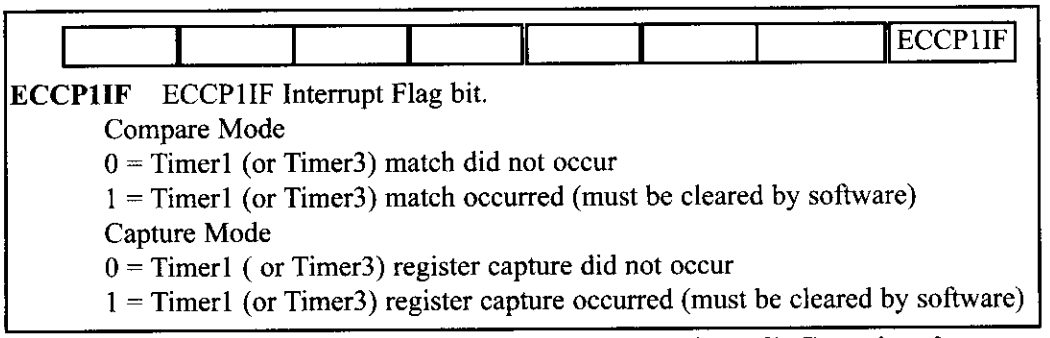


Figure 15-21. PIR2 (Peripheral Interrupt Flag Register 2) Contains the ECCP1IF Flag

Table 15-3: PIC18 Use of Timers for ECCP1

ECCP mode	Timer
Capture	Timer1 or Timer3
Compare	Timer1 or Timer3
PWM	Timer2

Steps for programming the Compare mode in ECCP

Programming the ECCP1 in compare mode is identical to the standard CCP, except we use the ECCP registers. The following steps are taken in programming the Compare mode for ECCP1:

1. Initialize the ECCP1CON register for the compare option.
2. Initialize the T3CON register for Timer1 (or Timer3).
3. Initialize the ECCPR1H:ECCPR1L registers.
4. Make the ECCP1 pin an output pin.
5. Initialize the Timer1 (or Timer3) register values.
6. Start Timer1 (or Timer3).
7. Monitor the ECCP1IF flag (or use an interrupt).

Program 15-6 shows an example of the Compare mode. It uses Timer3 as a counter and counts the number of pulses fed to Timer3. When the count reaches 20, it toggles the LED connected to the ECCP1 pin.

For Program 15-6 assume that a 1-Hz pulse is connected to the Timer3 pin and an LED is connected to the CCP1 pin. Timer3 is being used as a counter. Using the Compare mode, this Assembly language program will toggle the LED every 20 pulses.

```
;Program 15-6
      MOVLW  0x02
      MOVWF  ECCP1CON  ;Compare mode, toggle upon match
      MOVLW  0x42
      MOVWF  T3CON     ;Timer3 for Compare, 1:1 prescaler
      BCF    TRISD,ECCP1     ;ECCP pin as output
      BSF    TRISC,T1CKI     ;T3CLK pin as input pin
      MOVLW  D'20'
      MOVWF  ECCPR1L         ;ECCPR1L = 20
      MOVLW  0x0             ;ECCPR1H = 0
      MOVWF  ECCPR1H
 OVER CLRF   TMR3H           ;clear TMR3H
      CLRF   TMR3L           ;clear TMR3L
      BCF    PIR2,ECCP1IF    ;clear ECCP1IF
      BSF    T3CON,TMR3ON    ;start Timer3
 B1   BTFSS  PIR2,ECCP1IF
      BRA    B1
 ;---------CCP toggle CCP pin upon match
 B2   BCF    T3CON,TMR3ON    ;stop Timer3
      GOTO   OVER            ;keep doing it
```

```
//Program 15-6C is the C version of Program 15-6.
    ECCP1CON=0x02;    //Compare mode, toggle upon match
    T3CON=0x42;       //Timer3 for Compare, 1:1 prescaler
    TRISDbits.TRISD4=0;    //make ECCP1 pin an output
    TRISCbits.TRISC0=1;    //make T3CLK pin an input
    ECCPR1L=20;            //load ECCPR1L
    ECCPR1H=0;             //load ECCPR1H
    while(1)
    {
      TMR3H=0;
      TMR3L=0;
      PIR2bits.ECCP1IF=0;              //clear ECCP1IF flag
      T3CONbits.TMR3ON=1;              //turn on Timer3
      while(PIR2bits.ECCP1IF==0);      //wait for CECP1IF
      //ECCP toggles ECCP pin upon match
      T3CONbits.TMR3ON=0;              //stop Timer3
    }
```

Steps for programming the Capture mode in ECCP

Programming the ECCP1 in capture mode is identical to the standard CCP, except that we use the ECCP registers. The following steps are taken in programming the Capture mode of ECCP1 for measuring the period of a pulse:

1. Initialize the ECCP1CON register for the Capture option.
2. Make the ECCP1 pin an input pin.
3. Initialize the T3CON register to select Timer1 or Timer3.
4. Read the Timer1 (or Timer3) register value on the first rising edge and save it.
5. Read the Timer1 (or Timer3) register value on the second rising edge and save it.
6. Subtract the value in step 4 from the value in step 3.

For Program 15-7 assume that a pulse is being fed to the ECCP1 pin. Using the Capture mode, this Assembly language program measures the period of the pulse and puts it on PORTB and PORTC. The measure is in terms of the Fosc/4 clock period.

```
;Program 15-7
      MOVLW  0x05
      MOVWF  ECCP1CON          ;Capture mode on rising edge
      MOVLW  0x0
      MOVWF  T3CON             ;Timer1 for capture
      MOVLW  0x0
      MOVWF  T1CON      ;Timer1, internal clk, 1:1 prescale
      CLRF   TRISB             ;make PORTB output port
      CLRF   TRISC             ;make PORTC output port
      BSF    TRISD,ECCP1       ;make ECCP1 pin an input
      MOVLW  0x0
      MOVWF  CCPR1H            ;ECCPR1H = 0
      MOVWF  CCPR1L            ;ECCPR1L = 0
OVER  CLRF   TMR1H             ;clear TMR1H
      CLRF   TMR1L             ;clear TMR1L
      BCF    PIR2,ECCP1IF      ;clear ECCP1IF
RE_1  BTFSS  PIR2,ECCP1IF
      BRA    RE_1              ;stay here for 1st rising edge
      BSF    T1CON,TMR1ON      ;start Timer1
      BCF    PIR2,ECCP1IF      ;clear ECCP1IF for next
RE_2  BTFSS  PIR2,ECCP1IF
      BRA    RE_2              ;stay here for 2nd rising edge
      BCF    T1CON,TMR1ON      ;stop Timer1
      MOVFF  TMR1L,PORTC       ;put low byte on PORTC
      MOVFF  TMR1H,PORTD       ;put high byte on PORTD
      GOTO   OVER              ;keep doing it

//Program 15-7C is the C version of Program 15-7.
    ECCP1CON=0x05;    //Capture mode on every rising edge
```

```
   T3CON=0x0;              //Timer1 for capture
   T1CON=0x0;          //Timer1, internal clk, 1:1 prescaler
   TRISC=0;                //make PORTB output port
   TRISD=0;                //make PORTD output port
   TRISDbits.TRISD4=1;     //make ECCP1 pin an input
   ECCPR1L=0;              //ECCPR1L = 0
   ECCPR1H=0;              //ECCPR1H = 0
while(1)
   {
   TMR1H=0;                //clear Timer1
   TMR1L=0;
   PIR2bits.ECCP1IF=0;     //clear ECCP1IF flag
   while(PIR2bits.ECCP1IF==0); //wait for 1st rising edge
   T1CONbits.TMR1ON=1;     //start Timer1
   PIR2bits.ECCP1IF=0;     //clear ECCPIF for next edge
   while(PIR2bits.ECCP1IF==0); //wait for 2nd rising edge
   T1CONbits.TMR1ON=0;     //stop Timer1
   PORTC=CCPR1L;
   PORTD=CCPR1H;           //display the clock count
   }
```

PWM features of ECCP

The main difference between the ECCP and standard CCP module is the PWM capability. The standard CCP allows only a single pin for PWM output. This is not enough for implementation of the H-Bridge used widely in DC motor control. As we will see in Chapter 17, we need four pins to drive the H-Bridge for DC motor control. The ECCP allows the use of four pins for the implementation of Full-Bridge or two pins for the Half-Bridge. The four pins used by the ECCP are shown in Table 15-4. In terms of the duty cycle calculation, ECCP1 is the same as CCP1. It uses the PR2 for the duty cycle.

Table 15-4: PIC18 Usage of Pins for ECCP1

ECCP mode	RD4	RD5	RD6	RD7
Compare/Capture	ECCP1	I/O	I/O	I/O
Dual Output PWM	P1A	P1B	I/O	I/O
Quad Output PWM	P1A	P1B	P1C	P1D

Note: I/O means they are used for input/output purpose or other functions associated with the pins.

Steps in programming PWM of ECCP

The following steps are taken to program the PWM feature of the ECCP module:

1. Set the PWM period by writing to the PR2 register.
2. Set the PWM duty cycle by writing to ECCPR1L for the higher 8 bits.

3. Set the ECCP pins as output.
4. Using the T2CON register, set the prescale value.
5. Clear the TMR2 register.
6. Configure the ECCP1CON register for PWM and set the EDC1B2:EDC1B1 bits for the decimal portion of the duty cycle.
7. Start Timer2.

Notice that in programming the compare/capture features, we can assign Timer1 to standard CCP1 and Timer3 to ECCP1 (or vice versa). For the PWM, however, there is only one register for setting the duty cycle. As a result, if we program the PWM feature for both CCP1 and ECCP1, then they will have the same period because there is only one PR2 to set the period. In Chapter 17 we will show how to use ECCP for DC motor control using all four pins in H-Bridge implementations.

Review Questions

1. True or false. Every ECCP module can use only one pin for PWM.
2. How many ECCP modules does the PIC18F458/4580 have?
3. True or false. For ECCP1, we must use PR2 to set the PWM period.
4. True or false. For ECCP1, we must use CCPR1L to set the PWM duty cycle.
5. Which pins of the PIC18F458/4580 are used for PWM?

SUMMARY

This chapter began by describing the CCP features of the PIC18 family. We discussed both the standard CCP and enhanced CCP (ECCP) modules and described each of the compare, capture, and PWM features. We showed how to use Timer1 or Timer3 as the time basis for the compare and capture modes. We also showed how PWM uses Timer2 to create the pulse width modulation.

PROBLEMS

SECTION 15.1: STANDARD AND ENHANCED CCP MODULES

1. True or false. Every member of the PIC18 family has an on-chip CCP module.
2. True or false. The PIC18F452/458 has only one standard CCP.
3. True or false. The PIC18F452/458 has only one ECCP module.
4. True or false. Each CCP module has a 16-bit register accessible as CCPRL and CCPRH.
5. True or false. Each CCP module has a single pin.
6. Give the number of standard and enhanced CCP (ECCP) modules in the PIC18F4520/4580.
7. Give the pin used for standard CCP in the 40-pin DIP package of the PIC18F458/4580.

SECTION 15.2: COMPARE MODE PROGRAMMING

8. True or false. We use register CCP1CON to choose the Compare mode.
9. True or false. We can use Timer0 and Timer2 for Compare mode.
10. True or false. To use Compare mode, we must make the CCP pin an output pin.
11. Which timers can be used for the Compare mode?
12. Assuming that we are using Timer1 for the Compare mode, indicate when the CCP pin is driven HIGH.
13. Which register holds the CCP flag bit?
14. Find the value for the CCP1CON register in compare mode if we want to drive HIGH the CCP pin upon match.
15. Find the value for the CCP1CON register in compare mode if we want to drive LOW the CCP pin upon match.
16. Find the value for the CCP1CON register in compare mode if we want to toggle the CCP pin upon match.
17. Rewrite Program 15-1 (or 15-1C) for Timer1.
18. Rewrite Program 15-1 (or 15-1C) for the count of 1000.
19. Rewrite Program 15-2 (or 15-2C) for Timer3.
20. Rewrite Program 15-2 (or 15-2C) to create a square wave with a frequency of 100 Hz.

SECTION 15.3: CAPTURE MODE PROGRAMMING

21. True or false. We use the CCP1CON register to choose the Capture mode.
22. True or false. We can use Timer0 and Timer2 for Capture mode.
23. True or false. To use Capture mode, we must make the CCP pin an output pin.
24. Which timers can be used for the capture mode?
25. Find the value for the CCP1CON register in capture mode if we want to capture on the falling edge.
26. Find the value for the CCP1CON register in capture mode if we want to capture every fourth rising edge.
27. Find the value for the T3CON register if we want to use Timer1 for capture mode.
28. Rewrite Program 15-3 (or 15-3C) for Timer3.

SECTION 15.4: PWM PROGRAMMING

29. True or false. We use the CCP1CON register to choose the PWM mode.
30. True or false. We can use Timer0 and Timer1 for the PWM mode.
31. True or false. To use PWM mode, we must make the CCP pin an output pin.
32. Which timer can be used for PWM mode for the standard CCP1?
33. Find the value for the CCP1CON register for PWM mode.
34. Of the CCPR1L and CCPR1H registers, which one is used to set the duty cycle?
35. Which register holds the DC1B2:DC1B1 bits?
36. What is the role of the DC1B2:DC1B1 bits in creating the duty cycle?
37. What is the value for the DC1B2:DC1B1 bits if we want 0.75 for the decimal

points part of the duty cycle?

38. In programming the PWM, the value loaded into the CCPRL1 is always a _____ (fraction, multiple) of the PR2 value.
39. Find the values of registers PR2, CCP1RL, and DCB1B2:DC1B1 bits for the PWM frequency of 2 kHz with 25% duty cycle. Assume XTAL = 10 MHz.
40. Find the values of registers PR2, CCP1RL, and DCB1B2:DC1B1 bits for the PWM frequency of 1.8 kHz with duty cycle of 25%. Assume XTAL = 10 MHz.
41. Find the values of registers PR2, CCP1RL, and DCB1B2:DC1B1 bits or the PWM frequency of 1.5 kHz with duty cycle of 25%. Assume XTAL = 10 MHz.
42. Find the values of registers PR2, CCP1RL, and DCB1B2:DC1B1 bits for the PWM frequency of 1.2 kHz with duty cycle of 25%. Assume XTAL = 10 MHz.

SECTION 15.5: ECCP PROGRAMMING

43. True or false. We use ECCP1CON to choose the PWM mode.
44. True or false. We can use Timer1 or Timer3 for the PWM mode in ECCP.
45. True or false. To use capture mode, we must make the ECCP pin an output pin.
46. Which timer can be used for the PWM mode for ECCP1?
47. Which register holds the ECCP1IF flag bit?
48. Find the value for the ECCP1CON register in compare mode if we want to drive HIGH the ECCP pin upon match.
49. In the PIC18F452/458, give the pin used for ECCP for compare/capture mode.
50. Which timers can be used for the compare mode in ECCP?
51. Which pins are used for PWM in ECCP1?
52. Find the value for the ECCP1CON register in compare mode if we want to drive HIGH the ECCP pin upon match.
53. Find the value for the ECCP1CON register in PWM mode if we want to have H-Bridge where P1A and P1C are active high and the rest are active low.
54. Of the ECCPR1L and ECCPR1H registers, which one is used to set the duty cycle?
55. Which register holds the EDC1B2:EDC1B1 bits?
56. What is role of the EDC1B2:EDC1B1 bits in creating duty cycle?
57. What is value for the EDC1B2:EDC1B1 bits if we want 0.5 for the decimal points part of the duty cycle?
58. In programming the PWM, the value loaded into ECCPRL1 is always a _____ (fraction, multiple) of the PR2 value.
59. Find the values of registers PR2, ECCP1RL, and EDCB1B2:EDC1B1 bits for the PWM frequency of 2 kHz with 25% duty cycle. Assume XTAL = 10 MHz.
60. Find the values of registers PR2, ECCP1RL, and EDCB1B2:DC1B1 bits for the PWM frequency of 1.8 kHz with duty cycle of 25%. Assume XTAL = 10 MHz.
61. Find the values of registers PR2, ECCP1RL, and EDCB1B2:DC1B1 bits for the PWM frequency of 1.5 kHz with duty cycle of 25%. Assume XTAL = 10 MHz.
62. Find the values of registers PR2, ECCP1RL, and EDCB1B2:DC1B1 bits for the PWM frequency of 1.2 kHz with duty cycle of 25%. Assume XTAL = 10 MHz.

ANSWERS TO REVIEW QUESTIONS

SECTION 15.1: STANDARD AND ENHANCED CCP MODULES

1. True
2. True
3. True
4. RC2 (PORTC.2)

SECTION 15.2: COMPARE MODE PROGRAMMING

1. False
2. True
3. True
4. T3CON

SECTION 15.3: CAPTURE MODE PROGRAMMING

1. True
2. True
3. False
4. False
5. T3CON

SECTION 15.4: PWM PROGRAMMING

1. True
2. One
3. True
4. True
5. RC2
6. True

SECTION 15.5: ECCP PROGRAMMING

1. False. Up to four pins.
2. One
3. True
4. False
5. RD4–RD7

CHAPTER 16

SPI PROTOCOL AND DS1306 RTC INTERFACING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Understand the Serial Peripheral Interfacing (SPI) protocol**
- **>> Explain how the SPI read and write operations work**
- **>> Examine the SPI pins SDO, SDI, CE, and SCLK**
- **>> Code programs in Assembly and C for SPI**
- **>> Explain how the real-time clock (RTC) chip works**
- **>> Explain the function of the DS1306 RTC pins**
- **>> Explain the function of the DS1306 RTC registers**
- **>> Understand the interfacing of the DS1306 RTC to the PIC18**
- **>> Code programs to display time and date in Assembly and C**
- **>> Explore and program the alarm and interrupt features of the RTC**

This chapter discusses the SPI bus and shows the interfacing and programming of the DS1306 real-time clock (RTC), an SPI chip. In Section 16.1, we describe SPI bus connection and protocol. In Section 16.2, we describe the DS1306 RTC's pin functions and show its interfacing and programming with the PIC18. The C programming of DS1306 is shown in Section 16.3. The alarm feature of the DS1306 is discussed in Section 16.4.

SECTION 16.1: SPI BUS PROTOCOL

The SPI (serial peripheral interface) is a bus interface connection incorporated into many devices such as ADC, DAC, and EEPROM. In this section we examine the pins of the SPI bus and show how the read and write operations in the SPI work.

SPI bus

The SPI bus was originally started by Motorola Corp. (now Freescale), but in recent years has become a widely used standard adapted by many semiconductor chip companies. SPI devices use only 2 pins for data transfer, called SDI (Din) and SDO (Dout), instead of the 8 or more pins used in traditional buses. This reduction of data pins reduces the package size and power consumption drastically, making them ideal for many applications in which space is a major concern. The SPI bus has the SCLK (shift clock) pin to synchronize the data transfer between two chips. The last pin of the SPI bus is CE (chip enable), which is used to initiate and terminate the data transfer. These four pins, SDI, SDO, SCLK, and CE, make the SPI a 4-wire interface. See Figure 16-1. There is also a widely used standard called a 3-wire interface bus. In a 3-wire interface bus, we have SCLK and CE, and only a single pin for data transfer. The SPI 4-wire bus can become a 3-wire interface when the SDI and SDO data pins are tied together. However, there are some major differences between the SPI and 3-wire devices in the data transfer protocol. For that reason, a device must support the 3-wire protocol internally in order to be used as a 3-wire device. Many devices such as the DS1306 RTC (real-time clock) support both SPI and 3-wire protocols.

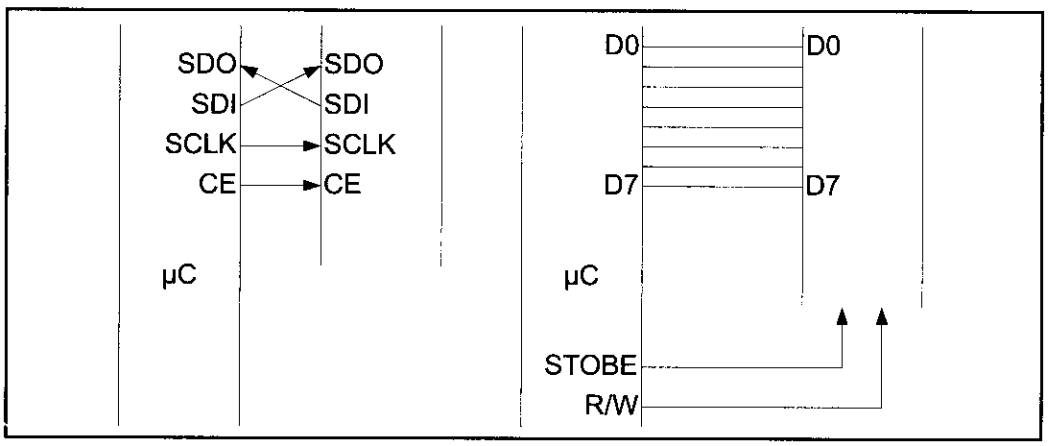


Figure 16-1. SPI Bus vs. Traditional Parallel Bus Connection to Microcontroller

SPI read and write protocol

In connecting a device with an SPI bus to a microcontroller, we use the microcontroller as the master while the SPI device acts as a slave. This means that the microcontroller generates the SCLK, which is fed to the SCLK pin of the SPI device. The SPI protocol uses SCLK to synchronize the transfer of information one bit at a time, where the most-significant bit (MSB) goes in first. During the transfer, the CE must stay HIGH. The information (address and data) is transferred between the microcontroller and the SPI device in groups of 8 bits, where the address byte is followed immediately by the data byte. To distinguish between the read and write, the D7 bit of the address byte is always 1 for write, while for the read, the D7 bit is LOW, as we will see next.

Steps for writing data to an SPI device

In accessing SPI devices, we have two modes of operation: single-byte and multibyte. We will explain each one separately.

Single-byte write

The following steps are used to send (write) data in single-byte mode for SPI devices, as shown in Figure 16-2:

1. Make CE = 1 to begin writing.
2. The 8-bit address is shifted in one bit at a time, with each edge of SCLK. Notice that A7 = 1 for the write operation, and the A7 bit goes in first.
3. After all 8 bits of the address are sent in, the SPI device expects to receive the data belonging to that address location immediately.
4. The 8-bit data is shifted in one bit at a time, with each edge of the SCLK.
5. Make CE = 0 to indicate the end of the write cycle.

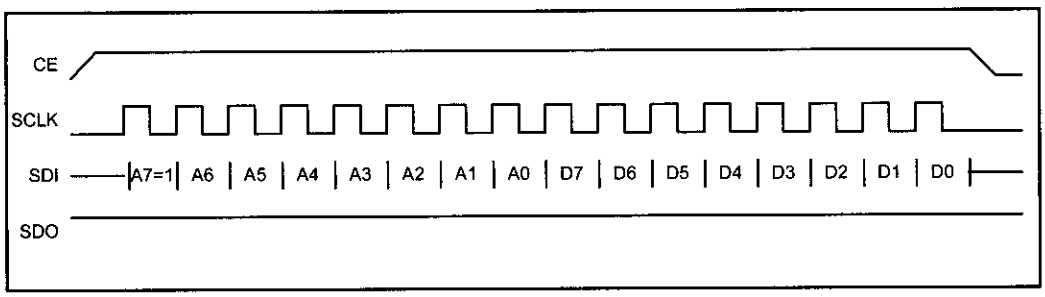


Figure 16-2. SPI Single-Byte Write Timing (Notice A7 = 1)

Multibyte burst write

Burst mode writing is an effective means of loading consecutive locations. In burst mode, we provide the address of the first location, followed by the data for that location. From then on, while CE = 1, consecutive bytes are written to consecutive memory locations. In this mode, the SPI device internally increments the

address location as long as CE is HIGH. The following steps are used to send (write) multiple bytes of data in burst mode for SPI devices as shown in Figure 16-3:

1. Make CE = 1 to begin writing.
2. The 8-bit address of the first location is provided and shifted in one bit at a time, with each edge of SCLK. Notice that A7 = 1 for the write operation and the A7 bit goes in first.
3. The 8-bit data for the first location is provided and shifted in one bit at a time, with each edge of the SCLK. From then on, we simply provide consecutive bytes of data to be placed in consecutive memory locations. In the process, CE must stay high to indicate that this is a burst mode multibyte write operation.
4. Make CE = 0 to end writing.

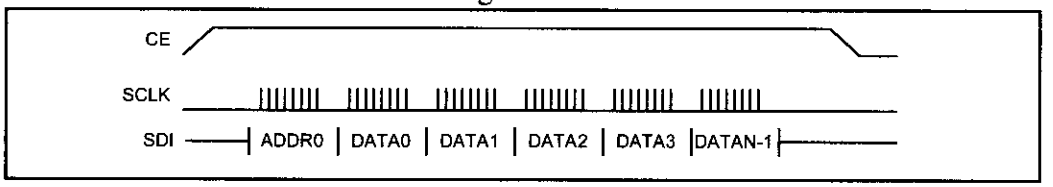


Figure 16-3. SPI Burst (MultiByte) Mode Writing

Steps for reading data from an SPI device

In reading SPI devices, we also have two modes of operation: single-byte and multibyte. We will explain each one separately.

Single-byte read

The following steps are used to get (read) data in single-byte mode from SPI devices as shown in Figure 16-4:

1. Make CE = 1 to begin writing.
2. The 8-bit address is shifted in one bit at a time, with each edge of SCLK. Notice that A7 = 0 for the read operation, and the A7 bit goes in first.
3. After all 8 bits of the address are sent in, the SPI device sends out data belonging to that location.
4. The 8-bit data is shifted out one bit at a time, with each edge of the SCLK.
5. Make CE = 0 to indicate the end of the read cycle.

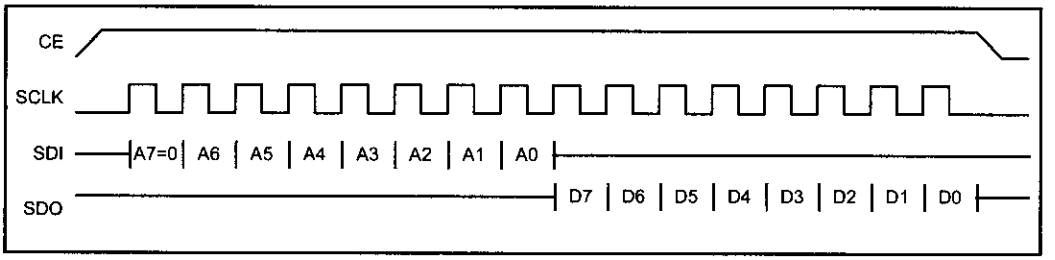


Figure 16-4. SPI Single-Byte Read Timing (Notice A7 = 0)

Multibyte burst read

Burst mode reading is an effective means of bringing out the contents of consecutive locations. In burst mode, we provide the address of the first location only. From then on, while CE = 1, consecutive bytes are brought out from consecutive memory locations. In this mode, the SPI device internally increments the address location as long as CE is HIGH. The following steps are used to get (read) multiple bytes of data in burst mode for SPI devices, as shown in Figure 16-5:

1. Make CE = 1 to begin reading.
2. The 8-bit address of the first location is provided and shifted in one bit at a time, with each edge of SCLK. Notice that A7 = 0 for the read operation, and the A7 bit goes in first.
3. The 8-bit data for the first location is shifted out one bit at a time, with each edge of the SCLK. From then on, we simply keep getting consecutive bytes of data belonging to consecutive memory locations. In the process, CE must stay HIGH to indicate that this is a burst mode multibyte read operation.
4. Make CE = 0 to end reading.

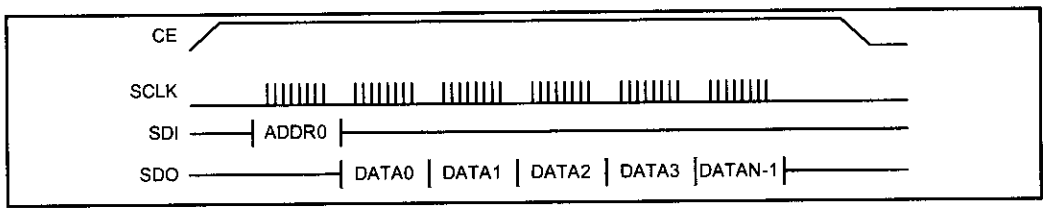


Figure 16-5. SPI Burst (MultiByte) Mode Reading

Review Questions

1. True or false. The SPI protocol writes and reads information in 8-bit chunks.
2. True or false. In SPI, the address is immediately followed by the data.
3. True or false. In an SPI write cycle, bit A7 of the address is LOW.
4. True or false. In an SPI write, the LSB goes in first.
5. State the difference between the single-byte and burst modes in terms of the CE signal.

SECTION 16.2: DS1306 RTC INTERFACING AND PROGRAMMING

The real-time clock (RTC) is a widely used device that provides accurate time and date information for many applications. Many systems such as the x86 IBM PC come with such a chip on the motherboard. The RTC chip in the IBM PC provides the time components of hour, minute, and second, in addition to the date/calendar components of year, month, and day. Many RTC chips use an internal battery, which keeps the time and date even when the power is off. Although some microcontrollers, such as the DS5000T, come with the RTC already embedded into the chip, we have to interface the vast majority of them to an external RTC chip. One of the most widely used RTC chips is the DS12887 from Dallas Semiconductor/Maxim Corp. This chip is found in the vast majority of x86 PCs. The original IBM PC/AT used the MC14618B RTC from Motorola. The DS12887 is the replacement for that chip. It uses an internal lithium battery to keep operating for over 10 years in the absence of external power. The DS12887 is a parallel RTC with 8 pins for the data bus. In this chapter, we interface and program the DS1306 RTC, which has an SPI bus. According to the DS1306 data sheet from Maxim, it keeps track of "seconds, minutes, hours, day of week, date, month, and year with leap-year compensation valid up to year 2099." The DS1306 RTC provides the above information in BCD format only. It supports both 12-hour and 24-hour clock modes, with AM and PM in the 12-hour mode. It does not support the Daylight Savings Time option. The DS1306 has a total of 128 bytes of nonvolatile RAM. It uses 28 bytes of RAM for clock/calendar and control registers, and the other 96 bytes of RAM are for general-purpose data storage. Next, we describe the pins of the DS1306. See Figure 16-6.

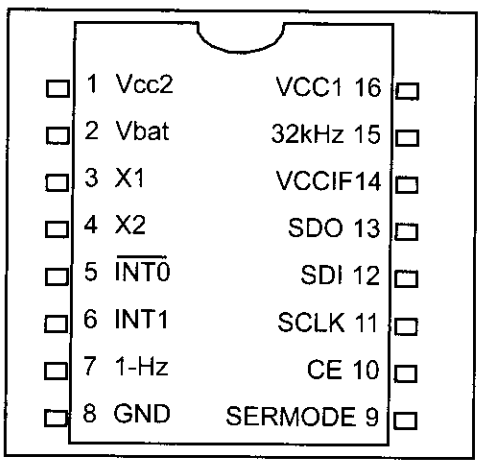


Figure 16-6. DS1306 RTC Chip (from Maxim/Dallas Semiconductor)

V_{CC2}

Pin 1 provides an external back-up supply voltage to the chip. This pin is connected to an external rechargeable power source. This option is called *trickle charge*. If this pin is not used, it must be grounded.

V_{bat}

Pin 2 can be connected to an external +3 V lithium battery, thereby providing the power source to the chip externally as back-up supply voltage. We must connect this pin to ground if it is not used.

V_{CC1}

Pin 16 is used as the primary external voltage supply to the chip. This primary external voltage source is generally set to +5 V. When V_{cc1} falls below the Vbat voltage level, the DS1306 switches to Vbat and the external lithium battery provides power to the RTC. According to the DS1306 data sheet "upon power-up, the device switches from Vbat to Vcc1 when Vcc1 is greater than Vbat+0.2 Volts." Because we can connect the standard 3 V lithium battery to the Vbat pin, the Vcc1 voltage level must remain above 3.2 V in order for the Vcc1 to remain as the primary voltage source to the chip. This nonvolatile capability of the RTC prevents

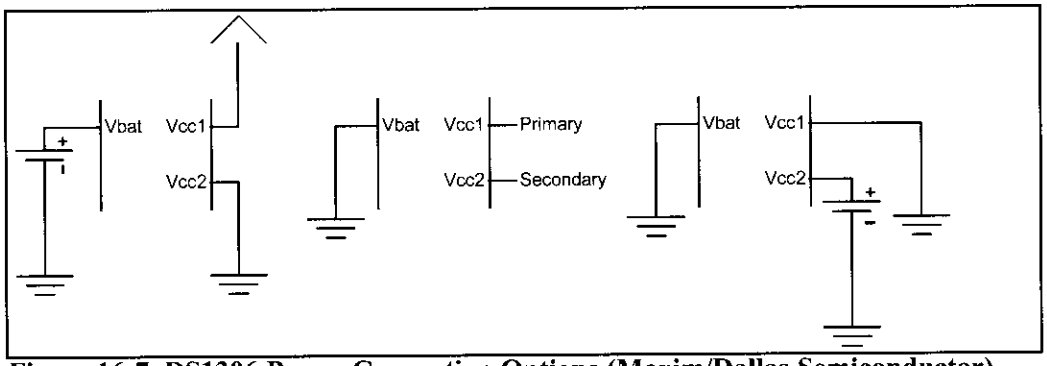


Figure 16-7. DS1306 Power Connection Options (Maxim/Dallas Semiconductor)

any loss of data. See Figure 16-7.

GND

Pin 8 is the ground.

SDI (Serial Din)

The SDI pin provides the path to bring data into the chip, one bit at a time.

SDO (Serial Dout)

The SDO pin provides the path to bring data out of the chip, one bit at a time.

32KHz

This is an output pin providing a 32.768 kHz frequency. This frequency is always present at the pin.

X1–X2

These are input pins that allow the DS1306 connection to an external crystal oscillator to provide the clock source to the chip. We must use the standard 32.768 kHz quartz crystal. The accuracy of the clock depends on the quality of this crystal oscillator. See Figure 16-8. Heat can cause a drift on the oscillator. To avoid this, we use the DS32KHZ chip, which automatically adjusts for temperature variations. Note that when using the DS32KHZ or similar clock generators, we only need to connect X1 because the X2 loopback is not required.

SCLK (serial clock)

An input pin is used for the serial clock to synchronize the data transfer between the DS1306 and the microcontroller.

1-Hz

An output pin provides a 1-Hz square wave frequency. The DS1306 creates the 1-Hz square wave automatically. To get this 1-Hz frequency to show up on the pin, however, we must enable the associated bit in the DS1306 control register.

CE

Chip enable is an input pin and an active-HIGH signal. During the read and write cycle time, CE must be high.

INT0#

Interrupt request is an output pin and an active-LOW signal. To use INT0, the interrupt-enable bit in the RTC control register must be set HIGH. The interrupt feature of the DS1306 is discussed in Section 16.4.

INT1

Interrupt request is an output pin and an active-HIGH signal. To use INT1, the interrupt-enable bit in the RTC control register must be set HIGH. The inter-

rupt feature of the DS1306 is discussed in Section 16.4.

SERMODE (serial mode selection)

Pin 9 is an input pin. If it is HIGH, then the SPI mode is selected. If it is connected to ground, the 3-wire mode is used. In our application, the SERMODE pin is connected to the V_{cc} pin because we program the 1306 chip using the SPI protocol.

V_{CCif}

Pin 14 is the interface logic power-supply input. This pin allows interfacing of the DS1306 with systems with 3 V logic in mixed supply systems. See the DS1306 data sheet if you are using a power source other than 5 V in your system.

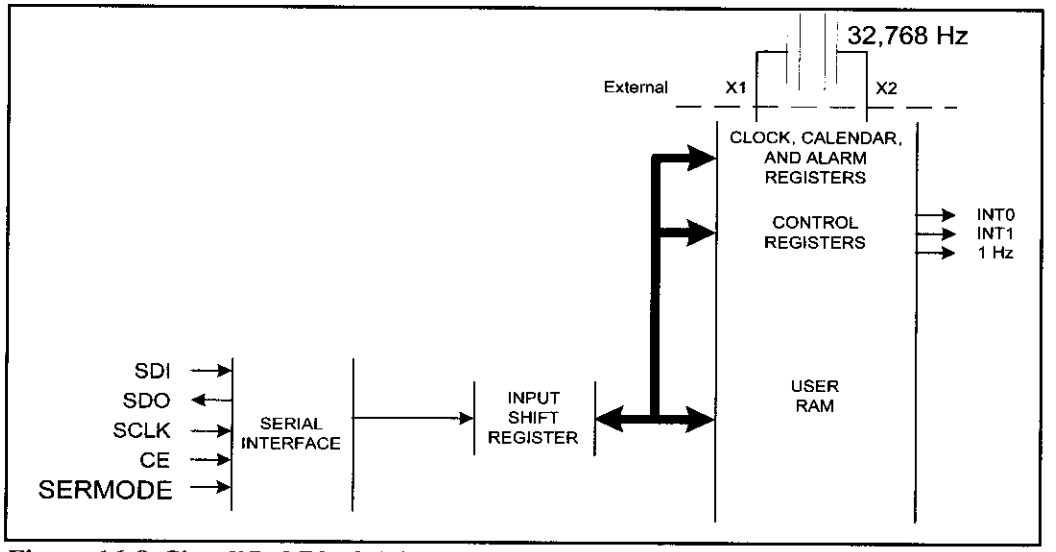


Figure 16-8. Simplified Block Diagram of DS1306 (Maxim/Dallas Semiconductor)

Importance of the WP bit in the Control register

As shown in Table 16-1, the Control register has an address of 8FH for write and 0FH for read. The most important bit in the Control register is the WP bit. The WP bit is undefined upon reset. In order to write to any of the registers of the DS1306, we must clear the WP bit first. See Figure 16-9. Upon powering up the DS1306, we have to clear the WP bit at least once. This means that after initializing the DS1306 we can write protect all the registers by making WP = 1.

WP (Write Protect) If the WP bit is set high, the DS1306 prevents any write operation to its registers. Upon power-up, the WP bit is undefined. Therefore, we must make WP = 0 before we can write to any of the registers. This must be done once upon power-up of the DS1306.

The other bits of the Control register are explained in the next section.

Figure 16-9. WP Bit of DS1306 Control Register (write location address is 8FH)

Address map of the DS1306

The DS1306 has a total of 128 bytes of RAM space with addresses 00–7FH. The first fifteen locations, 00–0E, are set aside for RTC values of time, date, and alarm data. The next three bytes are used for the control and status registers. They are located at addresses 0F–11 in hex. The next 14 bytes from addresses 12H to 1FH are reserved and cannot be used. That leaves 96 bytes, from addresses 20H to 7FH, available for general purpose data storage. That means the entire 128 bytes of RAM are accessible directly for read or write except the addresses 12–1FH. Table 16-1 shows the address map of the DS1306. In this section we study the time and date. The alarm is examined in Section 16.4.

Table 16-1: Registers of the DS1306 (modified from datasheet)

HEX ADDRESS		D7	D6	D5	D4	D3	D2	D1	D0	RANGE
READ	WRITE									in HEX
0x00	0x80	0	10 SEC				SEC			00-59
0x01	0x81	0	10 MIN				MIN			00-59
0x02	0x82	0	24 / 12	20 HR P/A	10 HR		HOURS			00-23 01-12 P/A
0x03	0x83	0	0	0	0	0	DAY			01-07
0x04	0x84	0	0	10 DATE			DATE			01-31
0x05	0x85	0	0	10 MONTH			MONTH			01-12
0x06	0x86	0	10 YEAR				YEAR			00-99
0x07	0x87	M	10 SEC ALARM0				SEC ALARM0			00-59
0x08	0x88	M	10 MIN ALARM0				MIN ALARM0			00-59
0x09	0x89	M	24 / 12	20 HR P/A	10 HR		HOUR ALARM0			00-23 01-12 P/A
0x0A	0x08A	M	0	0	0	0	DAY ALARM0			01-07
0x0B	0x8B	M	10 SEC ALARM1				SEC ALARM1			00-59
0x0C	0x8C	M	10 MIN ALARM1				MIN ALARM1			00-59
0x0D	0x8D	M	24 / 12	20 HR P/A	10 HR		HOUR ALARM1			00-23 01-12 P/A
0x0E	0x8E	M	0	0	0	0	DAY ALARM1			01-07
0x0F	0x8F	CONTROL REGISTER								
0x10	0x90	STATUS REGISTER								
0x11	0x91	TRICKLE CHARGER REGISTER								
0x12-0x1F	0x92-0x9F	RESERVED								

Time and date address locations and modes

The byte addresses 0–6 are set aside for the time and date, as shown in Table 16-2. Table 16-2 is extracted from Table 16-1. It shows a summary of the address locations in read/write modes with data ranges for each location. The DS1306 provides data in BCD format only. Notice the data range for the hour mode. We can select 12-hour or 24-hour mode with bit 6 of hour location 02. When D6 = 1, the 12-hour mode is selected, and D6 = 0 provides us the 24-hour mode. In the 12-hour mode, we decide the AM and PM with the bit 5. If D5 = 0, the AM is selected and D5 = 1 is for the PM. Example 16-1 shows how to get the range of the data acceptable for the hour location.

Table 16-2: DS1306 Address Locations for Time and Date (extracted from Table 16-1)

Hex Address Location		Function	Data Range	Range in hex
Read	Write		BCD	
00	80	Seconds	00–59	00–59
01	81	Minutes	00–59	00–59
02	82	Hours, 12-Hour Mode	01–12	41–52 AM
		Hours, 12-Hour Mode	01–12	61–72 PM
		Hours, 24-Hour Mode	00–23	00–23
03	83	Day of the Week, Sun = 1	01–07	01–07
04	84	Day of the Month	01–31	01–31
05	85	Month	01–12	01–12
06	86	Year	00–99	00–99

Example 16-1

Using Table 16-1, verify the hour location values in Table 16-2.

Solution:

(a) For 24-hour mode, we have D6 = 0. Therefore, the range goes from 0000 **0000** to 0010 **0011,** which is 00–23 in BCD.
(b) For 12-hour mode, we have D6 = 1 and D5 = 0 for AM. Therefore, the range goes from 0100 **0001** to 0101 **0010,** which is 41–52 in BCD.
(c) For 12-hour mode, we have D6 = 1 and D5 = 1 for PMn. Therefore, the range goes from 0110 **0001** to 0111 **0010,** which is 61–72 in BCD.

PIC18 interfacing to DS1306 using MSSP module

The DS1306 supports both SPI and 3-wire modes. In DS1306, we select the SPI mode by connecting the SERMODE pin to Vcc. If SERMODE = Gnd, then the 3-wire protocol is used. In this section, we use SPI mode only. The MSSP (Master Synchronous Serial Port) module inside the PIC18 supports SPI bus protocol. Three registers are associated with SPI of the MSSP module. They are SSPBUF, SSPCON1, and SSPSTAT. To transfer a byte of data, we place it in SSPBUF. The SSPBUF register also holds the byte received via the SPI bus. Figures 16-10 and 16-11 show the other two major registers of the PIC18 for SPI interfacing. We use SSPCON1 to select the SPI mode operation of the PIC18. Notice that the SSPEN bit in the SSPCON1 register must be set to HIGH to allow the use of the PIC18 pins for SPI data bus protocol. We must also choose the SPI Master mode using the SSPM3:SSPM0 bits of SSPCON1. In our application, we will use Fosc/64 speed for best performance in data transfer between the PIC18 and the DS1306 RTC.

After the selection of SSPCON1, we must also select the proper bits for timing in the SSPSTAT register, as shown in Figure 16-11. In our application, we send data to an SPI device on the rising edge, and receive data from the SPI device in the middle of the SCLK clock pulse.

Because we are using the SPI feature of the PIC18 to communicate with

our SPI device, we must use the designated pins for the SPI signals. They are RC2 (CE), RC3 (SCLK), RC4 (SDI), and RC5 (SDO), as shown in Figure 16-12.

		SSPEN		SSPM3	SSPM2	SSPM1	SSPM0

SSPEN D5 Synchronous Serial Port Enable bit
1 = Enables serial port and configures SCK, SDO, and SDI as serial port pins
0 = Disables serial port and configures these pins as I/O ports

SSPM3:SSPM0 D3–D0 SPI Mode Selection bits
0010 = SPI Master, clock = Fosc/64
0001 = SPI Master, clock = Fosc/16
0000 = SPI Master, clock = Fosc/4

The rest of the bits are unused in our implementation of SPI.
We use SPI in master mode.

Figure 16-10. SSPCON1 - SSP Control Register 1
Note: Portion shown is used for SPI.

SMP	CKE						BF

SMP D7 Sample bit
1 = Input data sampled at end of data output time
0 = Input data sampled at middle of data output time

CKE D6 SPI Clock Edge Select bit
1 = Transmit occurs on transition from active to idle clock state.
0 = Transmit occurs on transition from idle to active clock state.

BF D0 Buffer Full Status bit. Used for receive only.
1 = Receive complete, SSPBUF is full.
0 = Receive not complete, SSPBUF is empty.

The rest of the bits are used for I^2C module.

Figure 16-11. SSPSTAT - SSP Status Register
Note: Portion shown is used for SPI.

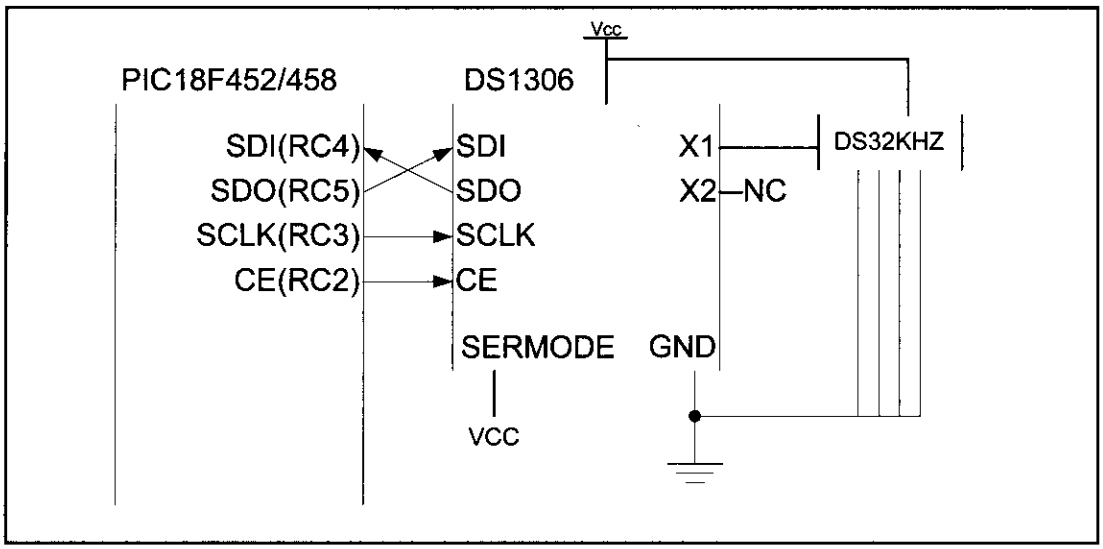


Figure 16-12. DS1306 Connection to PIC18

Note: For more accuracy, we use the DS32KHZ chip in place of a crystal.

Setting the time in Assembly

Program 16-1 initializes the clock at 16:58:55 using the 24-hour clock mode. It uses the single-byte operation for writing into the control register of the DS1306 and multibyte burst mode for writing seconds, minutes, and hours. Regarding the SPI subroutine in Program 16-1, we must note the following points:

1. In order for the PIC18 to transfer a byte of data using SPI protocol, it must be placed in SSPBUF.
2. After writing to SSPBUF, we must monitor the BF flag bit of the SSPSTAT register to ensure the entire byte has been transferred.
3. SSPBUF is also used as the destination for incoming data from an SPI device. This happens as data is being sent. The BF flag indicates that the entire byte has been received.

```
;Program 16-1: Setting the Time
      MOVLW 0x00
      MOVWF SSPSTAT ;read at middle, send on active edge
      MOVLW 0x22
      MOVWF SSPCON1 ;enable master SPI, Fosc / 64
      CLRF TRISC            ;make PORTC output
      BSF TRISC,SDI         ;except SDI
 ;-- send control byte to DS1306 in single-byte mode
      BSF   PORTC,RC2       ;make CE = 1 for single-byte
      CALL SDELAY
      MOVLW 0x8F            ;DS1306 control register address
      CALL SPI
      MOVLW 0x00            ;clear WP bit for write
```

```
     CALL SPI
     BCF PORTC,RC2
;make CE = 0 to end write (single-byte)
     CALL SDELAY
;-- send the data to DS1306 in burst mode
     BSF  PORTC,RC2  ;make CE = 1 (start multibyte write)
     MOVLW 0x80            ;seconds register address
     CALL SPI              ;send address
     MOVLW 0x55            ;55 seconds
     CALL SPI              ;send seconds
     MOVLW 0x58            ;58 minutes
     CALL SPI              ;send minutes
     MOVLW 0x16            ;24-hour clock at 16 hours
     CALL SPI              ;send hour
     BCF PORTC,RC2    ;make CE = 0 (end multibyte write)
;-- SPI write/read subroutine
SPI  MOVWF SSPBUF          ;load SSPBUF for transfer
WAIT BTFSS SSPSTAT,BF      ;wait for all bits
     BRA WAIT
     MOVF SSPBUF,W         ;get the received byte
     RETURN                ;return with byte in WREG
     END
```

Setting the date in Assembly

Program 16-2 shows how to set the date to October 19th, 2004.

```
;Program 16-2: Setting the Date
     MOVLW 0x00
     MOVWF SSPSTAT ;read at middle, send on active edge
     MOVLW 0x22
     MOVWF SSPCON1         ;master SPI enable, Fosc / 64
     CLRF TRISC            ;make PORTC output
     BSF TRISC,SDI         ;except SDI
     BSF  PORTC,RC2        ;enable the RTC
     MOVLW 0x8F            ;DS1306 control register address
     CALL SPI
     MOVLW 0x00            ;clear WP bit for write
     CALL SPI
     BCF PORTC,RC2         ;turn off RTC
;-- send the date to DS1306
     BSF  PORTC,RC2        ;enable the RTC
     MOVLW 0x84            ;date register address
     CALL SPI              ;send address
     MOVLW 0x19            ;19th of the month
     CALL SPI              ;send date
     MOVLW 0x10            ;October
     CALL SPI              ;send month
     MOVLW 0x04            ;2004
     CALL SPI              ;send year
     BCF PORTC,RC2         ;disable RTC
;-- SPI write/read subroutine
```

```
SPI  MOVWF SSPBUF          ;load SSPBUF for transfer
WAIT BTFSS SSPSTAT,BF      ;wait for all bits
     BRA WAIT
     MOVF SSPBUF,W         ;get the received byte
     RETURN                ;return with byte in WREG
     END
```

RTCs setting, reading, and displaying time and date

Program 16-3 is the complete Assembly code for setting, reading, and displaying the time and date. The times and dates are sent to the IBM PC screen via the serial port after they are converted from packed BCD to ASCII.

```
;Program 16-3
#include p18f458.inc
D1uL  EQU D'2'              ;1 microsecond delay byte
DR1uL EQU 0x0D          ;register for 1 microsecond delay
DAY   EQU 10H               ;for day of the week
MON   EQU 11H               ;fileReg starting with month
DAT   EQU 12H               ;for day of the month
YR    EQU 13H               ;for year
HR    EQU 14H               ;for hour
MIN   EQU 15H               ;for minutes
SEC   EQU 16H               ;for seconds
CNT   EQU 20H               ;for counter
TMP   EQU 21H               ;for conversions
      MOVLW 0x00
      MOVWF SSPSTAT ;read at middle, send on active edge
      MOVLW 0x22
      MOVWF SSPCON1         ;master SPI enable, Fosc / 64
      CLRF TRISC            ;make PORTC output
      BSF TRISC,SDI         ;except SDI
      BSF TRISC,RX          ;and RX
;-- enable USART communication
      MOVLW B'00100000'     ;enable transmit and low baud
      MOVWF TXSTA           ;write to reg
      MOVLW D'15'       ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG           ;write to reg
      BCF TRISC, TX     ;make TX pin of PORTC an output pin
      BSF RCSTA, SPEN       ;enable the serial port
;-- start a new line for USART communications
      MOVLW 0x0A            ;form feed
      CALL TRANS
      MOVLW 0x0D            ;new line
      CALL TRANS
;-- send control byte to DS1306
      BSF   PORTC,RC2       ;enable the RTC
      CALL SDELAY
      MOVLW 0x8F            ;control register address
      CALL SPI
      MOVLW 0x00            ;clear WP bit for write
```

```
        CALL SPI
        BCF PORTC,RC2          ;disable RTC
        CALL SDELAY
;-- send the time followed by date
        BSF  PORTC,RC2         ;enable the RTC
        MOVLW 0x80        ;seconds register address for write
        CALL SPI               ;send address
        MOVLW 0x55             ;55 seconds
        CALL SPI               ;send seconds
        MOVLW 0x58             ;58 minutes
        CALL SPI               ;send minutes
        MOVLW 0x16             ;24-hour clock at 16 hours
        CALL SPI               ;send hour
        MOVLW 0x3              ;Tuesday
        CALL SPI               ;send day of the week
        MOVLW 0x19             ;19th of the month
        CALL SPI               ;send day of the month
        MOVLW 0x10             ;October
        CALL SPI               ;send month
        MOVLW 0x04             ;2004
        CALL SPI               ;send year
        BCF PORTC,RC2          ;disable RTC
        CALL SDELAY
;-- get the time and date from DS1306
RDA  BSF  PORTC,RC2            ;enable the RTC
        CALL SDELAY
        MOVLW 0x00        ;seconds register address for read
        CALL SPI               ;send address to DS1306
        CALL SPI               ;start getting time/date
        MOVWF SEC              ;save the seconds
        CALL SPI               ;get the minutes
        MOVWF MIN              ;save the minutes
        CALL SPI               ;get the hour
        MOVWF HR               ;save the hour
        CALL SPI               ;get the day
        MOVWF DAY              ;save the day
        CALL SPI               ;get the date
        MOVWF DAT              ;save the date
        CALL SPI               ;get the month
        MOVWF MON              ;save the month
        CALL SPI               ;get the year
        MOVWF YR               ;save the year
        BCF PORTC,RC2          ;disable RTC
;-- convert packed BCD to ASCII and display
        LFSR FSR0,0x11  ;address of fileReg for time/date
        MOVLW D'6'             ;6 bytes of data to display
        MOVWF CNT              ;set up the counter
SND  MOVFF INDF0,TMP           ;get the data for high nibble
        MOVLW 0xF0             ;clear low nibble
        ANDWF TMP,F            ;keep in TMP register
        SWAPF TMP,F            ;switch high and low nibbles
```

```
      MOVLW 0x30              ;convert to ASCII
      IORWF TMP,W             ;put in WREG
      CALL TRANS              ;display the data
      MOVFF  POSTINC0,TMP     ;get the data and point to next
      MOVLW 0x0F              ;clear high nibble
      ANDWF TMP,F             ;keep in TMP register
      MOVLW 0x30              ;convert to ASCII
      IORWF TMP,W             ;put in WREG
      CALL TRANS              ;display the data
      MOVLW ':'
      CALL TRANS
      DECFSZ CNT              ;Is it the last one?
      BRA SND                 ;no
      MOVLW 0x0D              ;line feed
      CALL TRANS
      BRA RDA    ;keep reading time/date and display them
 ;-- SPI write/read subroutine
 SPI  MOVWF SSPBUF            ;load SSPBUF for transfer
 WAIT BTFSS SSPSTAT,BF        ;wait for all bits
      BRA WAIT
      MOVF SSPBUF,W           ;get the received byte
      RETURN                  ;return with byte in WREG
;----serial data transfer subroutine
TRANS BTFSS PIR1, TXIF     ;wait until the last bit is gone
      BRA TRANS               ;stay in loop
      MOVWF TXREG      ;load the value to be transmitted
      RETURN                  ;return to caller
;----short delay
SDELAY: MOVLW   D1uL          ;low byte of delay
      MOVWF DR1uL             ;store in register
DS1   DECF DR1uL,F            ;stay until DR1uL becomes 0
      BNZ  DS1
      RETURN
      END
```

Review Questions

1. True or false. All of the RAM contents of the DS1306 are nonvolatile.
2. How many bytes of RAM in the DS1306 are set aside for the clock and date?
3. How many bytes of RAM in the DS1306 are set aside for general-purpose applications?
4. True or false. The DS1306 has a single pin for Din.
5. Which pin of the DS1306 is used for Clock in SPI connection?
6. True or false. To use the DS1306 in SPI mode, we make SERMODE = GND.

SECTION 16.3: DS1306 RTC PROGRAMMING IN C

In this section, we program the DS1306 in PIC18 C language. Before you embark on this section, make sure you understand the basic concepts of the DS1306 chip covered in the first section.

Setting the time and date in C

Program 16-4C shows how to set the time and date for the DS1306 configuration in Figure 16-12.

```
//Program 16-4C : Setting time and date
#include <p18f458.h>
unsigned char SPI(unsigned char);
void SDELAY(int ms);
void main()
  {
     SSPSTAT = 0; //read at middle, send on active edge
     SSPCON1 = 0x22;       //master SPI enable, Fosc / 64
     TRISC = 0;            //make PORTC output
     TRISCbits.TRISC4 = 1; //except SDI
     TRISCbits.TRISC7 = 1; //and RX
     PORTCbits.RC2 = 1;    //enable the RTC
     SDELAY(1);
     SPI(0x8F);            //control register address
     SPI(0x00);            //clear WP bit for write
     PORTCbits.RC2 = 0;    //end of single-byte write
     SDELAY(1);
     PORTCbits.RC2 = 1;    //begin multibyte write
     SPI(0x80);            //seconds register address
     SPI(0x55);            //55 seconds
     SPI(0x58);            //58 minutes
     SPI(0x16);            //24-hour clock at 16 hours
     SPI(0x3);             //Tuesday
     SPI(0x19);            //19th of the month
     SPI(0x10);            //October
     SPI(0x04);            //2004
     PORTCbits.RC2 = 0;    //end multibyte write
     SDELAY(1);
     }
//-- SPI Write/Read subroutine
unsigned char SPI(unsigned char myByte)
  {
     SSPBUF = myByte;           //load SSPBUF for transfer
     while(!SSPSTATbits.BF);    //wait for all bits
     return SSPBUF;         //return with received byte
     }
```

Reading and displaying the time and date in C

Program 16-5C shows how to read the time, convert it to ASCII, and send it to the PC screen via the serial port.

```
//Program 16-5C : Reading and Displaying Time
#include <p18f458.h>
unsigned char SPI(unsigned char);
void TRANS(unsigned char);
void BCDtoASCIIandSEND(unsigned char);
void SDELAY(int ms);

void main()
  {
  unsigned char data[7];   //holds date and time
  unsigned char tmp;       //for BCD to ASCII conversion
  int i;
  SSPSTAT = 0;         //read at middle, send on active edge
  SSPCON1 = 0x22;          //master SPI enable, Fosc / 64
  TRISC   = 0;             //make PORTC output
  TRISCbits.TRISC4 = 1;    //except SDI
  TRISCbits.TRISC7 = 1;    //and RX
  TXSTA = 0x20;            //enable transmit and low baud
  SPBRG = 15;          //9600 bps (Fosc / (64 * Speed) - 1)
  RCSTAbits.SPEN = 1;      //enable the serial port
  TRANS(0x0A);             //form feed
  TRANS(0x0D);             //new line
//-- get the time and date from RTC and save them
  while(1)
    {
    PORTCbits.RC2 = 1;     //begin multibyte read
    SDELAY(1);
    SPI(0x00);             //seconds register address
    for(i=0;i<7;i++)
      {
      data[i] = SPI(0x00); //get time/date and save
      }
    PORTCbits.RC2 = 0;     //end of multibyte read
//-- convert time/date and display MM:DD:YY:HH:MM:SS
    BCDtoASCIIandSEND(data[5]); //the month
    BCDtoASCIIandSEND(data[4]); //the date
    BCDtoASCIIandSEND(data[6]); //the year
    BCDtoASCIIandSEND(data[2]); //the hour
    BCDtoASCIIandSEND(data[1]); //the minute
    BCDtoASCIIandSEND(data[0]); //the second
    TRANS(0x0D);           //new line
    }
  }
```

```
//-- SPI Write/Read
unsigned char SPI(unsigned char myByte)
  {
  SSPBUF = myByte;
  while(!SSPSTATbits.BF);
  return SSPBUF;
  }
void TRANS(unsigned char myChar) //serial data transfer
  {
  while(!PIR1bits.TXIF);
  TXREG = myChar;     //load the value to be transmitted
  }
void BCDtoASCIIandSEND(unsigned char myValue)
  {
  unsigned char tmp = myValue;
  tmp = tmp & 0xF0;          //mask lower nibble
  tmp = tmp >> 4;            //swap it
  tmp = tmp | 0x30;          //make it ASCII
  TRANS(tmp);                //display
  tmp = myValue;             //for other digit
  tmp = tmp & 0x0F;          //mask upper nibble
  tmp = tmp | 0x30;          //make it ASCII
  TRANS(tmp);                //display
  TRANS(':');                //display separator
  }

void SDELAY(int ms)
  {
  unsigned int i, j;
  for(i=0;i<ms;i++)
    for(j=0;j<135;j++);
  }
```

Review Questions

1. True or false. All of the RAM contents of the DS1306 are volatile.
2. What locations of RAM in the DS1306 are set aside for the clock and date?
3. What locations of RAM in the DS1306 are set aside for general-purpose applications?
4. True or false. The DS1306 has a single pin for Dout.
5. True or false. CE is an output pin.
6. True or false. To use the DS1306 in SPI mode, we make SERMODE = VCC.

SECTION 16.4: ALARM AND INTERRUPT FEATURES OF THE DS1306

In this section, we program the alarm and interrupt features of the DS1306 chip using Assembly and C languages. These powerful features of the DS1306 can be very useful in many real-world applications. In the DS1306 there are two alarms, called Alarm0 and Alarm1, each with their own hardware interrupts. There is also a 1-Hz square wave output pin, which we discuss next. These features are accessed with the Control register shown in Figure 16-13.

0	WP	0	0	0	1-Hz	AIE1	AIE0

WP (Write Protect) If the WP bit is set high, the DS1306 prevents any write operation to its registers. We must make WP = 0 before we can write to any of the registers. Upon power-up, the WP bit is undefined. Therefore, we must make WP = 0 before we can write to any of the registers.

1-Hz (1-Hz output enable) If this bit is set HIGH, it allows the 1-Hz frequency to come out of the 1-Hz pin of the DS1306. By making it LOW, we get High-Z on the 1-Hz pin. Notice that the 1-Hz frequency is automatically generated by the DS1306, but it will not show up at the 1-Hz pin unless we set this bit to HIGH.

AIE0 Alarm interrupt0 enable. If AIE 0= 1, the INT0 pin will be asserted LOW when all three bytes of the real time (hh:mm:ss) are the same as the alarm bytes of hh:mm:ss. Also, if AIE = 1, the cases of once-per-second, once-per-minute, and once-per-hour will assert LOW the INT0 pin.

AIE1 Alarm interrupt1 enable. If AIE1 = 1, the INT1 pin will be asserted HIGH when all three bytes of the real time (hh:mm:ss) are the same as the alarm bytes of hh:mm:ss. Also, if AIE = 1, the cases of once-per-second, once-per-minute, and once-per-hour will assert HIGH the INT1 pin.

Figure 16-13. DS1306 Control Register (Write location address is 8FH)

Programming the 1-Hz feature

The 1-Hz pin of the DS1306 provides us a square wave output of 1-Hz frequency. Internally, the DS1306 generates the 1-Hz square wave automatically but it is blocked. We must enable the 1-Hz bit in the Control register to let it show up on the 1-Hz pin. This is shown below. Because we are writing to a single location, burst mode is not used.

```
      MOVLW 0x00
      MOVWF SSPSTAT     ;middle read, active edge send
      MOVLW 0x22
      MOVWF SSPCON1     ;master SPI enable, Fosc / 64
      CLRF  TRISC       ;make PORTC output
      BSF   TRISC,SDI   ;except SDI
;-- send control byte to enable write first (Figure 16-13)
      BSF   PORTC,RC2   ;enable the RTC
```

```
      CALL  SDELAY
      MOVLW  0x8F            ;Control register address
      CALL  SPI
      MOVLW  0x0             ;clear WP bit for write
      CALL  SPI
      BCF   PORTC,RC2        ;disable RTC
      CALL  SDELAY
;-- send control byte to enable 1 Hz signal after WP = 0
      BSF   PORTC,RC2        ;enable the RTC
      CALL  SDELAY
      MOVLW  0x8F            ;Control register address
      CALL  SPI
      MOVLW  0x04 ;enable 1 Hz signal in Control register
      CALL  SPI
      BCF   PORTC,RC2        ;disable RTC
      CALL  SDELAY
```

Alarm0, Alarm1, and interrupt

There are two time-of-day alarms in the DS1306 chip. They are referred to as Alarm0 and Alarm1. We can access Alarm0 by writing to its registers located at addresses 87H through 8AH, as shown in Table 16-3. Alarm1 is accessed by writing to its registers located at addresses 8BH through 8EH, as shown in Table 16-3. During each clock update, the RTC compares the clock registers and alarm registers. When the values stored in the timekeeping registers of 0, 1, and 2 match the values stored in the alarm registers, the corresponding alarm flag bit (IRQF0 or IRQF1) in the status register will go HIGH. See Figure 16-14. Because polling the IRQxF is too time-consuming, we can enable the AIEx bit in the Control register, and make it a hardware interrupt coming out of the INT0 and INT1 pins.

0	0	0	0	0	0	IRQF1	IRQF0

IRQF0 (Interrupt 0 Request Flag) The IRQF0 bit will go HIGH when all three bytes of the current real time (hh:mm:ss) are the same as the Alarm0 bytes of hh:mm:ss. Also, the cases of once-per-second, once-per-minute, and once-per-hour will assert HIGH the IRQ0 bit. We can use polling to see the status of IRQF0. However, in the Control register, if we make AIE0 = 1, IRQF0 will assert LOW the INT0 pin, making it a hardware interrupt. Any read or write of the Alarm0 registers will clear IRQF0.

IRQF1 (Interrupt 0 Request Flag) The IRQF1 bit will go HIGH when all three bytes of the current real time (hh:mm:ss) are the same as the Alarm1 bytes of hh:mm:ss. Also, the cases of once-per-second, once-per-minute, and once-per-hour will assert HIGH the IRQ1 bit. We can use polling to see the status of IRQF1. However, in the Control register, if we make AIE1 = 1, IRQF1 will assert HIGH the INT1 pin, making it a hardware interrupt. Any read or write of the Alarm1 registers will clear IRQF1.

Figure 16-14. Status Register (Read location address is 10H)

Table 16-3: DS1306 Address Locations for Time, Calendar, and Alarm

Hex Address		Function	Data Range		
Read	Write		D7	BCD	Possible Hex Range
00H	80H	Seconds	0	00–59	00–59
01H	81H	Minute	0	00–59	00–59
02H	82H	Hours, 12-Hour Mode	0	01–12	41–52 AM
		Hours, 12-Hour Mode	0	01–12	61–72 PM
		Hours, 24-Hour Mode	0	00–23	00–23
03H	83H	Day of the Week, Sun = 1	0	01–07	01–07
04H	84H	Day of the Month	0	01–31	01–31
05H	85H	Month	0	01–12	01–12
06H	86H	Year	0	00–99	00–99
07H	87H	SEC Alarm**0**	0 or 1	00–59	00–59 or 89–A9
08H	88H	MIN Alarm**0**	0 or 1	00–59	00–59 or 89–A9
09H	89H	Hour Alarm**0**, 12-Hour	0 or 1	01–12	41–52 or C1–A2 AM
		Hour Alarm**0**, 12-Hour	0 or 1	01–12	61–72 or D1–F2 PM
		Hour Alarm**0**, 24-Hour	0 or 1	00–23	00–23 or 80–A3
0A	8A	Day Alarm**0**	0 or 1	1–7	01–07
0BH	8BH	SEC Alarm**1**	0 or 1	00–59	00–59 or 89–A9
0CH	8CH	MIN Alarm**1**	0 or 1	00–59	00–59 or 89–A9
0DH	8DH	Hour Alarm**1**, 12-Hour	0 or 1	01–12	41–52 or C1–A2 AM
		Hour Alarm**1**, 12-Hour	0 or 1	01–12	61–72 or D1–F2 PM
		Hour Alarm**1**, 24-Hour	0 or 1	00–23	00–23 or 80–A3
0EH	8EH	Day Alarm**1**	0 or 1	1–7	01–07
0FH	8FH	CONTROL REGISTER			
10H	90H	STATUS REGISTER			
11H	91H	TRICKLE REGISTER			
12–1FH	82–9FH	RESERVED			
20–7FH	A0–FFH	96-BYTE USER RAM			

Alarm and IRQ output pins

The alarm interrupts of INT0 and INT1 can be programmed to occur at rates of (a) once per week (b) once per day, (c) once per hour, (d) once per minute, and (f) once per second. Next, we look at each of these.

Once-per-day alarm

Table 16-3 shows the address locations belonging to the alarm seconds, alarm minutes, alarm hours, and alarm days. Notice the D7 bits of these locations. An alarm is generated every day when D7 of the day alarm location is set to HIGH. Therefore, to program the alarm for once-per-day, we must (a) write the desired time for the alarm into the hour, minute, and second of Alarm locations, and (b) set HIGH D7 of the alarm day. See Table 16-4. As the clock keeps the time, when all three bytes of hour, minute, and second for the real-time clock match the values in the alarm hour, minute, and second, the IRQxF flag bit in the Status register of the DS1306 will go high. We can poll the IRQxF bit in the Status register, which is a waste of microcontroller resources, or allow the hardware INTx pin to be activated upon matching the alarm time with the real time. It must be noted that in order

to use the hardware INTx pin of the DS1306 for an alarm, the interrupt-enable bit for alarm in control register (AIEx) must be set HIGH. We will examine the process shortly.

Once-per-hour alarm

To program the alarm for once per hour, we must set HIGH D7 of both the day alarm and hour alarm registers. See Table 16-4.

Once-per-minute alarm

To program the alarm for once per minute, we must set HIGH D7 of all three, day alarm, hour alarm, and minute alarm locations. See Table 16-4.

Once-per-second alarm

To program the alarm for once per second, we must set HIGH D7 of all four locations of alarm day, alarm hour, alarm minute, and alarm second. See Table 16-4.

Once-per-week alarm

To program the alarm for once per week, we must clear D7 of all four locations of alarm day, alarm hour, alarm minute, and alarm second. See Table 16-4.

Table 16-4: DS1306 Time-of-day Alarm Mask Bits

Alarm Register Mask Bits (D7)				
Seconds	**Minutes**	**Hours**	**Days**	**Function**
1	1	1	1	Alarm once per second
0	1	1	1	Alarm when seconds match (once-per-minute)
0	0	1	1	Alarm when minutes and seconds match (once-per-hour)
0	0	0	1	Alarm when hours, minutes, and seconds match (once-per-day)
0	0	0	0	Alarm when day, hours, minutes, and seconds match (once-per-week)

Example 16-2

Using Table 16-4, find the values we must place in the Alarm1 register if we want to have an alarm activated at 16:05:07, and from then on once-per-minute at 7 seconds past the minute.

Solution:

Because we use 24-hour clock, we have D6 = 0 for the HR register. Therefore, we have 1001 0110 for 16 in BCD. This means that we must put value 96H into register location 8D of the DS1306. Notice that D7 is 1, according to Table 16-4.

For the MIN register, we have 1000 0101 for 05 in BCD. This means that we must put value 85H into register location 8C of the DS1306. Notice that D7 is 1, according to Table 16-4.

For the SEC register we have 0000 0111 for 07 in BCD. This means that we must put value 07H into register location 8B of the DS1306. Notice that D7 is 0 according to Table 16-4.

For once-per-minute to work, we must make sure that D7 of Alarm1 day is also set to 1. See Table 16-4.

Using INT0 of DS1306 to activate the PIC18 interrupt

We can connect the INT0 bit of the DS1306 to the external interrupt pin of the PIC18 (INT0). See Figure 16-15. This allows us to perform a task once per day, once per minute, and so on. Example 16-2 shows the values needed for the Alarm0 registers. Program 16-6 uses the Alarm0 interrupt (INT0) to send the message "YES" to the serial port once per minute, at exactly 8 seconds past the minute.

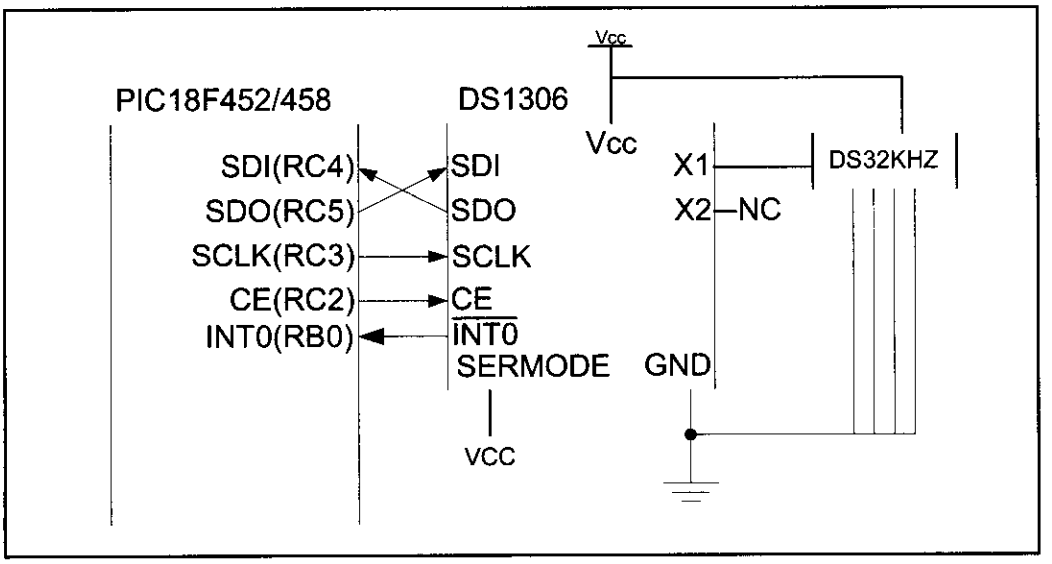


Figure 16-15. DS1306 Connection to PIC18 with Hardware INT0

```
;Program 16-6
D1uL  EQU   D'2'              ;1 microsecond delay byte
DR1uL EQU   0x0D        ;register for 1 microsecond delay
      ORG   0x00
      BRA   MAIN              ;bypass INT vector table
      ORG   0x08
      BTFSC  INTCON,INT0IF    ;Was it INT0?
      BRA   INT0_ISR          ;yes, go to INT0 ISR
      RETFIE
      ORG   0x28
;-- initialize SPI, INT0, and USART
MAIN  CLRF  TRISC             ;make PORTC output
      BSF   TRISC,SDI         ;except SDI
      BSF   TRISC,RX          ;and RX
      BSF   TRISB,INT0        ;make RB0 input for interrupt
      MOVLW 0x00
      MOVWF SSPSTAT           ;middle read, active edge send
      MOVLW 0x22
      MOVWF SSPCON1           ;master SPI enable, Fosc / 64
      BCF   INTCON2,INTEDG0   ;make INT0 negative-edge
                              ;triggered
      BSF   INTCON,INT0IE     ;enable INT0
      MOVLW B'00100000';enable transmit and choose low baud
      MOVWF TXSTA             ;write to reg
      MOVLW D'15'       ;9600 bps (Fosc / (64 * Speed) - 1)
      MOVWF SPBRG             ;write to reg
```

```
      BCF   TRISC, TX  ;make TX pin of PORTC an output pin
      BSF   RCSTA, SPEN      ;enable the serial port
      BSF   INTCON,GIE       ;enable interrupts globally
;-- send control byte to enable write
      BSF   PORTC,RC2        ;enable the RTC
      CALL  SDELAY
      MOVLW 0x8F             ;control register
      CALL  SPI
      MOVLW 0x0              ;clear WP bit for write
      CALL  SPI
      BCF   PORTC,RC2        ;disable RTC
      CALL  SDELAY
;-- send the data
      BSF   PORTC,RC2  ;enable for multibyte write
      MOVLW 0x87             ;Alarm0 address
      CALL SPI               ;send address
      MOVLW 0x08             ;alarm at 8 seconds
      CALL SPI               ;send second
      MOVLW 0x80             ;once-per-minute
      CALL SPI               ;send minute
      MOVLW 0x80             ;once-per-minute
      CALL SPI               ;send hour
      MOVLW 0x80             ;once-per-minute
      CALL SPI               ;send day
      BCF PORTC,RC2          ;end of multibyte write
      CALL SDELAY
;-- send control byte to enable INT0
      BSF   PORTC,RC2        ;enable the RTC
      CALL SDELAY
      MOVLW 0x8F             ;control register of DS1306
      CALL SPI
      MOVLW 0x01             ;enable INT0 pin of DS1306
      CALL SPI
      BCF PORTC,RC2          ;disable RTC
      CALL  SDELAY
LOOP  BRA LOOP               ;wait for interrupt
;-- service Alarm0
INT0_ISR
      BSF   PORTC,RC2        ;enable the RTC
      CALL SDELAY
      MOVLW 0x8F             ;control register
      CALL SPI
      MOVLW 0x04             ;1 Hz on, Alarm0 off
      CALL SPI
      BCF PORTC,RC2          ;disable RTC
      CALL SDELAY
;-- send Alarm0 seconds to reset alarm
      BSF   PORTC,RC2        ;enable the RTC
      CALL SDELAY
      MOVLW 0x87             ;Alarm0 seconds register
      CALL SPI
```

```
      MOVLW 0x08              ;at 8 seconds
      CALL SPI
      BCF PORTC,RC2           ;disable RTC
      CALL SDELAY
;-- begin displaying
      MOVLW upper(MESSAGE)
      MOVWF TBLPTRU
      MOVLW high(MESSAGE)
      MOVWF TBLPTRH
      MOVLW low(MESSAGE)
      MOVWF TBLPTRL
NEXT  TBLRD*+                 ;read the characters
      MOVF  TABLAT,W          ;place it in WREG
      IORLW 0x0
      BZ    OVER              ;if end of line, start over
      CALL  TRANS             ;send char to serial port
      BRA   NEXT              ;repeat for the next character
;-- send control byte to enable INT0
OVER  BSF   PORTC,RC2         ;enable the RTC
      CALL  SDELAY
      MOVLW 0x8F              ;control register
      CALL  SPI
      MOVLW 0x01              ;1 Hz off, Alarm0 on
      CALL  SPI
      BCF PORTC,RC2           ;disable RTC
      CALL  SDELAY
      BCF   INTCON,INT0IF
      RETFIE
;-- SPI subroutine
;-- serial data transfer subroutine
;-- delay for SPI communications
      RETURN;SEE PREVIOUS PROGRAMS FOR ABOVE SUBROUTINES
;--message to be displayed upon interrupt
MESSAGE:   DB    0x0A,0x0D,"Yes",0
      END
```

The following is the C version of the above program.

```
//Program 16-6C
#include <p18f458.h>
//INSERT FUNCTION PROTOTYPES
#pragma interrupt chk_isr  //used for high priority inter-
rupt only
void chk_isr (void)
{
  if (INTCONbits.INT0IF==1)//INT0 caused interrupt?
  INT0_ISR( );             //Yes. Execute INT0 program
}
#pragma code My_HiPrio_Int=0x0008 //high-priority interrupt
void My_HiPrio_Int (void)
{
```

```
  _asm
  GOTO chk_isr
  _endasm
}
#pragma code
void main(void)
  {
//-- initialize SPI, INT0, and USART
  TRISC=0x90;      //make PORTC output, except SDI and RX
  TRISBbits.TRISB0=1;       //make RB0 input for interrupt
  SSPSTAT=0x0;              //middle read, active edge send
  SSPCON1=0x22;             //master SPI enable, Fosc / 64
  INTCON2bits.INTEDG0=0;    //make INT0 negative edge
                            //triggered
  INTCONbits.INT0IE=1;      //enable INT0
  TXSTA=0x20;          //enable transmit and choose low baud
  SPBRG=15;            //9600 bps (Fosc / (64 * Speed) - 1)
  RCSTAbits.SPEN=1;         //enable the serial port
  INTCONbits.GIE=1;         //enable interrupts globally
//-- send control byte to enable write
  PORTCbits.RC2=1;          //enable the RTC
  MSDelay(1);
  SPI(0x8F);                //control register address
  SPI(0x0);                 //enable write
  PORTCbits.RC2=0;          //disable RTC
  MSDelay(1);
//-- send the data
  PORTCbits.RC2=1;          //enable the RTC
  MSDelay(1);
  SPI(0x87);                //Alarm0 address
  SPI(0x08);                //alarm at 8 seconds
  SPI(0x80);                //once-per-minute
  SPI(0x80);                //once-per-minute
  SPI(0x80);                //once-per-minute
  PORTCbits.RC2=0;          //disable RTC
  MSDelay(1);
//-- send control byte to enable INT0
  PORTCbits.RC2=1;          //enable the RTC
  MSDelay(1);
  SPI(0x8F);                //control register
  SPI(0x01);                //enable INT0
  PORTCbits.RC2=0;          //disable RTC
  MSDelay(1);
  while(1);                 //wait for interrupt
  }
//-- service Alarm0
void INT0_ISR()
  {
  unsigned char mess[]={0x0D,0x0A,'Y','E','S',0};
  unsigned char i;
  PORTCbits.RC2=1;          //enable the RTC
```

```
  MSDelay(1);
  SPI(0x8F);                  //control register
  SPI(0x04);                  //1 Hz on, Alarm0 off
  PORTCbits.RC2=0;            //disable RTC
  MSDelay(1);
//-- send Alarm0 seconds to reset alarm
  PORTCbits.RC2=1;            //enable the RTC
  MSDelay(1);
  SPI(0x87);                  //Alarm0 seconds register
  SPI(0x08);                  //at 8 seconds
  PORTCbits.RC2=0;            //disable RTC
  MSDelay(1);
//-- begin sending the data
  for(i=0;mess[i]!=0;i++)
   TRANS(mess[i]);
//-- send control byte to enable INT0
  PORTCbits.RC2=1;            //enable the RTC
  MSDelay(1);
  SPI(0x8F);                  //control register
  SPI(0x01);                  //1 Hz offbits. Alarm0 on
  PORTCbits.RC2=0;            //turn off RTC
  INTCONbits.INT0IF=0;
  }
//--SEE PREVIOUS EXAMPLES FOR SUBROUTINES
```

In the last program, we send a message to the serial port to indicate that the alarm has occurred. We can use the 32 kHz output to sound an actual alarm. Because 32 kHz is too high a frequency for human ears, however, we can use multiple D flip flops to bring down the frequency. See Figure 16-16. The modification of Program 16-6 for Figure 16-16 is left to the reader.

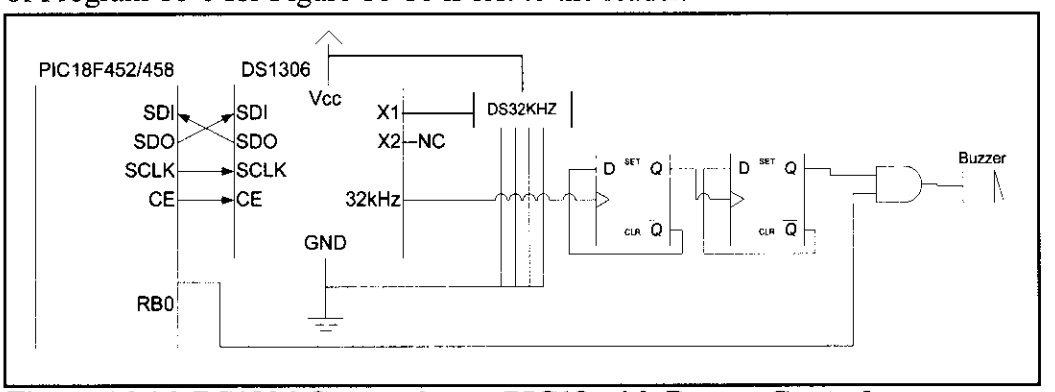


Figure 16-16. DS1306 Connection to PIC18 with Buzzer Control

Review Questions

1. Which bit of the Control register belongs to the 1-Hz pin?
2. True or false. The INT0 pin is an input for the DS1306.
3. True or false. The INT0 pin is active-LOW.
4. Which bit of the Control register belongs to the Alarm1 interrupt?
5. Give the address locations for Alarm1.

SUMMARY

This chapter began by describing the SPI bus connection and protocol. We also discussed the function of each pin of the DS1306 RTC chip. The DS1306 can be used to provide a real-time clock and dates for many applications. Various features of the RTC were explained, and numerous programming examples were given.

PROBLEMS

SECTION 16.1: SPI BUS PROTOCOL

1. True or false. The SPI bus needs an external clock.
2. True or false. The SPI CE is active-LOW.
3. True or false. The SPI bus has a single Din pin.
4. True or false. The SPI bus has multiple Dout pins.
5. True or false. When the SPI device is used as a slave, the SCLK is an input pin.
6. True or false. In SPI devices, data is transferred in 8-bit chunks.
7. True or false. In SPI devices, each bit of information (data, address) is transferred with a single clock pulse.
8. True or false. In SPI devices, the 8-bit data is followed by an 8-bit address.
9. In term of data pins, what is the difference between the SPI and 3-wire connections?
10. How does the SPI protocol distinguish between the read and write cycles?

SECTION 16.2: DS1306 RTC INTERFACING AND PROGRAMMING

11. The DS1306 DIP package is a(n) ____-pin package.
12. Which pin is assigned as primary V_{cc}?
13. In the DS1306, how many pins are designated as address/data pins?
14. True or false. The DS1306 needs an external crystal oscillator.
15. True or false. The DS1306's crystal oscillator and heat affect the time-keeping accuracy.
16. In DS1306, what is the maximum year that it can provide?
17. Describe the functions of pins SDI, SDO, and SCLK.
18 CE is an _____ (input, output) pin.
19. The CE pin is normally _____ (LOW, HIGH) and needs a _____ (LOW, HIGH) signal to be activated.
20. Who keeps the contents of the DS1306 time and date registers if power to the primary V_{cc} pin is cut off?
21. Vbat pin stands for _____ and is an_____ (input, output) pin.
22. For the DS1306 chip, pin Vcc2 is connected to _____ (V_{cc}, GND).
23. SERMODE is an _____ (input, output) pin and it is connected to _____ for SPI mode.
24. Vcc1 is an _____ (input, output) pin and is connected to _____ voltage.
25. 1-Hz is an _____ (input, output).

26. INT0 is an _____ (input, output) pin.
27. 32KHz is an _____ (input, output) pin.
28. INT1 is an _____ (input, output) pin.
29. DS1306 has a total of _____bytes of locations. Give the addresses for read and write operations.
30. What are the contents of the DS1306 time and date registers if power to the V_{cc} pin is lost?
31. What are the contents of the general-purpose RAM locations if power to the V_{cc1} is lost?
32. When does the DS1306 switch to a battery energy source?
33. What are the addresses assigned to the real-time clock (time) registers?
34. What are the addresses assigned to the calendar?
35. Which register is used to set the AM/PM mode? Give the bit location of that register.
36. Which register is used to set the 24-hour mode? Give the bit location of that register.
37. At what memory location does the DS1306 store the year 2007?
38. What is the address of the last location of RAM for the DS1306?
39. True or false. The DS1306 provides data in BCD format only.
40. Write a program to get the year data in BCD and send it to ports PORTB and PORTD.
41. Write a program to get the hour and minute data in BCD and send it to ports PORTB and PORTD.
42. Write a program to set the time to 9:15:05 PM.
43. Write a program to set the time to 22:47:19.
44. Write a program to set the date to May 14, 2009.
45. What are the roles of Vbat and Vcc2?

SECTION 16.3: DS1306 RTC PROGRAMMING IN C

46. Write a C program to display the time in AM/PM mode.
47. Write a C program to get the year data in BCD and send it to ports PORTB and PORTD.
48. Write a C program to get the hour and minute data and send it to ports PORTB and PORTD.
49. Write a C program to set the time to 9:15:05 PM.
50. Write a C program to set the time to 22:47:19.
51. Write a C program to set the date to May 14, 2009.
52. In Question 51, how does the RTC keep track of the century?

SECTION 16.4: ALARM AND INTERRUPT FEATURES OF THE DS1306

53. INT0 is an _____ (input, output) pin and active-_____(LOW, HIGH).
54. 1-Hz is an _____ (input, output) pin.
55. Give the bit location of the Control register belonging to the alarm interrupt. Show how to enable it.

56. Give the bit location of the Control register belonging to the 1-Hz pin. Show how to enable it.
57. Give the bit location of the Status register belonging to the Alarm0 interrupt.
58. Give the bit location of the Status register belonging to the Alarm1 interrupt.
59. True or False. For the 32KHz output pin, the frequency is set and cannot be changed.
60. Give sources of interrupts that can activate the INT1 pin.
61. Why do we want to direct the AIE0 (Alarm0 flag) to an IRQ pin?
62. What is the difference between the IRQF0 and AIE0 bits?
63. What is the difference between the IRQF1 and AIE1 bits?
64. How do we allow the square wave to come out of the 1-Hz pin?
65. Which register is used to set the once-per-second Alarm1?
66. Explain how the IRQ1F pin is activated due to the once-per-minute alarm option.

ANSWERS TO REVIEW QUESTIONS

SECTION 16.1: SPI BUS PROTOCOL

1. True
2. True
3. False
4. False
5. In single-byte mode, after each byte, the CE pin must go LOW before the next cycle. In burst mode, the CE pin stays HIGH for the duration of the burst (multibyte) transfer.

SECTION 16.2: DS1306 RTC INTERFACING AND PROGRAMMING

1. True. Only if Vbat is connected to an external battery.
2. 7
3. 96
4. True
5. Pin 11 is SCLK.
6. False. SERMODE = Vcc

SECTION 16.3: DS1306 RTC PROGRAMMING IN C

1. True
2. 0–6
3. 20–7FH
4. True
5. False
6. False

SECTION 16.4: ALARM AND INTERRUPT FEATURES OF THE DS1306

1. Bit 2
2. False
3. True
4. Bit 1
5. Byte addresses of 0B–0E (in hex) for read and 8B–8E (in hex) for write

CHAPTER 17

MOTOR CONTROL: RELAY, PWM, DC, AND STEPPER MOTORS

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Describe the basic operation of a relay**
- **>> Interface the PIC18 with a relay**
- **>> Describe the basic operation of an optoisolator**
- **>> Interface the PIC18 with an optoisolator**
- **>> Describe the basic operation of a stepper motor**
- **>> Interface the PIC18 with a stepper motor**
- **>> Code PIC18 programs to control and operate a stepper motor**
- **>> Define stepper motor operation in terms of step angle, steps per revolution, tooth pitch, rotation speed, and RPM**
- **>> Describe the basic operation of a DC motor**
- **>> Interface the PIC18 with a DC motor**
- **>> Code PIC18 programs to control and operate a DC motor**
- **>> Describe how PWM is used to control motor speed**
- **>> Code CCP programs to control and operate a DC motor**
- **>> Code ECCP programs to control and operate a DC motor**

This chapter discusses motor control and shows PIC18 interfacing with relays, optoisolators, stepper motors, and DC motors. In Section 17.1, the basics of relays and optoisolators are described. Then we show their interfacing with the PIC18. In Section 17.2, stepper motor interfacing with the PIC18 is shown. The characteristics of DC motors are discussed in Section 17.3, along with their interfacing to the PIC18. We will also discuss the topic of PWM (pulse width modulation). In Section 17.4, the CCP feature of PIC18 is used to control DC motors, while the ECCP usage in motor control is shown in Section 17.5. We use both Assembly and C in our programming examples.

SECTION 17.1: RELAYS AND OPTOISOLATORS

This section begins with an overview of the basic operations of electromechanical relays, solid-state relays, reed switches, and optoisolators. Then we describe how to interface them to the PIC18. We use both Assembly and C language programs to demonstrate their control.

Electromechanical relays

A *relay* is an electrically controllable switch widely used in industrial controls, automobiles, and appliances. It allows the isolation of two separate sections of a system with two different voltage sources. For example, a +5 V system can be isolated from a 120 V system by placing a relay between them. One such relay is called an electromechanical (or electromagnetic) relay (EMR) as shown in Figure 17-1. The EMRs have three components: the coil, spring, and contacts. In Figure 17-1, a digital +5 V on the left side can control a 12 V motor on the right side without any physical contact between them. When current flows through the coil, a magnetic field is created around the coil (the coil is energized), which causes the armature to be attracted to the coil. The armature's contact acts like a switch and closes or opens the circuit. When the coil is not energized, a spring pulls the armature to its normal state of open or closed. In the block diagram for electomechanical relays (EMR) we do not show the spring, but it does exist internally. There are all types of relays for all kinds of applications. In choosing a relay the following characteristics need to be considered:

1. The contacts can be normally open (NO) or normally closed (NC). In the NC type, the contacts are closed when the coil is not energized. In the NO, the contacts are open when the coil is unenergized.
2. There can one or more contacts. For example, we can have SPST (single pole, single throw), SPDT (single pole, double throw), and DPDT (double pole, double throw) relays.
3. The voltage and current needed to energize the coil. The voltage can vary from a few volts to 50 volts, while the current can be from a few mA to 20 mA. The relay has a minimum voltage, below which the coil will not be energized. This minimum voltage is called the "pull-in" voltage. In the datasheet for relays we might not see current, but rather coil resistance. The V/R will give you the pull-in current. For example, if the coil voltage is 5 V, and the coil resistance is 500 ohms, we need a minimum of 10 mA (5 V/500 ohms = 10 mA) pull-in current.

4. The maximum DC/AC voltage and current that can be handled by the contacts. This is in the range of a few volts to hundreds of volts, while the current can be from a few amps to 40 A or more, depending on the relay. Notice the difference between this voltage/current specification and the voltage/current needed for energizing the coil. The fact that one can use such a small amount of voltage/current on one side to handle a large amount of voltage/current on the other side is what makes relays so widely used in industrial controls. Examine Table 17-1 for some relay characteristics.

Table 17-1: Selected DIP Relay Characteristics (www.Jameco.com)

Part No.	Contact Form	Coil Volts	Coil Ohms	Contact Volts-Current
106462CP	SPST-NO	5 VDC	500	100 VDC-0.5 A
138430CP	SPST-NO	5 VDC	500	100 VDC-0.5 A
106471CP	SPST-NO	12 VDC	1000	100 VDC-0.5 A
138448CP	SPST-NO	12 VDC	1000	100 VDC-0.5 A
129875CP	DPDT	5 VDC	62.5	30 VDC-1 A

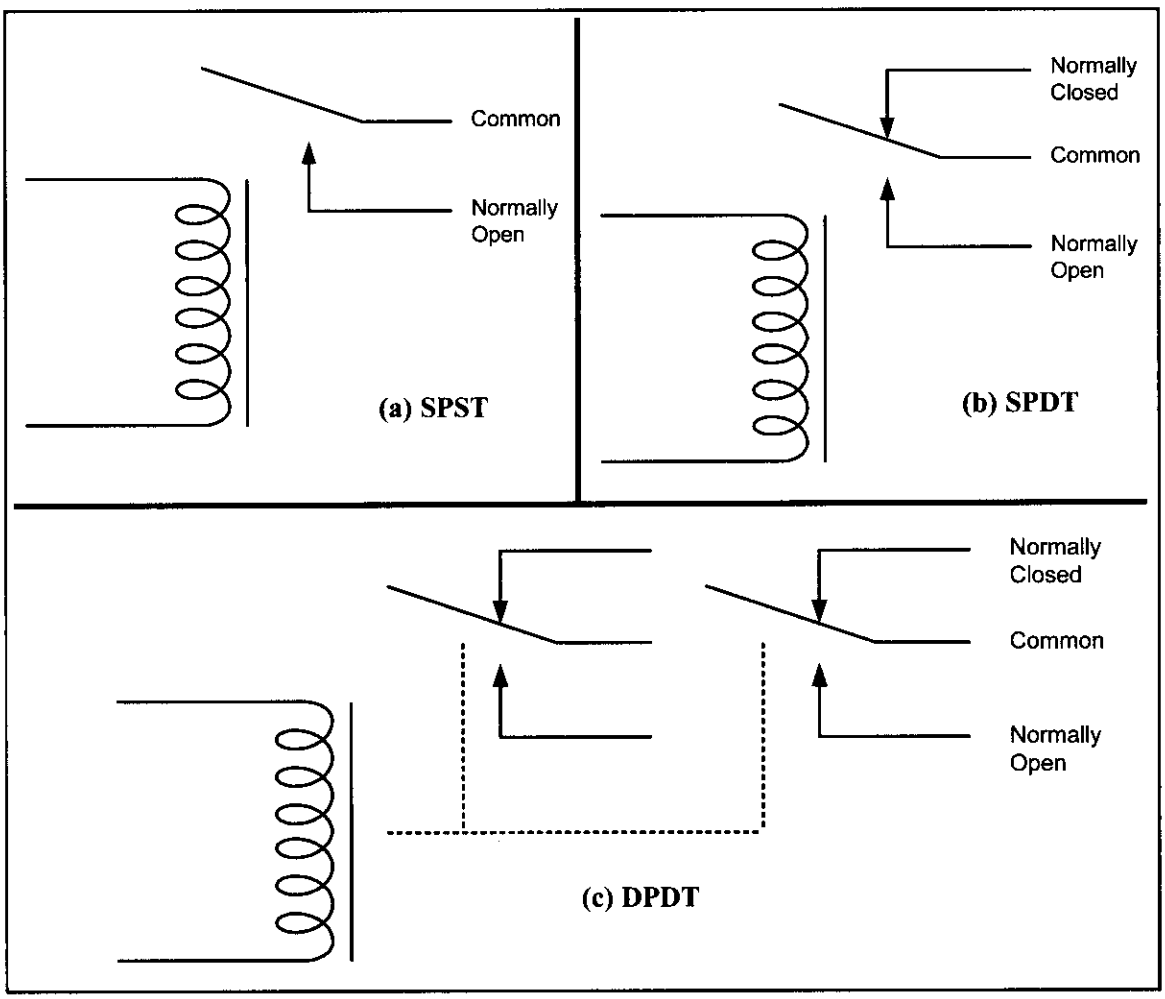


Figure 17-1. Relay Diagrams

Driving a relay

Digital systems and microcontroller pins lack sufficient current to drive the relay. While the relay's coil needs around 10 mA to be energized, the microcontroller's pin can provide a maximum of 1–2 mA current. For this reason, we place a driver, such as the ULN2803, or a power transistor between the microcontroller and the relay as shown in Figure 17-2.

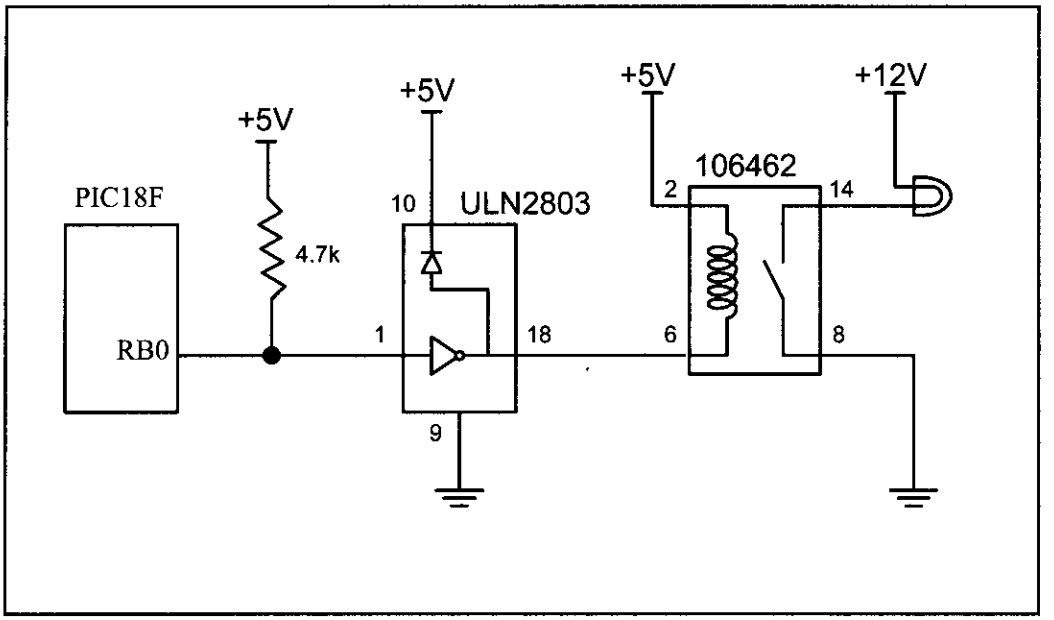


Figure 17-2. PIC18 Connection to Relay

Program 17-1 turns the lamp on and off shown in Figure 17-2 by energizing and de-energizing the relay every few ms.

```
;Program 17-1
R3    SET   0x20        ;set aside location 0x20 for R3
R4    SET   0x21        ;loc. 0x21 for R4
      ORG 0H
      BCF TRISB,0       ;PORTB.0 as output
OVER  BSF PORTB,0       ;turn on the lamp
      CALL DELAY
      BCF   PORTB,0     ;turn off the lamp
      CALL DELAY
      BRA OVER
DELAY MOVLW 0xFF
      MOVWF R4
D1    MOVLW 0xFF
      MOVWF R3
D2    NOP
      NOP
      DECF  R3,F
      BNZ   D2
      DECF R4,F
      BNZ   D1
      RETURN
```

Solid-state relay

Another widely used relay is the solid-state relay. See Table 17-2. In this relay, there is no coil, spring, or mechanical contact switch. The entire relay is made out of semiconductor materials. Because no mechanical parts are involved in solid-state relays, their switching response time is much faster than that of electromechanical relays. Another advantage of the solid-state relay is its greater life expectancy. The life cycle for the electromechanical relay can vary from a few hundred thousand to a few million operations. Wear and tear on the contact points can cause the relay to malfunction after a while. Solid-state relays, however, have no such limitations. Extremely low input current and small packaging make solid-state relays ideal for microprocessor and logic control switching. They are widely used in controlling pumps, solenoids, alarms, and other power applications. Some solid-state relays have a phase control option, which is ideal for motor-speed control and light-dimming applications. Figure 17-3 shows control of a fan using a solid-state relay (SSR).

Table 17-2: Selected Solid-State Relay Characteristics (www.Jameco.com)

Part No.	Contact Style	Control Volts	Contact Volts	Contact Current
143058CP	SPST	4–32 VDC	240 VAC	3 A
139053CP	SPST	3–32 VDC	240 VAC	25 A
162341CP	SPST	3–32 VDC	240 VAC	10 A
172591CP	SPST	3–32 VDC	60 VDC	2 A
175222CP	SPST	3–32 VDC	60 VDC	4 A
176647CP	SPST	3–32 VDC	120 VDC	5 A

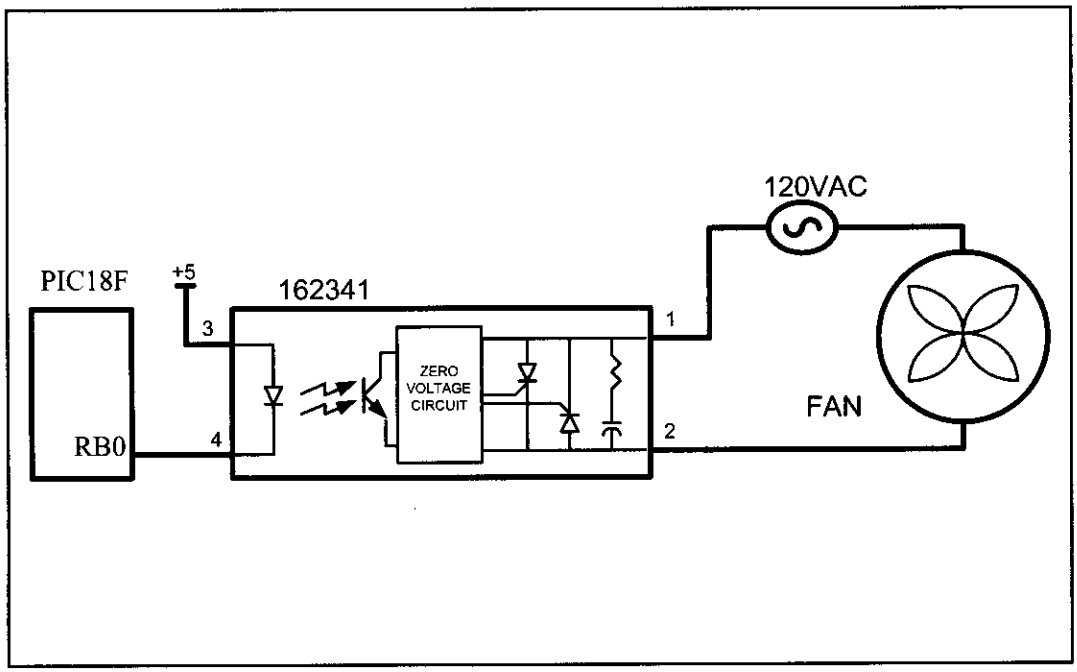


Figure 17-3. PIC18 Connection to a Solid-State Relay

Reed switch

Another popular switch is the reed switch. When the reed switch is placed in a magnetic field, the contact is closed. When the magnetic field is removed, the contact is forced open by its spring. See Figure 17-4. The reed switch is ideal for moist and marine environments where it can be submerged in fuel or water. Reed switches are also widely used in dirty and dusty atmospheres because they are tightly sealed.

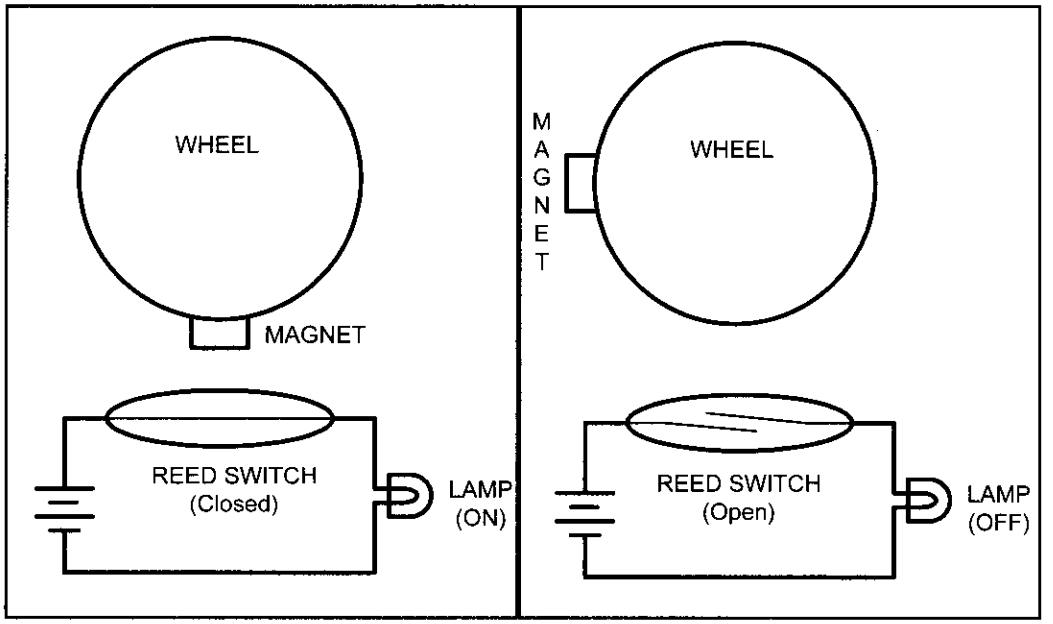


Figure 17-4. Reed Switch and Magnet Combination

Optoisolator

In some applications we use an optoisolator (also called optocoupler) to isolate two parts of a system. An example is driving a motor. Motors can produce what is called back EMF, a high-voltage spike produced by a sudden change of current as indicated in the V = Ldi/dt formula. In situations such as printed circuit board design, we can reduce the effect of this unwanted voltage spike (called ground bounce) by using decoupling capacitors (see Appendix C). In systems that have inductors (coil winding), such as motors, a decoupling capacitor or a diode will not do the job. In such cases we use optoisolators. An optoisolator has an LED (light-emitting diode) transmitter and a photosensor receiver, separated from each other by a gap. When current flows through the diode, it transmits a signal light across the gap and the receiver produces the same signal with the same phase but a different current and amplitude. See Figure 17-5. Optoisolators are also widely used in communication equipment such as modems. This device allows a computer to be connected to a telephone line without risk of damage from power surges. The gap between the transmitter and receiver of optoisolators prevents the electrical current surge from reaching the system.

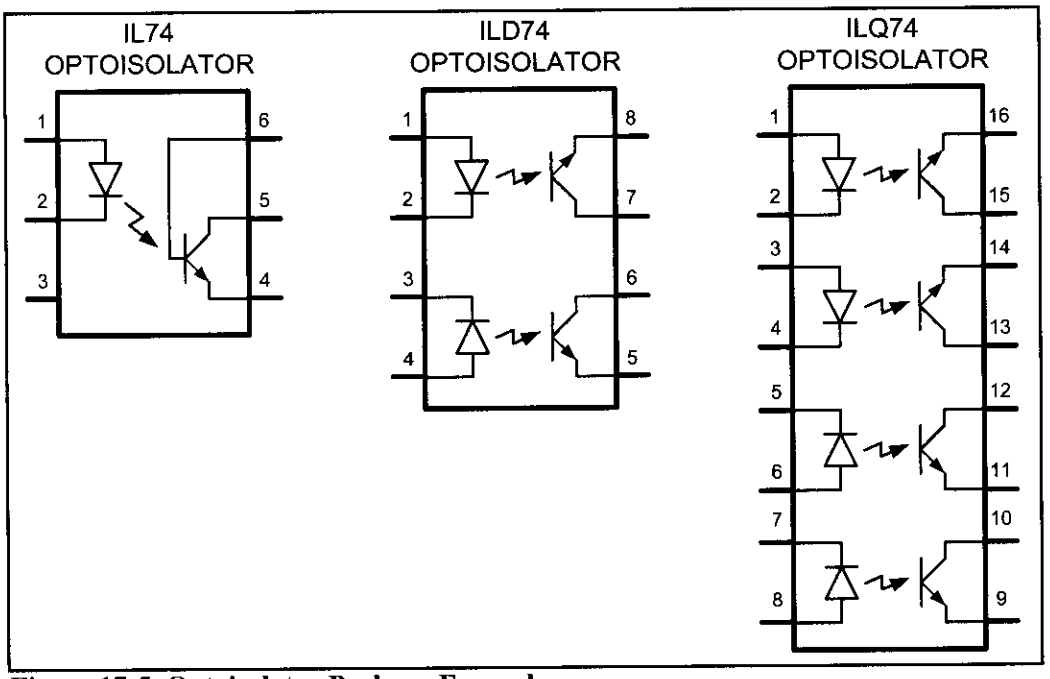


Figure 17-5. Optoisolator Package Examples

Interfacing an optoisolator

The optoisolator comes in a small IC package with four or more pins. There are also packages that contain more than one optoisolator. When placing an optoisolator between two circuits, we must use two separate voltage sources, one for each side, as shown in Figure 17-6. Unlike relays, no drivers need to be placed between the microcontroller/digital output and the optoisolators.

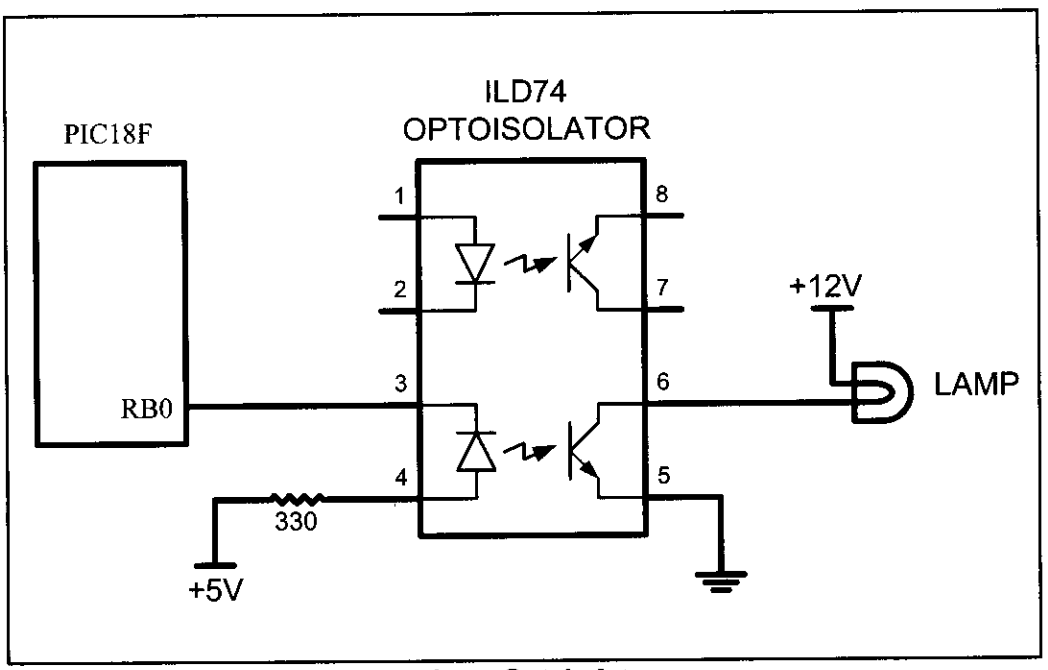


Figure 17-6. Controlling a Lamp via an Optoisolator

Review Questions

1. Give one application where would you use a relay.
2. Why do we place a driver between the microcontroller and the relay?
3. What is an NC relay?
4. Why are relays that use coils called electromechanical relays?
5. What is the advantage of a solid-state relay over EMR?
6. What is the advantage of an optoisolator over an EM relay?

SECTION 17.2: STEPPER MOTOR INTERFACING

This section begins with an overview of the basic operation of stepper motors. Then we describe how to interface a stepper motor to the PIC18. Finally, we use Assembly language programs to demonstrate control of the angle and direction of stepper motor rotation.

Stepper motors

A *stepper motor* is a widely used device that translates electrical pulses into mechanical movement. In applications such as disk drives, dot matrix printers, and robotics, the stepper motor is used for position control. Stepper motors commonly have a permanent magnet *rotor* (also called the *shaft*) surrounded by a *stator* (see Figure 17-7). There are also steppers called variable reluctance *stepper motors* that do not have a permanent magnet rotor. The most common stepper motors have four stator windings that are paired with a center-tapped common as shown in Figure 17-8. This type of stepper motor is commonly referred to as a *four-phase* or unipolar stepper motor. The center tap allows a change of current direction in

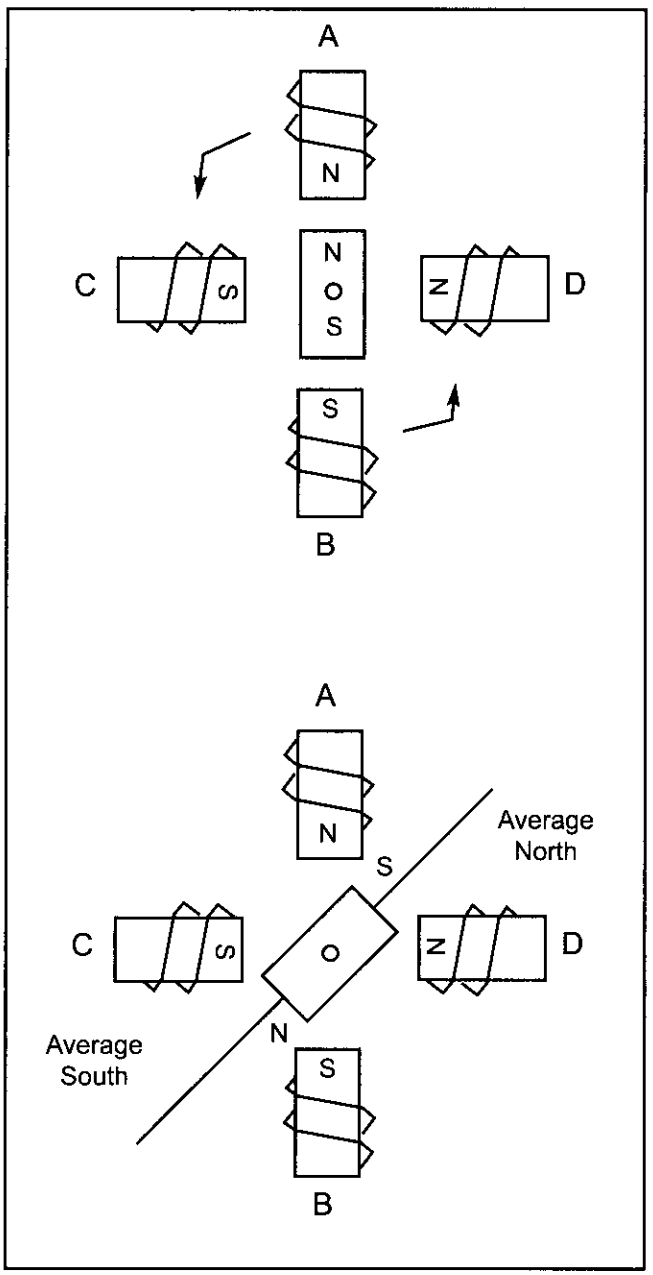


Figure 17-7. Rotor Alignment

each of two coils when a winding is grounded, thereby resulting in a polarity change of the stator. Notice that while a conventional motor shaft runs freely, the stepper motor shaft moves in a fixed repeatable increment, which allows one to move it to a precise position. This repeatable fixed movement is possible as a result of basic magnetic theory where poles of the same polarity repel and opposite poles attract. The direction of the rotation is dictated by the stator poles. The stator poles are determined by the current sent through the wire coils. As the direction of the current is changed, the polarity is also changed causing the reverse motion of the rotor. The stepper motor discussed here has a total of six leads: four leads representing the four stator windings and two commons for the center-tapped leads. As the sequence of power is applied to each stator winding, the rotor will rotate. There are several widely used sequences, each of which has a different degree of precision. Table 17-3 shows a two-phase, four-step stepping sequence.

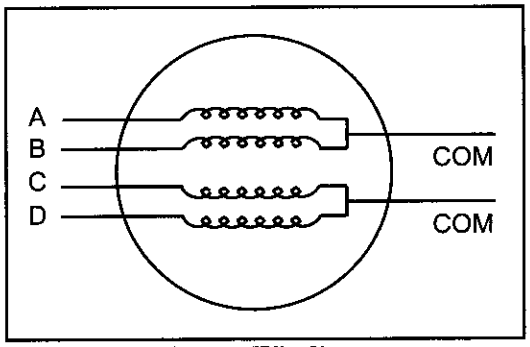


Figure 17-8. Stator Winding Configuration

Note that although we can start with any of the sequences in Table 17-3, once we start we must continue in the proper order. For example, if we start with step 3 (0110), we must continue in the sequence of steps 4, 1, 2, etc.

Table 17-3: Normal Four-Step Sequence

Clockwise ↓

Step #	Winding A	Winding B	Winding C	Winding D
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

↑ Counter-clockwise

Step angle

How much movement is associated with a single step? This depends on the internal construction of the motor, in particular the number of teeth on the stator and the rotor. The *step angle* is the minimum degree of rotation associated with a single step. Various motors have different step angles. Table 17-4 shows some step angles for various motors. In Table 17-4, notice the term *steps per revolution*. This is the total number of steps needed to rotate one complete rotation or 360 degrees (e.g., 180 steps × 2 degrees = 360).

Table 17-4: Stepper Motor Step Angles

Step Angle	Steps per Revolution
0.72	500
1.8	200
2.0	180
2.5	144
5.0	72
7.5	48
15	24

It must be noted that perhaps contrary to one's initial impression, a stepper motor does not need more terminal leads for the stator to achieve smaller steps. All the stepper motors

discussed in this section have four leads for the stator winding and two COM wires for the center tap. Although some manufacturers set aside only one lead for the common signal instead of two, they always have four leads for the stators. See Example 17-1. Next we discuss some associated terminology in order to understand the stepper motor further.

Example 17-1

Describe the PIC18 connection to the stepper motor of Figure 17-9 and code a program to rotate it continuously.

Solution:

The following steps show the PIC18 connection to the stepper motor and its programming:

1. Use an ohmmeter to measure the resistance of the leads. This should identify which COM leads are connected to which winding leads.
2. The common wire(s) are connected to the positive side of the motor's power supply. In many motors, +5 V is sufficient.
3. The four leads of the stator winding are controlled by four bits of the PIC18 port (RB0–RB3). Because the PIC18 lacks sufficient current to drive the stepper motor windings, we must use a driver such as the ULN2003 to energize the stator. Instead of the ULN2003, we could have used transistors as drivers, as shown in Figure 17-11. However, notice that if transistors are used as drivers, we must also use diodes to take care of inductive current generated when the coil is turned off. One reason that using the ULN2003 is preferable to the use of transistors as drivers is that the ULN2003 has an internal diode to take care of back EMF.

```
MyReg       SET   0x30            ;loc 30H for MyReg
R2          SET   0x20            ;loc 20H for R2 Reg
            CLRF  TRISB           ;Port B as output
            MOVLW 0x66            ;load step sequence
            MOVWF MyReg
BACK        MOVFF MyReg,PORTB     ;issue sequence to motor
            RRNCF MyReg,F         ;rotate right clockwise
            CALL DELAY            ;wait
            BRA   BACK            ;keep going
DELAY
            MOVLW 0xFF
            MOVWF R2
D1          NOP
            DECF R2,F
            BNZ D1
            RETURN
            END
```

Change the value of DELAY to set the speed of rotation.
We can use the single-bit instructions BSF and BCF instead of RRNCF to create the sequences.

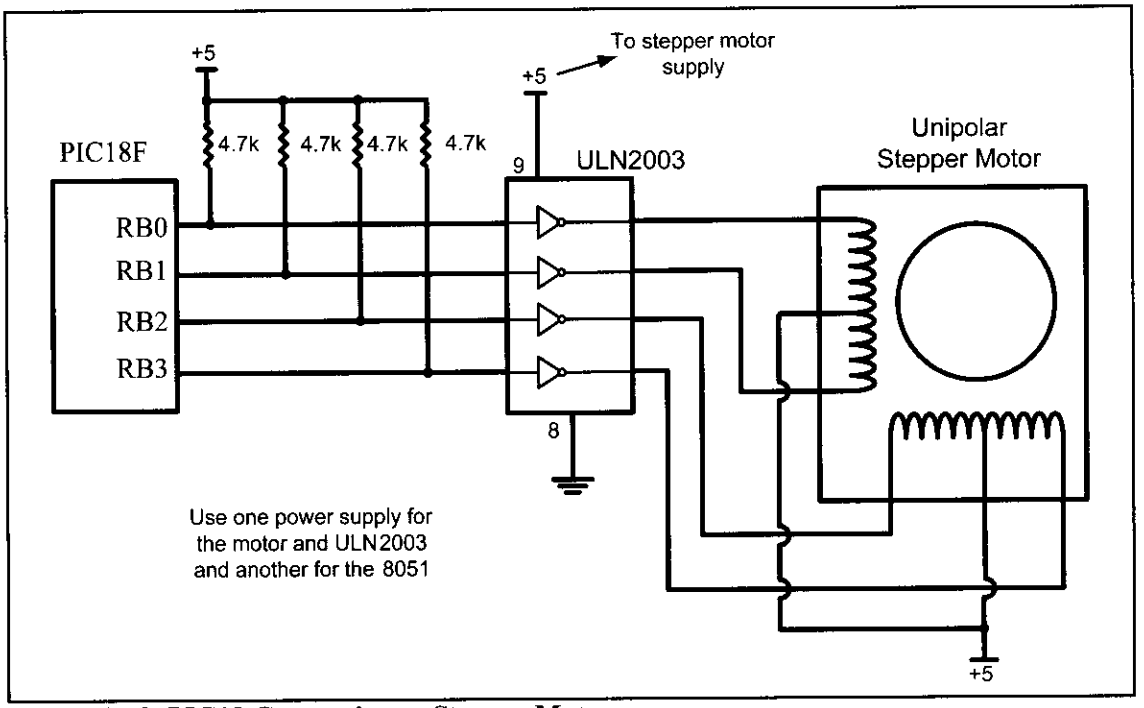


Figure 17-9. PIC18 Connection to Stepper Motor

Steps per second and rpm relation

The relation between rpm (revolutions per minute), steps per revolution, and steps per second is as follows.

$$Steps\ per\ second = \frac{rpm \times Steps\ per\ revolution}{60}$$

The 4-step sequence and number of teeth on rotor

The switching sequence shown earlier in Table 17-3 is called the 4-step switching sequence because after four steps the same two windings will be "ON". How much movement is associated with these four steps? After completing every four steps, the rotor moves only one tooth pitch. Therefore, in a stepper motor with 200 steps per revolution, the rotor has 50 teeth because $4 \times 50 = 200$ steps are needed to complete one revolution. This leads to the conclusion that the minimum step angle is always a function of the number of teeth on the rotor. In other words, the smaller the step angle, the more teeth the rotor passes. See Example 17-2.

Example 17-2

Give the number of times the four-step sequence in Table 17-3 must be applied to a stepper motor to make an 80-degree move if the motor has a 2-degree step angle.

Solution:

A motor with a 2-degree step angle has the following characteristics:

Step angle:	2 degrees	Steps per revolution:	180
Number of rotor teeth:	45	Movement per 4-step sequence:	8 degrees

To move the rotor 80 degrees, we need to send 10 consecutive 4-step sequences, because 10×4 steps $\times$ 2 degrees = 80 degrees.

Looking at Example 17-2, one might wonder what happens if we want to move 45 degrees, because the steps are 2 degrees each. To allow for finer resolutions, all stepper motors allow what is called an *8-step* switching sequence. The 8-step sequence is also called *half-stepping,* because in the 8-step sequence each step is half of the normal step angle. For example, a motor with a 2-degree step angle can be used as a 1-degree step angle if the sequence of Table 17-5 is applied.

Table 17-5: Half-Step 8-Step Sequence

Clockwise ↓

Step #	Winding A	Winding B	Winding C	Winding D
1	1	0	0	1
2	1	0	0	0
3	1	1	0	0
4	0	1	0	0
5	0	1	1	0
6	0	0	1	0
7	0	0	1	1
8	0	0	0	1

↑ Counter-clockwise

Motor speed

The motor speed, measured in steps per second (steps/s), is a function of the switching rate. Notice in Example 17-1 that by changing the length of the time delay loop, we can achieve various rotation speeds.

Holding torque

The following is a definition of holding torque: "With the motor shaft at standstill or zero rpm condition, the amount of torque, from an external source, required to break away the shaft from its holding position. This is measured with rated voltage and current applied to the motor." The unit of torque is ounce-inch (or kg-cm).

Wave drive 4-step sequence

In addition to the 8-step and the 4-step sequences discussed earlier, there is another sequence called the wave drive 4-step sequence. It is shown in Table 17-6. Notice that the 8-step sequence of Table 17-5 is simply the combination of the wave drive 4-step and normal 4-step normal sequences shown in Tables 17-6 and 17-3, respectively. Experimenting with the wave drive 4-step sequence is left to the reader.

Table 17-6: Wave Drive 4-Step Sequence

Clockwise ↓

Step #	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

↑ Counter-clockwise

Table 17-7: Selected Stepper Motor Characteristics (www.Jameco.com)

Part No.	Step Angle	Drive System	Volts	Phase Resistance	Current
151861CP	7.5	unipolar	5 V	9 ohms	550 mA
171601CP	3.6	unipolar	7 V	20 ohms	350 mA
164056CP	7.5	bipolar	5 V	6 ohms	800 mA

Unipolar versus bipolar stepper motor interface

There are three common types of stepper motor interfacing: universal, unipolar, and bipolar. They can be identified by the number of connections to the motor. A universal stepper motor has eight, while the unipolar has six and the bipolar has four. The universal stepper motor can be configured for all three modes, while the unipolar can be either unipolar or bipolar. Obviously the bipolar cannot be configured for universal nor unipolar mode. Table 17-7 shows selected stepper motor characteristics. Figure 17-10 shows the basic internal connections of all three type of configurations.

Unipolar stepper motors can be controlled using the basic interfacing shown in Figure 17-11, whereas the bipolar stepper requires H-Bridge circuitry. Bipolar stepper motors require a higher operational current than the unipolar; the advantage of this is a higher holding torque.

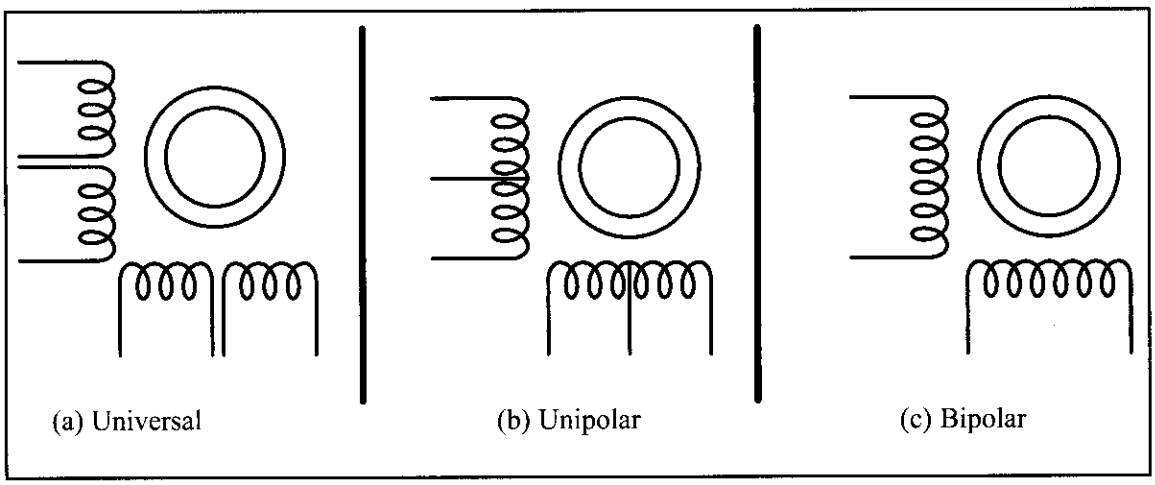


Figure 17-10. Common Stepper Motor Types

Using transistors as drivers

Figure 17-11 shows an interface to a unipolar stepper motor using transistors. Diodes are used to reduce the back EMF spike created when the coils are energized and de-energized, similar to the electromechanical relays discussed earlier. TIP transistors can be used to supply higher current to the motor. Table 17-8 lists the common industrial Darlington transistors. These transistors can accommodate higher voltages and currents.

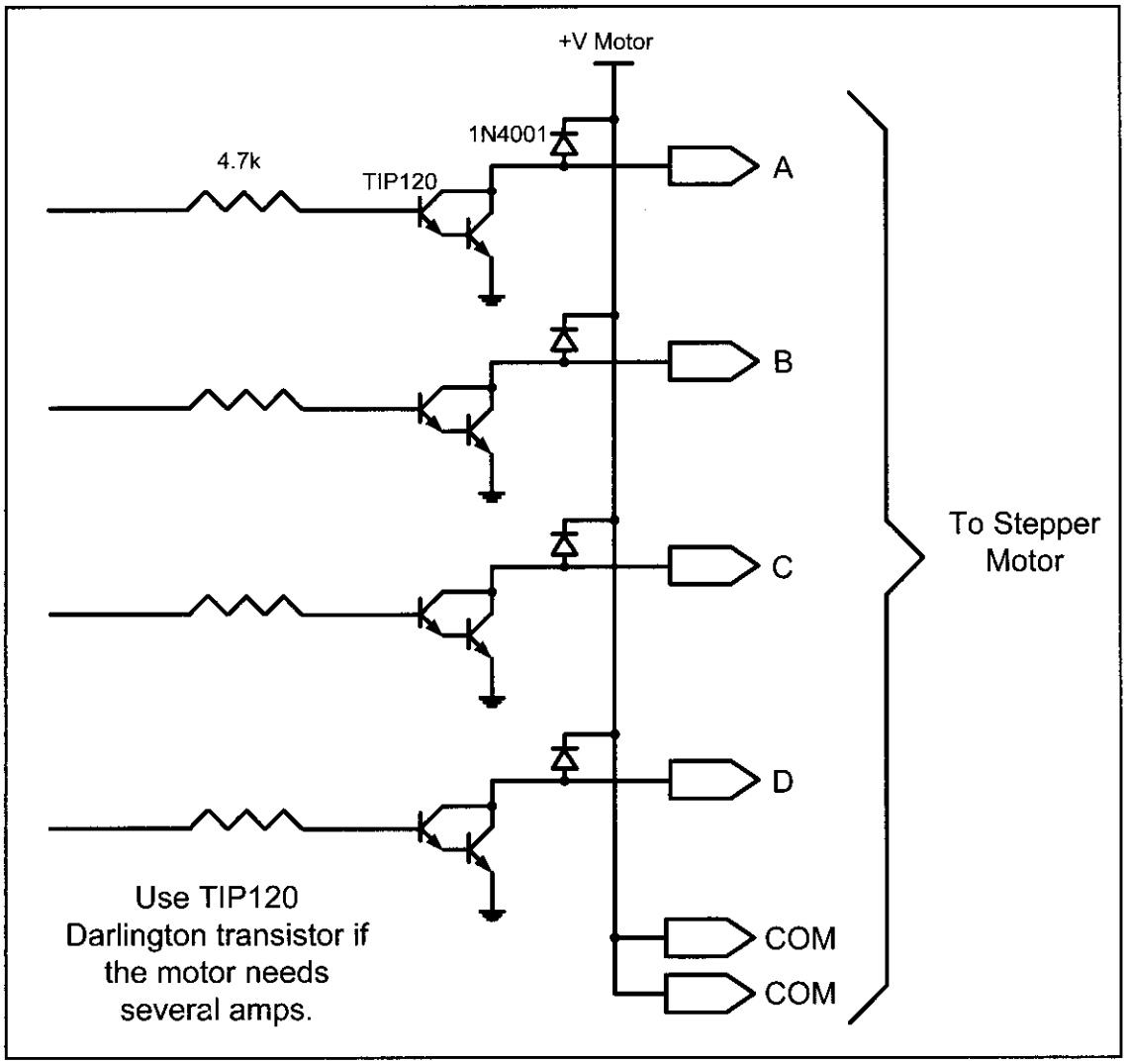


Figure 17-11. Using Transistors for Stepper Motor Driver

Table 17-8: Darlington Transistor Listing

NPN	PNP	Vceo (volts)	Ic (amps)	hfe (common)
TIP110	TIP115	60	2	1000
TIP111	TIP116	80	2	1000
TIP112	TIP117	100	2	1000
TIP120	TIP125	60	5	1000
TIP121	TIP126	80	5	1000
TIP122	TIP127	100	5	1000
TIP140	TIP145	60	10	1000
TIP141	TIP146	80	10	1000
TIP142	TIP147	100	10	1000

Controlling stepper motor via optoisolator

In the first section of this chapter we examined the optoisolator and its use. Optoisolators are widely used to isolate the stepper motor's EMF voltage and keep it from damaging the digital/microcontroller system. This is shown in Figure 17-12. See Examples 17-3 and 17-4.

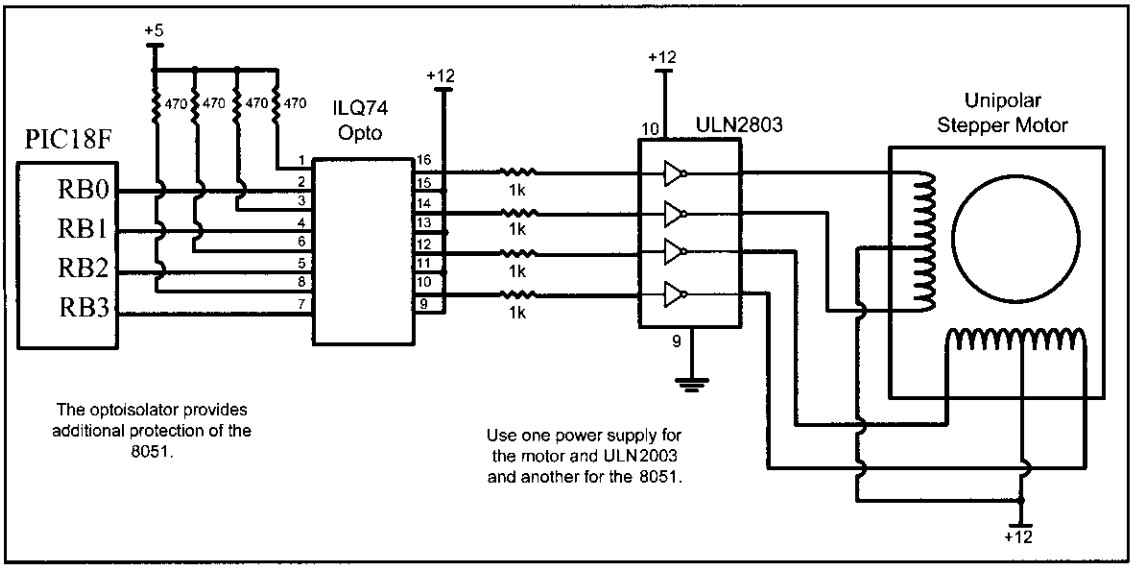


Figure 17-12. Controlling Stepper Motor via Optoisolator

Example 17-3

A switch is connected to pin RD7 (PORTD.7). Write a program to monitor the status of SW and perform the following:
(a) If SW = 0, the stepper motor moves clockwise.
(b) If SW = 1, the stepper motor moves counterclockwise.

Solution:

```
MyReg     SET  0x30            ;loc 30H for MyReg
          BSF  TRISD,RD7       ;RD7 as input pin
          CLRF TRISB           ;Port B as output
          MOVLW 0x66           ;load step sequence
          MOVWF MyReg
BACK      BTFSS PORTD,RD7      ;check the SW
          BRA  OVER       ;It is high. Make it clockwise
          MOVFF MyReg,PORTB    ;issue sequence to motor
          RRNCF MyReg,F        ;rotate right clockwise
          CALL DELAY           ;wait
          BRA  BACK            ;keep going
OVER      MOVFF MyReg,PORTB    ;issue sequence to motor
          RLNCF MyReg,F        ;rotate left clockwise
          CALL DELAY           ;wait
          BRA  BACK            ;keep going
```

Stepper motor control with PIC18 C

The PIC18 C version of the stepper motor control is given below. In this program we could have used << (shift left) and >> (shift right) as was shown in Chapter 7.

```
#include <p18f458.h>
void main()
  {
    TRISB=0x0;          //PORTB as output
    while(1)
      {
        PORTB = 0x66;
        MSDelay(100);
        PORTB = 0xCC;
        MSDelay(100);
        PORTB = 0x99;
        MSDelay(100);
        PORTB = 0x33;
        MSDelay(100);
      }
  }
```

Example 17-4

A switch is connected to pin RD7. Write a C program to monitor the status of SW and perform the following:
(a) If SW = 0, the stepper motor moves clockwise.
(b) If SW = 1, the stepper motor moves counterclockwise.

Solution:

```
#include <p18f458.h>
#define SW PORTDbits.RD7
void MSDelay(int ms);
void main()
  {
    TRISD=0x80;         //RD7 as input pin
    TRISB=0x0;          //PORTB as output
    while(1)
      {
        if(SW == 0)
          {
          PORTB = 0x66;
          MSDelay(100);
          PORTB = 0xCC;
          MSDelay(100);
          PORTB = 0x99;
          MSDelay(100);
          PORTB = 0x33;
          MSDelay(100);
          }
        else
          {
          PORTB = 0x66;
          MSDelay(100);
          PORTB = 0x33;
```

Example 17-4 Cont.

```
            MSDelay(100);
            PORTB = 0x99;
            MSDelay(100);
            PORTB = 0xCC;
            MSDelay(100);
            }
        }
    }

void MSDelay(unsigned int value)
    {
      unsigned int x, y;
      for(x=0;x<1275;x++)
        for(y=0;y<value;y++);
    }
```

Review Questions

1. Give the 4-step sequence of a stepper motor if we start with 0110.
2. A stepper motor with a step angle of 5 degrees has ____ steps per revolution.
3. Why do we put a driver between the microcontroller and the stepper motor?

SECTION 17.3: DC MOTOR INTERFACING AND PWM

This section begins with an overview of the basic operation of DC motors. Then we describe how to interface a DC motor to the PIC18. Finally, we use Assembly and C language programs to demonstrate the concept of pulse width modulation (PWM) and show how to control the speed and direction of a DC motor.

DC motors

A direct current (DC) motor is another widely used device that translates electrical pulses into mechanical movement. In the DC motor we have only + and – leads. Connecting them to a DC voltage source moves the motor in one direction. By reversing the polarity, the DC motor will move in the opposite direction. One can easily experiment with the DC motor. For example, small fans used in many motherboards to cool the CPU are run by DC motors. By connecting their leads to the + and – voltage source, the DC motor moves. While a stepper motor moves in steps of 1 to 15 degrees, the DC motor moves continuously. In a stepper motor, if we know the starting position we can easily count the number of steps the motor has moved and calculate the final position of the motor. This is not possible in a DC motor. The maximum speed of a DC motor is indicated in rpm and is given in the data sheet. The DC motor has two rpms: no-load and loaded. The manufacturer's data sheet gives the no-load rpm. The no-load rpm can be from a few thousand to tens of thousands. The rpm is reduced when moving a load and it decreases as the load is increased. For example, a drill turning a screw has a much lower rpm speed than when it is in the no-load situation. DC motors also have voltage and current ratings. The nominal voltage is the voltage for that motor under normal conditions, and can be from 1 to 150 V, depending on the motor. As we increase the voltage, the rpm goes up. The current rating is the current consump-

tion when the nominal voltage is applied with no load, and can be from 25 mA to a few amps. As the load increases, the rpm is decreased, unless the current or voltage provided to the motor is increased, which in turn increases the torque. With a fixed voltage, as the load increases, the current (power) consumption of a DC motor is increased. If we overload the motor it will stall, and that can damage the motor due to the heat generated by high current consumption.

Unidirectional control

Figure 17-13 shows the DC motor rotation for clockwise (CW) and counterclockwise (CCW) rotations. See Table 17-9 for selected DC motors.

Table 17-9: Selected DC Motor Characteristics (www.Jameco.com)

Part No.	Nominal Volts	Volt Range	Current	RPM	Torque
154915CP	3 V	1.5–3 V	0.070 A	5,200	4.0 g-cm
154923CP	3 V	1.5–3 V	0.240 A	16,000	8.3 g-cm
177498CP	4.5 V	3–14 V	0.150 A	10,300	33.3 g-cm
181411CP	5 V	3–14 V	0.470 A	10,000	18.8 g-cm

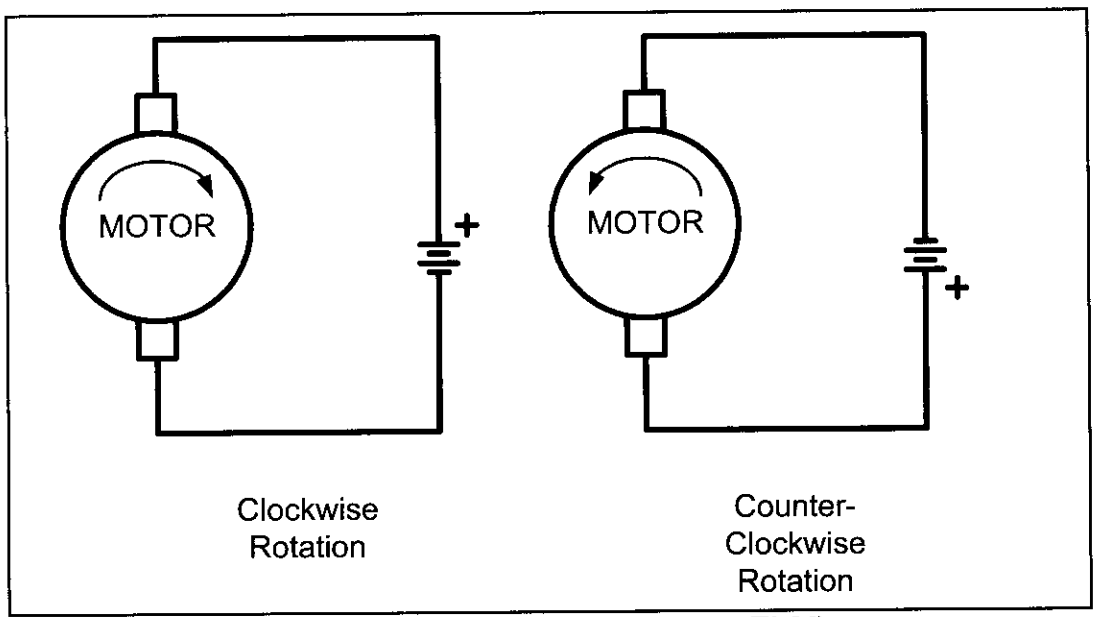


Figure 17-13. DC Motor Rotation (Permanent Magnet Field)

Bidirectional control

With the help of relays or some specially designed chips we can change the direction of the DC motor rotation. Figures 17-14 through 17-17 show the basic concepts of H-Bridge control of DC motors.

Figure 17-14 shows the connection of an H-Bridge using simple switches. All the switches are open, which does not allow the motor to turn.

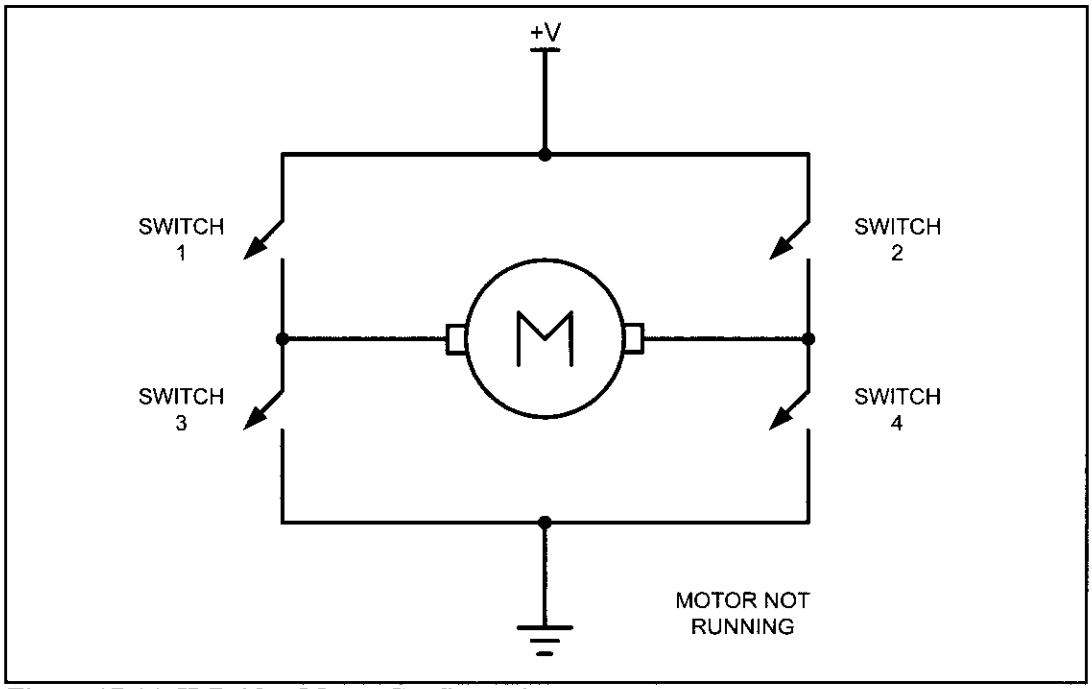


Figure 17-14. H-Bridge Motor Configuration

Figure 17-15 shows the switch configuration for turning the motor in one direction. When switches 1 and 4 are closed, current is allowed to pass through the motor.

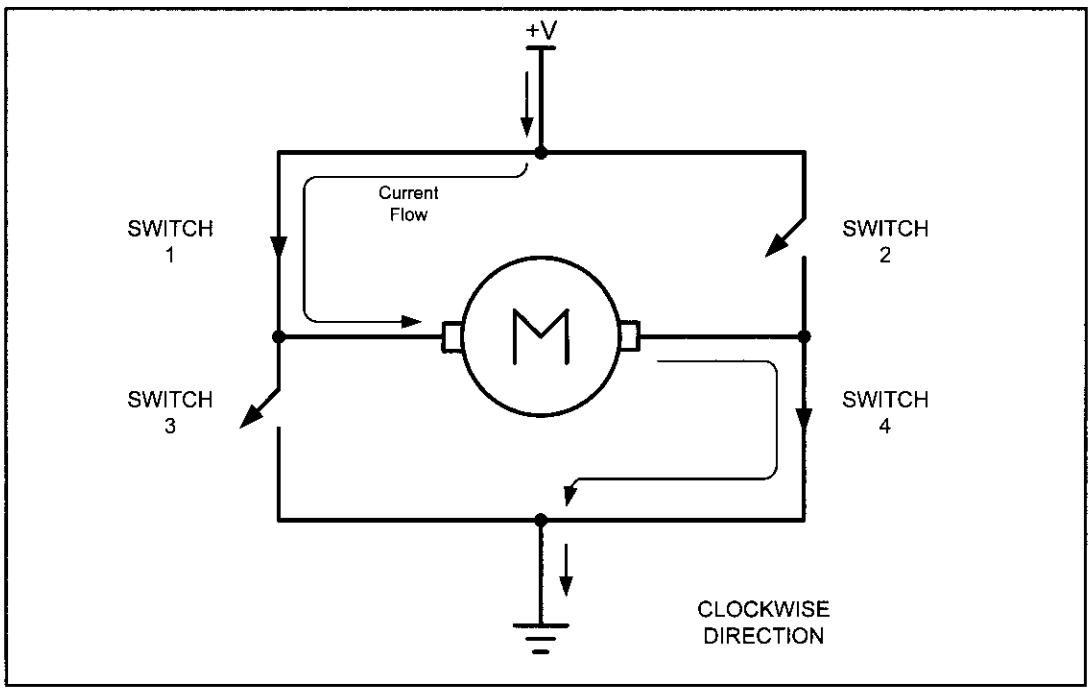


Figure 17-15. H-Bridge Motor Clockwise Configuration

Figure 17-16 shows the switch configuration for turning the motor in the opposite direction from the configuration of Figure 17-15. When switches 2 and 3 are closed, current is allowed to pass through the motor.

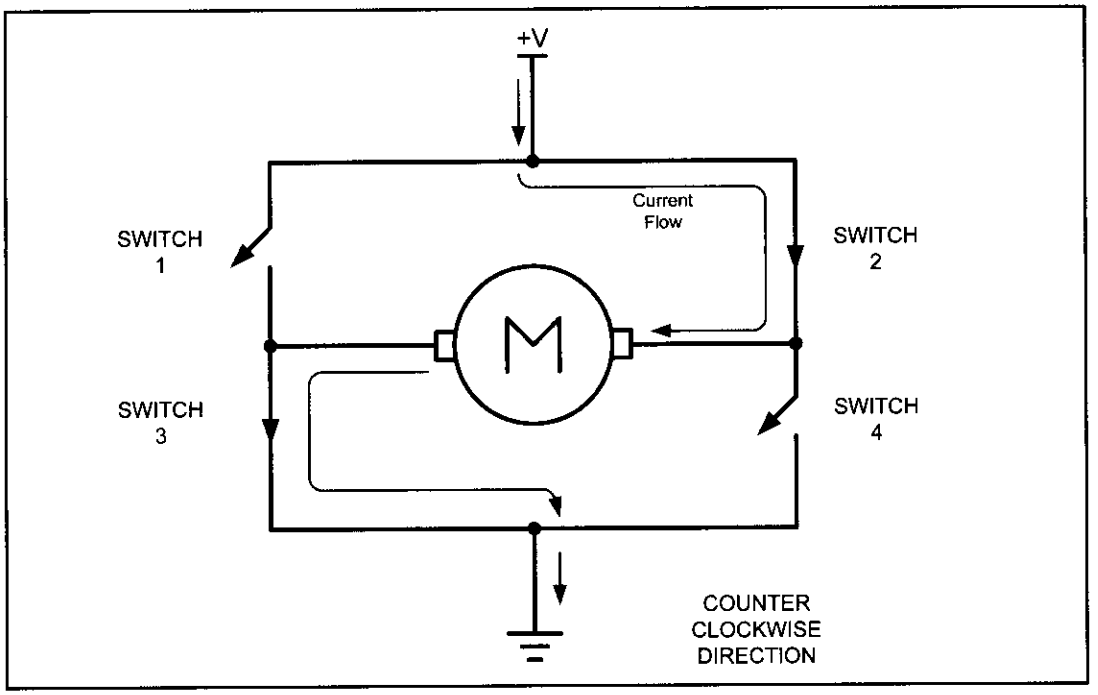


Figure 17-16. H-Bridge Motor Counterclockwise Configuration

Figure 17-17 shows an invalid configuration. Current flows directly to ground, creating a short circuit. The same effect occurs when switches 1 and 3 are closed or switches 2 and 4 are closed.

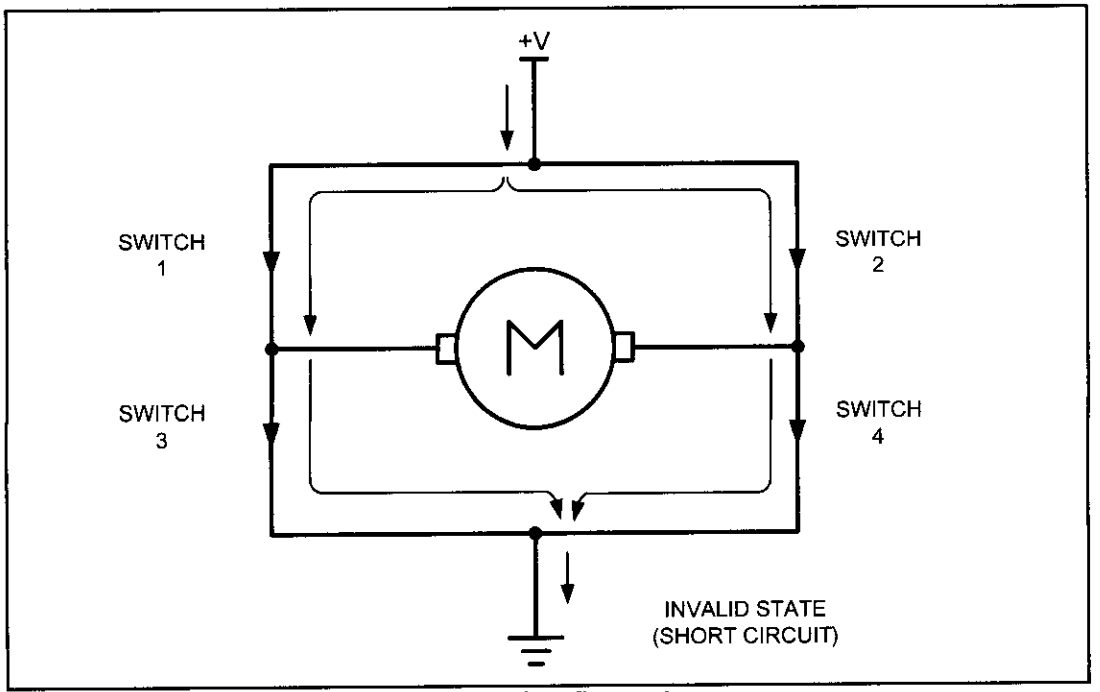


Figure 17-17. H-Bridge in an Invalid Configuration

Table 17-10 shows some of the logic configurations for the H-Bridge design.

H-Bridge control can be created using relays, transistors, or a single IC solution such as the L293. When using relays and transistors, you must ensure that invalid configurations do not occur.

Table 17-10: Some H-Bridge Logic Configurations for Figure 17-14

Motor Operation	SW1	SW2	SW3	SW4
Off	Open	Open	Open	Open
Clockwise	Closed	Open	Open	Closed
Counterclockwise	Open	Closed	Closed	Open
Invalid	Closed	Closed	Closed	Closed

Although we do not show the relay control of an H-Bridge, Example 17-5 shows a simple program to operate a basic H-Bridge.

Example 17-5

A switch is connected to pin RD7 (PORTD.7). Using a simulator, write a program to simulate the H-Bridge in Table 17-10. We must perform the following:
(a) If DIR = 0, the DC motor moves clockwise.
(b) If DIR = 1, the DC motor moves counterclockwise.

Solution:

```
      BCF TRISB,0              ;PORTB.0 as output for switch 1
      BCF TRISB,1              ;      .1   "          switch 2
      BCF TRISB,2              ;      .2   "          switch 3
      BCF TRISB,3              ;      .3   "          switch 4
      BSF TRISD,7              ;make PORTD.7 an input DIR
MONITOR:
      BTFSS PORTD,7
      BRA CLOCKWISE
      BSF PORTB,0              ;switch 1
      BCF PORTB,1              ;switch 2
      BCF PORTB,2              ;switch 3
      BSF PORTB,3              ;switch 4
      BRA MONITOR
CLOCKWISE:
      BCF PORTB,0              ;switch 1
      BSF PORTB,1              ;switch 2
      BSF PORTB,2              ;switch 3
      BCF PORTB,3              ;switch 4
      BRA MONITOR
```

View the results on your simulator. This example is for simulation only and should not be used on a connected system.

See http://www.MicroDigitalEd.com for additional information on using H-Bridges.

Figure 17-18 shows the connection of the L293 to an PIC18. Be aware that the L293 will generate heat during operation. For sustained operation of the motor, use a heat sink. Example 17-6 shows control of the L293.

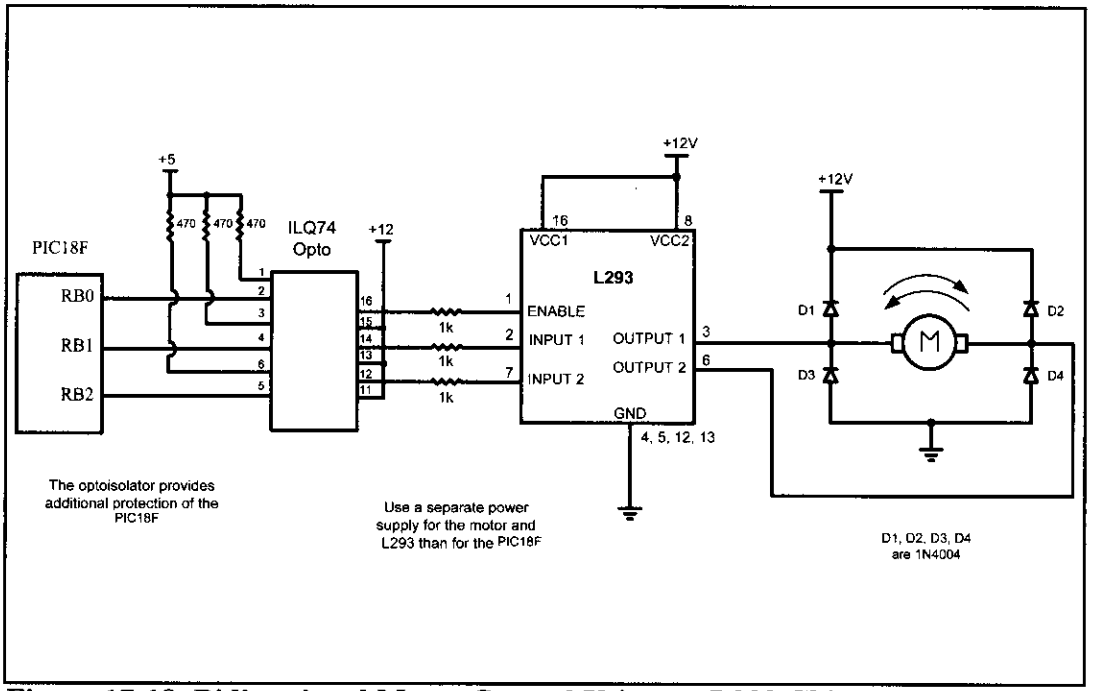


Figure 17-18. Bidirectional Motor Control Using an L293 Chip

Example 17-6

Figure 17-18 shows the connection of an L293. Add a switch to pin RD7 (PORTD.7). Write a program to monitor the status of SW and perform the following:

(a) If SW = 0, the DC motor moves clockwise.
(b) If SW = 1, the DC motor moves counterclockwise.

Solution:

```
      BCF   TRISB,0
      BCF   TRISB,1
      BCF   TRISB,2
      BSF   TRISD,7
      BSF   PORTB,0          ;enable the chip
CHK   BTFSS PORTD,7
      BRA   CWISE
      BCF   PORTB,1          ;turn the motor counterclockwise
      BSF   PORTB,2
      BRA   CHK
CWISE BSF   PORTB,1
      BCF   PORTB,2          ;turn motor clockwise
      BRA   CHK
```

Pulse width modulation (PWM)

The speed of the motor depends on three factors: (a) load, (b) voltage, and (c) current. For a given fixed load we can maintain a steady speed by using a method called *pulse width modulation* (PWM). By changing (modulating) the width of the pulse applied to the DC motor we can increase or decrease the amount of power provided to the motor, thereby increasing or decreasing the motor speed. Notice that, although the voltage has a fixed amplitude, it has a variable duty cycle. That means the wider the pulse, the higher the speed. PWM is so widely used in DC motor control that some microcontrollers come with the PWM circuitry embedded in the chip. In such microcontrollers all we have to do is load the proper registers with the values of the high and low portions of the desired pulse, and the rest is taken care of by the microcontroller. This allows the microcontroller to do other things. For microcontrollers without PWM circuitry, we must create the various duty cycle pulses using software, which prevents the microcontroller from doing other things. The ability to control the speed of the DC motor using PWM is one reason that DC motors are preferable over AC motors. AC motor speed is dictated by the AC frequency of the voltage applied to the motor and the frequency is generally fixed. As a result, we cannot control the speed of the AC motor when the load is increased. As was shown earlier, we can also change the DC motor's direction and torque. See Figure 17-19 for PWM comparisons.

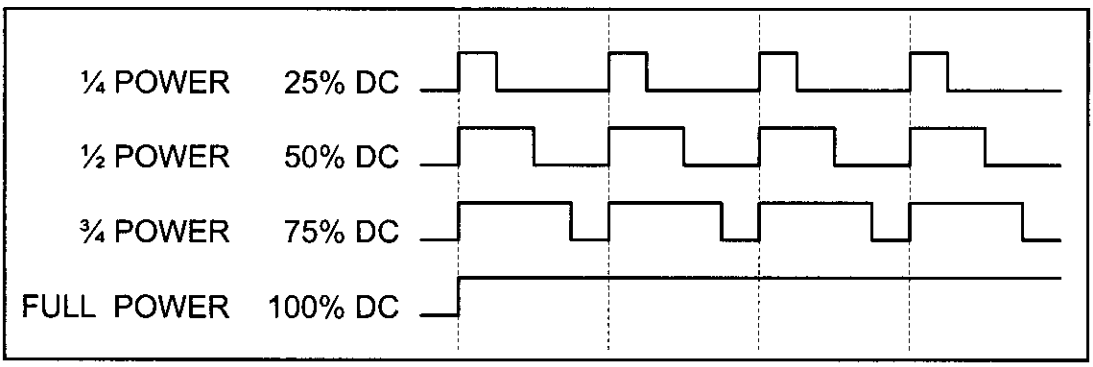


Figure 17-19. Pulse Width Modulation Comparison

DC motor control with optoisolator

As we discussed in the first section of this chapter, the optoisolator is indispensable in many motor control applications. Figures 17-20 and 17-21 show the connections to a simple DC motor using a bipolar and a MOSFET transistor. Notice that the PIC18 is protected from EMI created by motor brushes by using an optoisolator and a separate power supply.

Figures 17-20 and 17-21 show optoisolators for control of single directional motor control, and the same principle should be used for most motor applications. Separating the power supplies of the motor and logic will reduce the possibility of damage to the control circuity.

Figure 17-20 shows the connection of a bipolar transistor to a motor. Protection of the control circuit is provided by the optoisolator. The motor and PIC18 use separate power supplies. The separation of power supplies also allows the use of high-voltage motors. Notice that we use a decoupling capacitor across the motor; this helps reduce the EMI created by the motor. The motor is switched on by clearing bit P1.0.

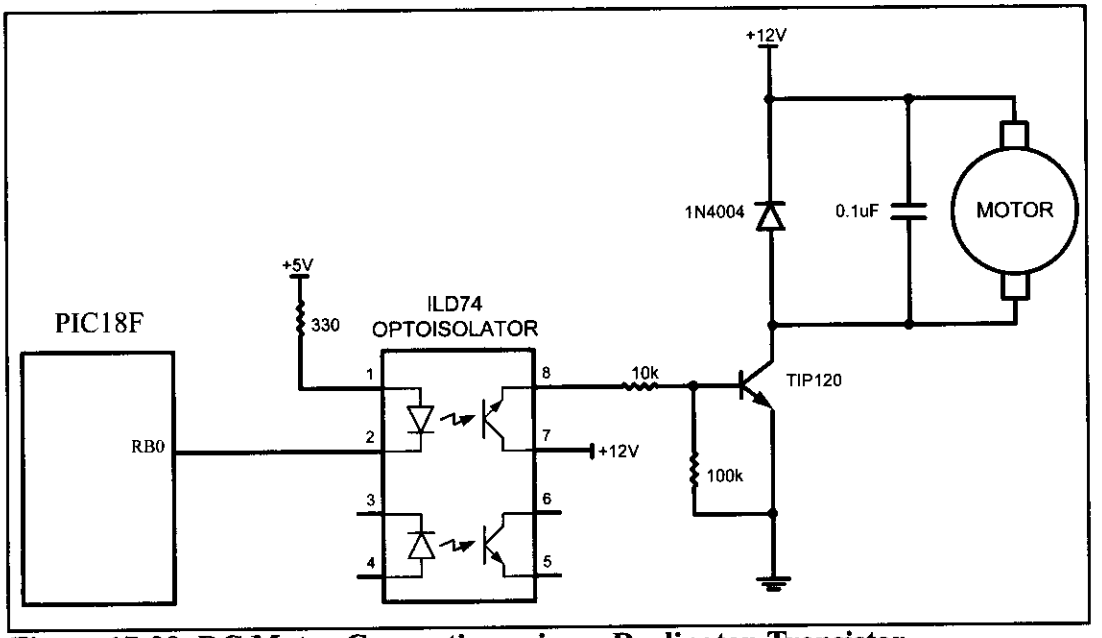


Figure 17-20. DC Motor Connection using a Darlington Transistor

Figure 17-21 shows the connection of a MOSFET transistor. The optoisolator protects the PIC18 from EMI. The zener diode is required for the transistor to reduce gate voltage below the rated maximum value. See Example 17-7.

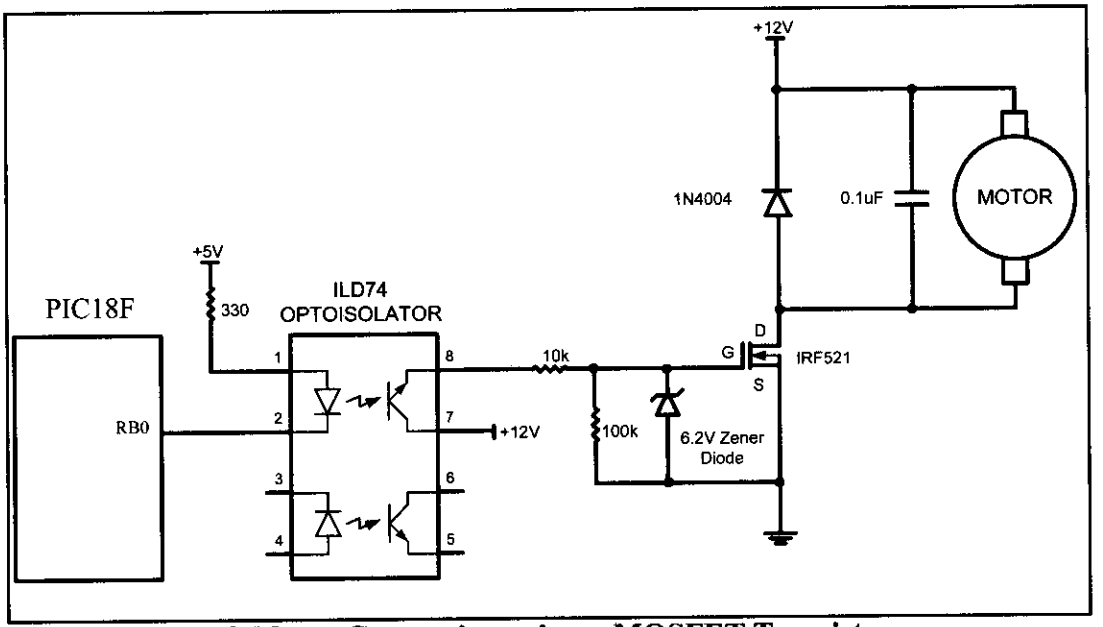


Figure 17-21. DC Motor Connection using a MOSFET Transistor

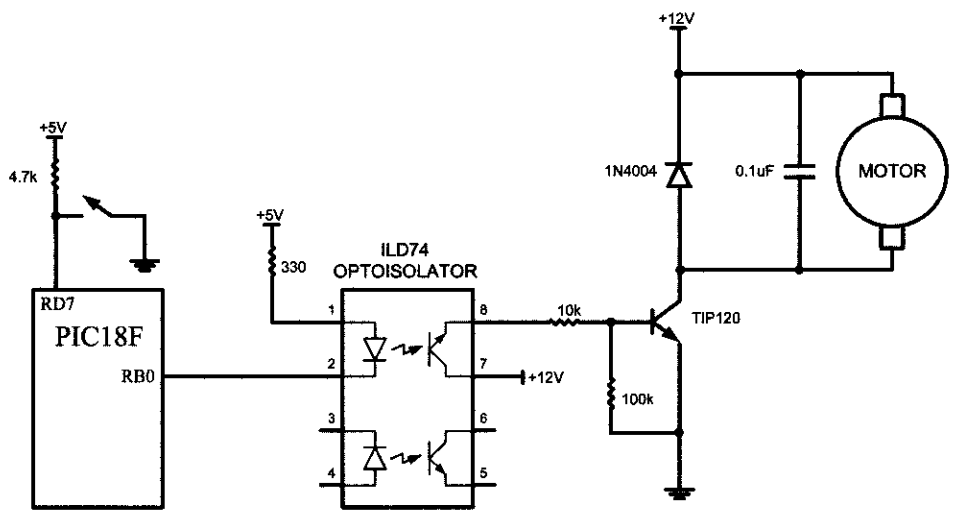
Example 17-7

Refer to the figure in this example. Write a program to monitor the status of the switch and perform the following:

(a) If PORTD.7 = 1, the DC motor moves with 25% duty cycle pulse.
(b) If PORTD.7 = 0, the DC motor moves with 50% duty cycle pulse.

Solution:

```
            BCF TRISB,RB0          ;PORTB.0 as output
            BSF TRISD,RD7          ;PORTD.7 as input
            BCF PORTB,RB0          ;turn off motor
CHK
            BTFSS PORTD,RD7
            BRA   PWM_50
            BSF PORTB,RB0          ;high portion of pulse
            CALL DELAY
            BCF PORTB,RB0          ;low portion of pulse
            CALL DELAY
            CALL DELAY
            CALL DELAY
            BRA   CHK
PWM_50
            BSF PORTB,RB0          ;high portion of pulse
            CALL DELAY
            CALL DELAY
            BCF PORTB,RB0          ;low portion of pulse
            CALL DELAY
            CALL DELAY
            BRA CHK
```



DC motor control and PWM using C

Examples 17-8 through 17-10 show the PIC18 C version of the earlier programs controlling the DC motor.

Example 17-8

Refer to Figure 17-18 for connection of the motor. A switch is connected to pin RD7. Write a C program to monitor the status of SW and perform the following:

(a) If SW = 0, the DC motor moves clockwise.
(b) If SW = 1, the DC motor moves counterclockwise.

Solution:

```
#include <p18f458.h>

#define SW PORTDbits.RD7
#define ENABLE PORTBbits.RB0
#define MTR_1 PORTBbits.RB1
#define MTR_2 PORTBbits.RB2

void main()
  {
  TRISD=0x80;    //make RD7 input pin
  TRISB=0x0;     //make PORTB output
  SW = 1;
  ENABLE = 0;
  MTR_1 = 0;
  MTR_2 = 0;

  while(1)
    {
      ENABLE = 1;
      if(SW == 1)
        {
          MTR_1 = 1;
          MTR_2 = 0;
        }
      else
        {
          MTR_1 = 0;
          MTR_2 = 1;
        }
    }
  }
```

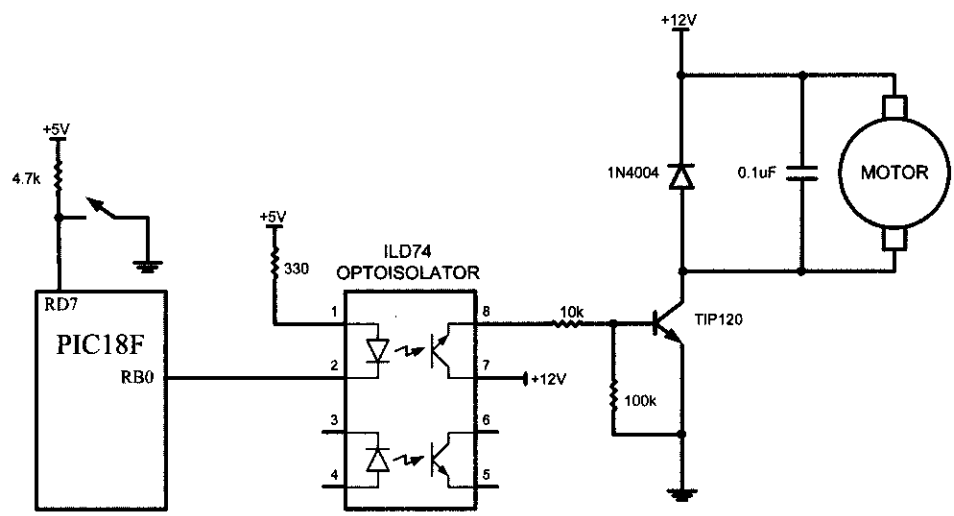
Example 17-9

Refer to the figure in this example. Write a C program to monitor the status of SW and perform the following:

(a) If SW = 0, the DC motor moves with 50% duty cycle pulse.
(b) If SW = 1, the DC motor moves with 25% duty cycle pulse.

Solution:

```
#include <p18f458.h>
#define SW PORTDbits.RD7
#define MTR  PORTBbits.RB1
void MSDelay(unsigned int value);
void main()
  {
  TRISD=0x80;          //make RD7 input pin
  TRISB=0xFD;          //make RB1 output pin
  while(1)
    {
      if(SW == 1)
        {
          MTR = 1;
          MSDelay(25);
          MTR = 0;
          MSDelay(75);
        }
      else
        {
          MTR = 1;
          MSDelay(50);
          MTR = 0;
          MSDelay(50);
        }
    }
  }
void MSDelay(unsigned int value)
  {
    unsigned char x, y;
    for(x=0; x<1275; x++)
      for(y=0; y<value; y++);
  }
```



Example 17-10

Refer to Figure 17-20 for connection to the motor. Two switches are connected to pins RD0 and RD1. Write a C program to monitor the status of both switches and perform the following:

SW2 (RD1)	SW1 (RD0)	
0	0	DC motor moves slowly (25% duty cycle).
0	1	DC motor moves moderately (50% duty cycle).
1	0	DC motor moves fast (75% duty cycle).
1	1	DC motor moves very fast (100% duty cycle).

Solution:

```
#include <p18f458.h>
#define MTR PORTBbits.RB1
void MSDelay(unsigned int value);

void main()
  {
  unsigned int duty;
  TRISB = 0xFD;
  TRISD = 0xFF;
  while(1)
    {
     duty = PORTD&0x03;
     duty++;
     duty *= 25;
     MTR = 1;
     MSDelay(duty);
     MTR = 0;
     MSDelay(100-duty);
    }
  }
```

Review Questions

1. True or false. The permanent magnet field DC motor has only two leads for + and – voltages.
2. True or false. Just like a stepper motor, one can control the exact angle of a DC motor's move.
3. Why do we put a driver between the microcontroller and the DC motor?
4. How do we change a DC motor's rotation direction?
5. What is stall in a DC motor?
6. True or false. PWM allows the control of a DC motor with the same phase, but different amplitude pulses.
7. The RPM rating given for the DC motor is for _____ (no-load, loaded).

SECTION 17.4: PWM MOTOR CONTROL WITH CCP

We examined the CCP (Compare Capture Pulse-Width-Modulation) part of the PIC452/458 in Chapter 15. One of the features of the CCP is the pulse width modulation (PWM) as we saw in Section 15.4 of Chapter 15. In this section we use the PWM feature of the CCP to control DC motors. Review the programming of the PWM in Section 15.4 before embarking on this section.

DC motor control with CCP

Recall from Section 15.4 that the PWM part of the CCP is programmed by using the PR2 and Timer2 registers. Program 17-2 is the rewrite of Example 17-7 using the PWM feature of the CCP1. Notice that Program 17-2 is the modified version of Program 15-5 in Chapter 15. Program 17-2C is the C version of Program 17-2. In Program 17-2 (and 17-2C), an input switch is being monitored. If the switch is low, the PIC18 creates a 50% duty cycle PWM using the CCP1 module. If the switch is high, a 25% duty cycle PWM is created. Recall from Chapter 15 that we must use PR2 and Timer2 registers for creating PWM pulses.

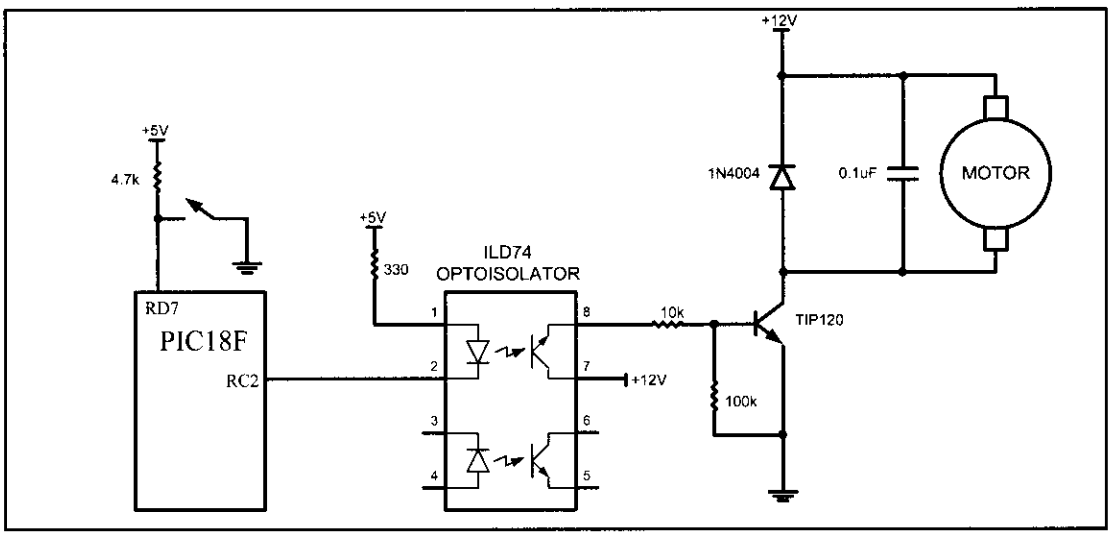


Figure 17-22: DC Motor Control Using CCP1 Pin

```
;Program 17-2
      BCF   TRISC,CCP1        ;make PWM output pin
      BSF   TRISD,RD7         ;make RD7 input pin
      MOVLW 0x3C              ;PWM MODE, 11 for DC1B1:B0
      MOVWF CCP1CON
      MOVLW D'100'            ;set period to 100 * Fosc/4
      MOVWF PR2
      MOVLW 0x01        ;Timer2, 4 prescale, no postscaler
      MOVWF T2CON
AGAIN BTFSS PORTD,RD7         ;Is the switch high?
      BRA T2DUTY              ;no, then 50%
      MOVLW D'25'             ;25% duty cycle
      BRA LOAD
T2DUTY MOVLW D'50'            ;50% duty cycle
```

```
      BRA LOAD
LOAD  MOVWF CCPR1L          ;load duty cycle
      CLRF  TMR2            ;clear Timer2
      BSF   T2CON,TMR2ON    ;turn on Timer2
      BCF   PIR1,TMR2IF     ;clear Timer2 flag
OVER  BTFSS PIR1,TMR2IF     ;wait for end of period
      BRA   OVER
      GOTO  AGAIN           ;continue
```

The following is the C version of the above program.

```
//Program 17-2C
#include <p18f458.h>
void main()
  {
  TRISC = 0xFB;             //make CCP1 output pin
  TRISD = 0x80;             //make RD7 input pin
  CCP1CON = 0x3C;           //PWM MODE, 11 for DC1B1:B0
  PR2=100;                  //set period to 100 * 16/Fosc
  T2CON=0x01;               //4 prescaler, no postscaler
  while(1)
    {
    if(PORTDbits.RD7==1)
      CCPR1L = 25;          //25% duty cycle
    else
      CCPR1L = 50;          //50% duty cycle
    TMR2=0x0;               //clear Timer2
    PIR1bits.TMR2IF=0;      //clear Timer2 flag
    T2CONbits.TMR2ON=1;     //start Timer2
    while(PIR1bits.TMR2IF==0);//wait for end of period
    }
  }
```

Review Questions

1. True or false. For standard CCP1, we use the RC2 pin for PWM.
2. True or false. For standard CCP1, the CCP1 pin must be configured as output.
3. In standard CCP1, we use _____ to set the period for PWM.
4. In standard CCP1, we use _____ to set the duty cycle for PWM.
5. True or false. In standard CCP1, we must use Timer1 for PWM.

SECTION 17.5: DC MOTOR CONTROL WITH ECCP

The PIC18F452/458 (or 4520/4580) comes with one standard CCP and one enhanced CCP (ECCP). Indeed, in recent years the CCP module has been de-emphasized while the ECCP is becoming more prominent in the PIC18 family. The reason is that ECCP allows the implementation of the H-Bridge for bidirectional control of the DC motor in addition to the capture/compare mode present in the standard CCP. In this section, we use the ECCP feature of the PIC18 to control the DC motor. Before embarking on this section, the basic concept of ECCP programming in Chapter 15 needs to be reviewed.

Bidirectional DC motor control with ECCP

ECCP allows the implementation of the H-Bridge for bidirectional movement of the DC motor because it uses 4 pins instead of a single pin as is used in standard CCP. As we saw in Section 17.3 of this chapter, the bidirectional DC movement needs some kind of H-Bridge circuitry. The ECCP module of the PIC18 implements the entire H-Bridge circuitry internally. It uses RD7–RD4 (PORTD.7–PORTD.4) for this purpose as shown in Figures 17-23 through 17-26.

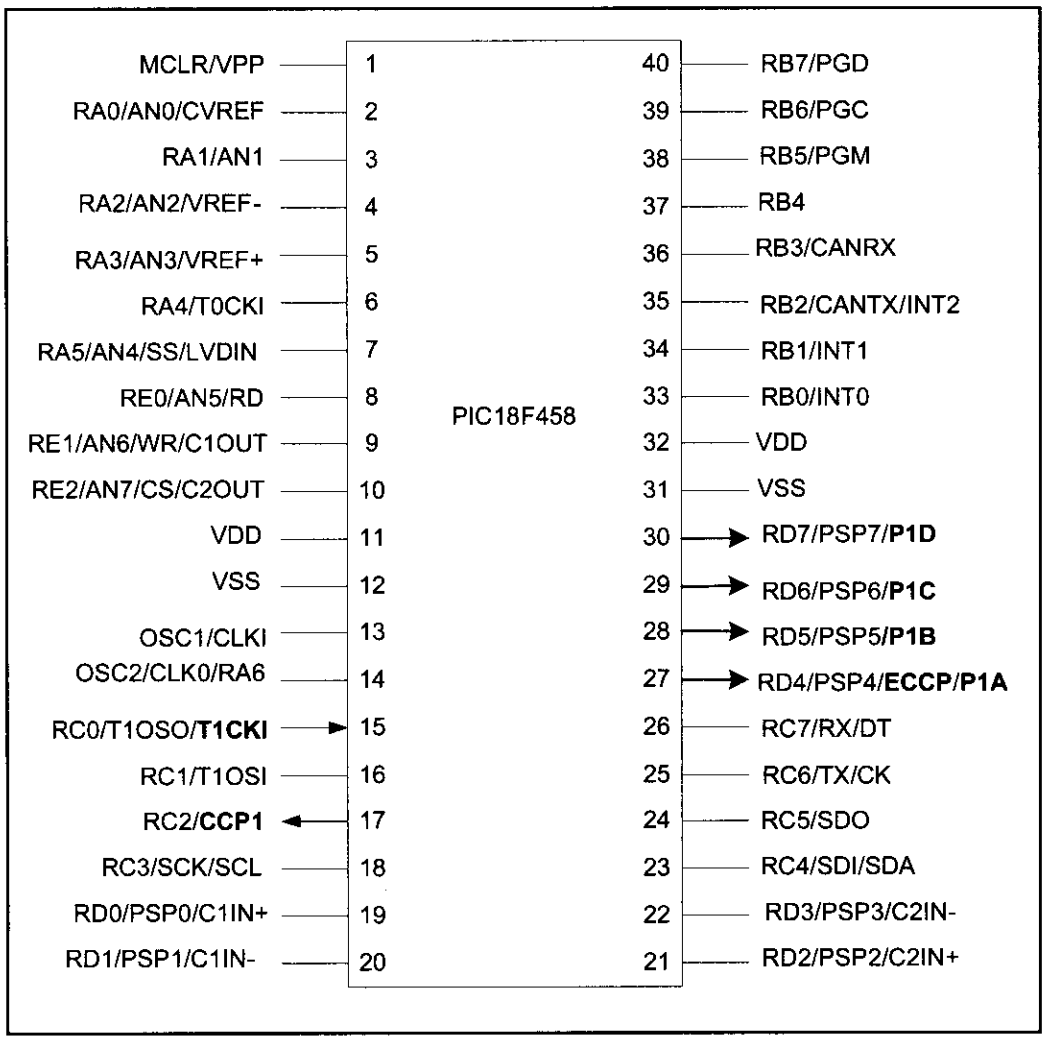


Figure 17-23. ECCP Pins for PWM in PIC18F458/4580 (452/4520)

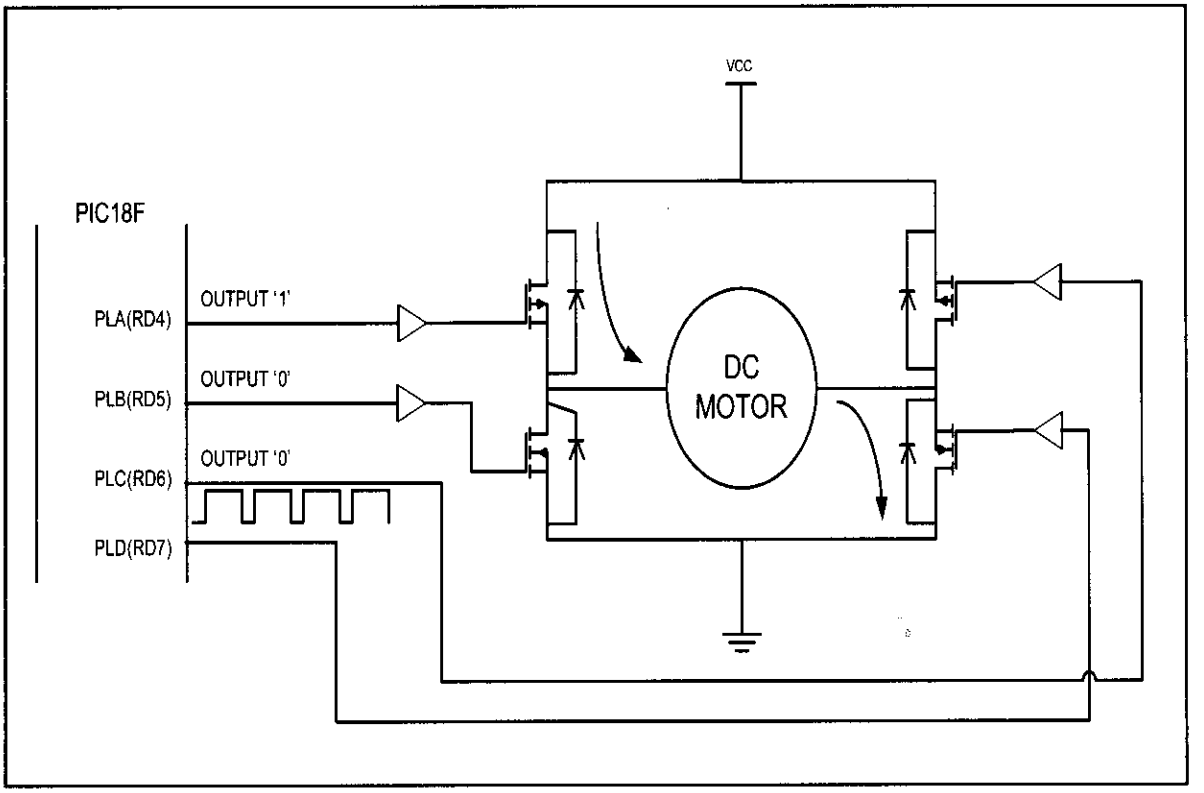


Figure 17-24. Forward Current Flow Using ECCP (from Microchip)

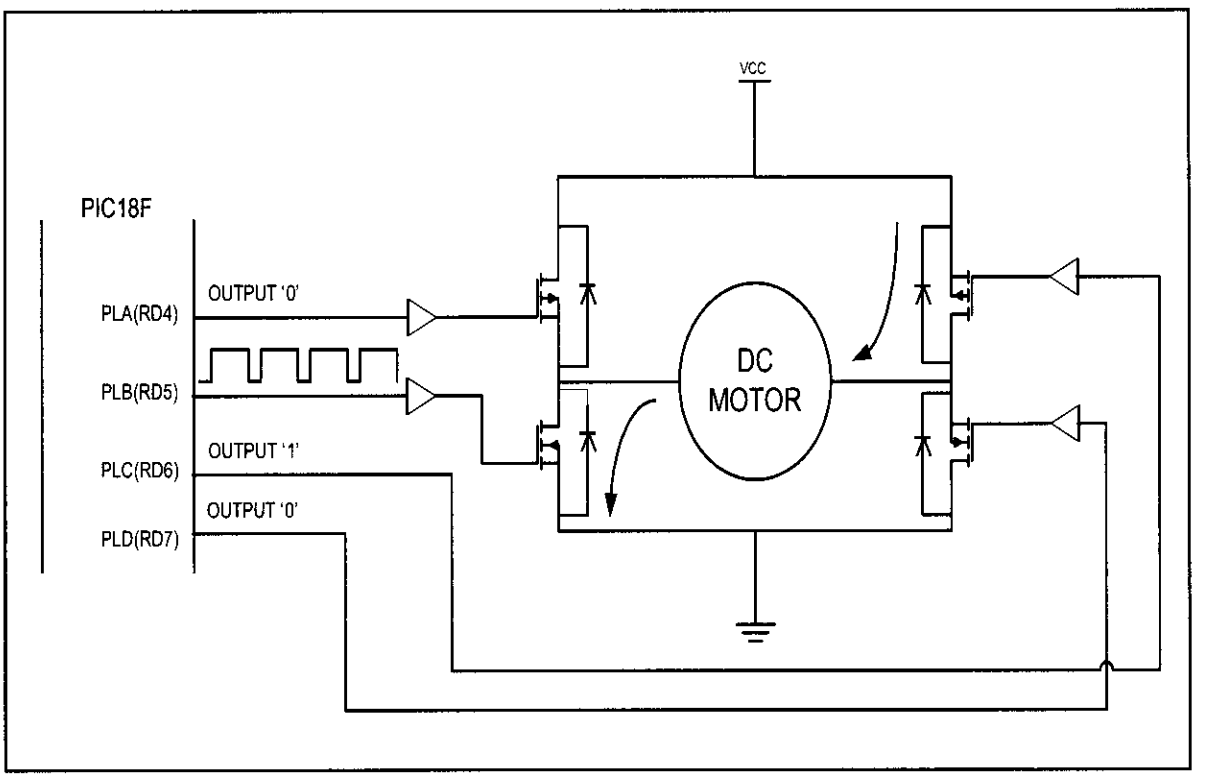


Figure 17-25. Reverse Current Flow Using ECCP (from Microchip)

EPWM1M1	EPWM1M0	EDC1B1	EDC1B0
D7			

ECCP1M3	ECCP1M2	ECCP1M1	ECCP1M0
			D0

EPWM1M1:EPWM1M0 PWM output pin configuration. It allows the use of a single pin for the capture/compare mode, or four pins for the PWM.
In compare/capture mode, only pin P1A (RD4) is used. In that case, there is no selection for these two bits.
In the PWM mode the options for these two bits are as follows:

00 P1A is used as a modulated output. P1B, P1C, and P1D are used as I/O.
01 Full-Bridge output forward. P1D modulated, P1A active. P1B and P1C inactive.

10 Half-Bridge output. P1A and P1D modulated with deadband control, P1C and P1D used as I/O.
11 Full-Bridge output reverse. P1B modulated, P1C active. P1A and P1D inactive.

EDC1B10:EDC1B1 PWM Duty Cycle least-significant bits. Used in PWM only.
The least-significant bits (Bit 1 and Bit 0) of the 10-bit duty cycle register are used in PWM. The ECCPR1L register is used as Bit 2 to Bit 9 of the 10-bit duty cycle register.

ECCP1M3–ECC1M0 ECCP1 Mode Select

0 0 0 0	ECCP1 is off
0 0 0 1	Reserved
0 0 1 0	Compare Mode. Toggle ECCP1 output pin on match. (ECCP1IF bit is set.)
0 0 1 1	Reserved
0 1 0 0	Capture mode, every falling edge
0 1 0 1	Capture mode, every rising edge
0 1 1 0	Capture mode, every 4th rising edge
0 1 1 1	Capture mode, every 16th rising edge
1 0 0 0	Compare mode. Initialize ECCP1 pin low; on compare match, force CCP1 pin HIGH. (ECCP1IF is set.)
1 0 0 1	Compare mode. Initialize CCP1 pin HIGH; on compare match, force CCP1 pin LOW. (ECCP1IF is set.)
1 0 1 0	Compare mode. Generate software interrupt on compare match. (ECCP1IF bit is set, ECCP1 pin is unaffected.)
1 0 1 1	Compare mode. Trigger special event. (ECCP1IF bit is set, and Timer1 or Timer3 is reset to zero.)
1 1 0 0	PWM Mode; P1A, P1C active-HIGH; P1B and P1D active-HIGH
1 1 0 1	PWM Mode; P1A, P1C active-HIGH; P1B and P1D active-LOW
1 1 1 0	PWM Mode; P1A, P1C active-LOW; P1B and P1D active-HIGH
1 1 1 1	PWM Mode; P1A, P1C active-LOW; P1B and P1D active-LOW

Figure 17-26. ECCP1 Control Register. (This register selects one of the operation modes of Capture, Compare, or PWM of EECP1)

Program 17-3 shows Full-Bridge implementation of the PWM for ECCP module. For the implementation of Half-Bridge and other applications of PWM using the ECCP module, see the PIC18 manual.

```
;Program 17-3
      CLRF TRISD              ;make PORTD output
      MOVLW D'100'
      MOVWF PR2               ;period = 100 * 16/Fosc
      MOVLW D'50'
      MOVWF ECCPR1L           ;duty = 50%
      MOVLW 0xCF
      MOVWF ECCP1CON          ;reverse full-bridge PWM
      MOVLW 0x24
      MOVWF T2CON             ;4 postscaler, turn on Timer2
AGAIN CLRF TMR2               ;start pulse
      BCF   PIR1,TMR2IF       ;clear flag
WAIT  BTFSS PIR1,TMR2IF       ;wait for period
      BRA   WAIT
      BRA   AGAIN             ;do it again
```

The following is the C version of the above program.

```
//Program 17-3C
#include <p18f458.h>

void main()
  {
      TRISD=0;                //make PORTD output
      PR2=100;                //period = 100 * 16/Fosc
      ECCPR1L=50;             //duty = 50%
      ECCP1CON=0xCF;          //reverse full-bridge PWM
      T2CON=0x24;             //4 postscaler,turn on Timer2
while(1)
      {
      TMR2=0;                 //start pulse
      PIR1bits.TMR2IF=0;          //clear flag
      while(PIR1bits.TMR2IF==0); //wait for period
      }
  }
```

Review Questions

1. True or false. For ECCP1, we use the RD3–RD0 pins for Full-Bridge.
2. True or false. For ECCP1, the P1A to P1D pins must be configured as output.
3. In ECCP1, we use _____ to set the period for PWM.
4. In ECCP1, we use _____ to set the duty cycle for PWM.
5. True or false. In ECCP1, we must use Timer2 for PWM.

SUMMARY

This chapter continued showing how to interface the PIC18 with real-world devices. Devices covered in this chapter were the relay, optoisolator, stepper motor, and DC motor.

First, the basic operation of relays and optoisolators was defined, along with key terms used in describing and controlling their operations. Then the PIC18 was interfaced with a stepper motor. The stepper motor was then controlled via an optoisolator using PIC18 Assembly and C programming languages.

The PIC18 was interfaced with DC motors. A typical DC motor will take electronic pulses and convert them to mechanical motion. This chapter showed how to interface the PIC18 with a DC motor. Then, simple Assembly and C programs were written to show the concept of PWM.

Control systems that require motors must be evaluated for the type of motor needed. For example, you would not want to use a stepper in a high-velocity application or a DC motor for a low-speed, high-torque situation. The stepper motor is ideal in an open-loop positional system and a DC motor is better for a high-speed conveyer belt application. DC motors can be modified to operate in a closed-loop system by adding a shaft encoder, then using a microcontroller to monitor the exact position and velocity of the motor. In the last two sections, we showed how to use CCP and ECCP features of PIC18 to control DC motors.

PROBLEMS

SECTION 17.1: RELAYS AND OPTOISOLATORS

1. True or false. The minimum voltage needed to energize a relay is the same for all relays.
2. True or false. The minimum current needed to energize a relay depends on the coil resistance.
3 Give the advantages of a solid-state relay over an EM relay.
4. True or false. In relays, the energizing voltage is the same as the contact voltage.
5. Find the current needed to energize a relay if the coil resistance is 1,200 ohms and the coil voltage is 5 V.
6. Give two applications for an optoisolator.
7 Give the advantages of an optoisolator over an EM relay.
8. Of the EM relay and solid-state relay, which has the problem of back EMF?
9. True or false. The greater the coil resistance, the worse the back EMF voltage.
10. True or false. We should use the same voltage sources for both the coil voltage and contact voltage.

SECTION 17.2: STEPPER MOTOR INTERFACING

11. If a motor takes 90 steps to make one complete revolution, what is the step angle for this motor?
12. Calculate the number of steps per revolution for a step angle of 7.5 degrees.

13. Finish the normal four-step sequence clockwise if the first step is 0011 (binary).
14. Finish the normal four-step sequence clockwise if the first step is 1100 (binary).
15. Finish the normal four-step sequence counterclockwise if the first step is 1001 (binary).
16. Finish the normal four-step sequence counterclockwise if the first step is 0110 (binary).
17. What is the purpose of the ULN2003 placed between the PIC18 and the stepper motor? Can we use that for 3A motors?
18. Which of the following cannot be a sequence in the normal four-step sequence for a stepper motor?
    (a) CCH (b) DDH (c) 99H (d) 33H
19. What is the effect of a time delay between issuing each step?
20. In Question 19, how can we make a stepper motor go faster?

SECTION 17.3: DC MOTOR INTERFACING AND PWM

21. Which motor is best for moving a wheel exactly 90 degrees?
22. True or false. Current dissipation of a DC motor is proportional to the load.
23. True or false. The rpm of a DC motor is the same for no-load and loaded.
24. The rpm given in data sheets is for _____ (no-load, loaded).
25. What is the advantage of DC motors over AC motors?
26. What is the advantage of stepper motors over DC motors?
27. True or false. Higher load on a DC motor slows it down if the current and voltage supplied to the motor are fixed.
28. What is PWM, and how is it used in DC motor control?
29. A DC motor is moving a load. How do we keep the rpm constant?
30. What is the advantage of placing an optoisolator between the motor and the microcontroller?

ANSWERS TO REVIEW QUESTIONS

SECTION 17.1: RELAYS AND OPTOISOLATORS

1. With a relay we can use a 5 V digital system to control 12 V–120 V devices such as horns and appliances.
2. Because microcontroller/digital outputs lack sufficient current to energize the relay, we need a driver.
3. When the coil is not energized, the contact is closed.
4. When current flows through the coil, a magnetic field is created around the coil, which causes the armature to be attracted to the coil.
5. It is faster and needs less current to get energized.
6. It is smaller and can be connected to the microcontroller directly without a driver.

SECTION 17.2: STEPPER MOTOR INTERFACING

1. 0110, 0011, 1001, 1100 for clockwise; and 0110, 1100, 1001, 0011 for counterclockwise
2. 72
3. Because the microcontroller pins do not provide sufficient current to drive the stepper motor

SECTION 17.3: DC MOTOR INTERFACING AND PWM

1. True
2. False
3. Because microcontroller/digital outputs lack sufficient current to drive the DC motor, we need a driver.
4. By reversing the polarity of voltages connected to the leads
5. The DC motor is stalled if the load is beyond what it can handle.
6. False
7. No-load

SECTION 17.4: PWM MOTOR CONTROL WITH CCP

1. True
2. True
3. PR2
4. CCPR1L
5. False

SECTION 17.5: DC MOTOR CONTROL WITH ECCP

1. False
2. True
3. PR2
4. CCPR1L
5. True