Unit 4: Facilities in PIC18 Part-II

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

$$\text{Desired Baud Rate} = \text{Fosc}/(64X - 64) = \text{Fosc}/64(X + 1)$$

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

$$\text{Desired Baud Rate} = \text{Fosc}/64(X + 1) = 10\ \text{MHz}/64(X + 1) = 6250\ \text{Hz}/(X + 1)$$

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

$$X = (156250/\text{Desired Baud Rate}) - 1$$

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.

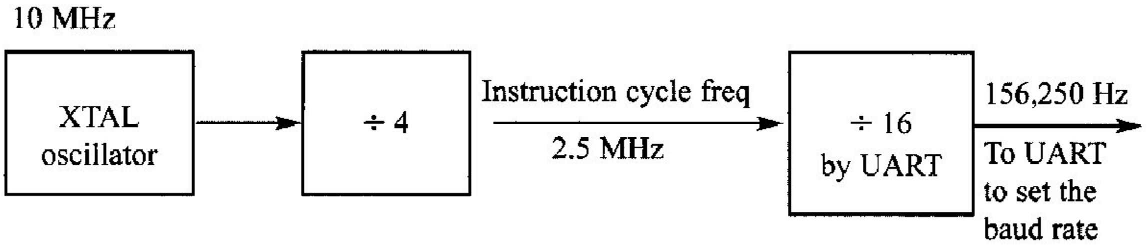


Table 10-4: SPBRG Values for Various Baud Rates (Fosc = 10 MHz, BRGH = 0)

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.

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'
```

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.

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

Program thc 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
```

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

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

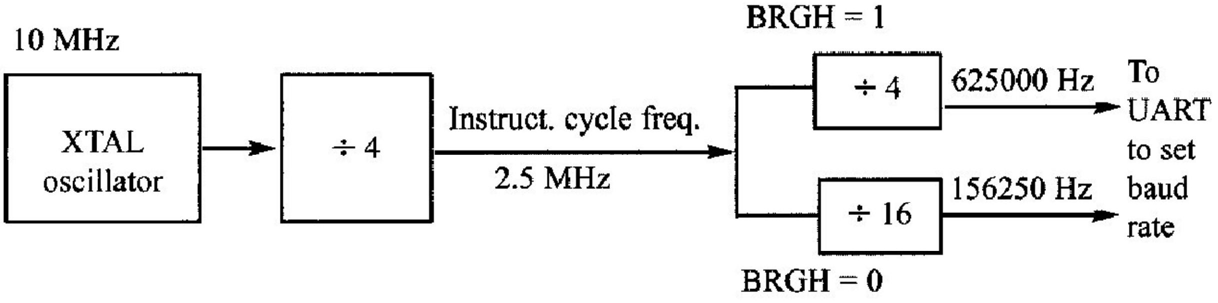
Assume that XTAL = 10 MHz.

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).



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

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.

ADC programming in PIC18:-

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).

(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)

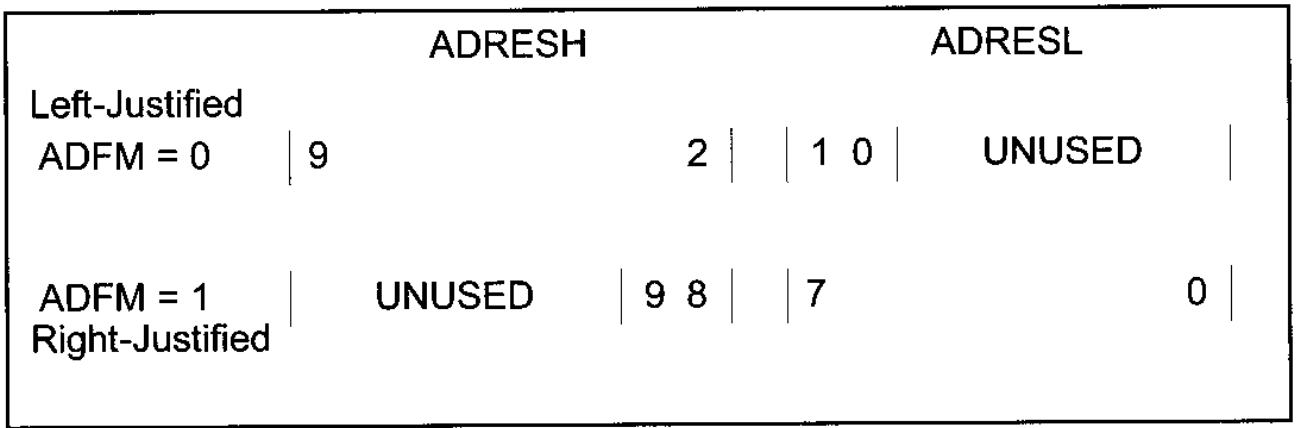


Figure 13-8. ADFM Bit and ADRESx Registers

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.

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.

A PIC18 is connected to thc 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.

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

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

```
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
```

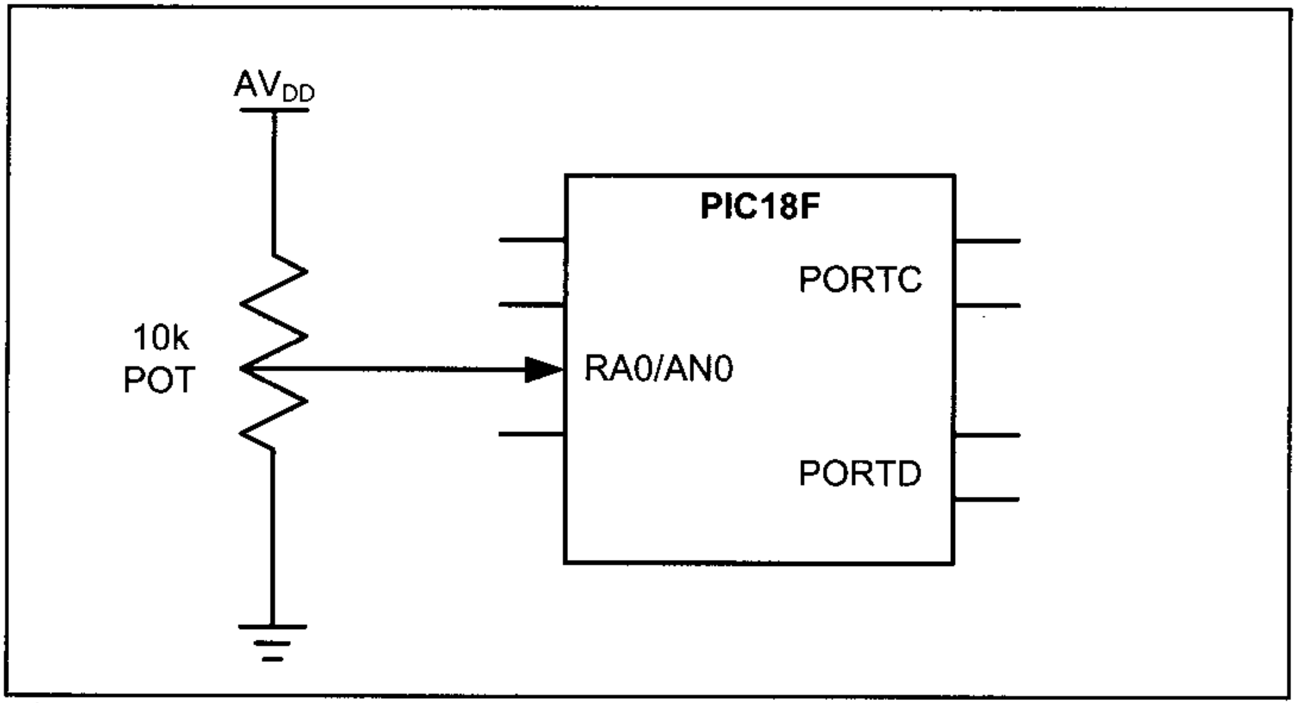


Figure 13-9. A/D Connection for Program 13-1

Programming 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

```
;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
```