8

Timers and CCP Modules

8.1 Objectives

After completing this chapter, you should be able to

- Explain the overall timer structure of the PIC18 microcontroller
- Use timer function to create time delays
- Use timer function to measure the frequency of an unknown signal
- Use CCP in capture mode to measure the duration of a pulse or the period of a square wave
- Use CCP in capture mode to measure the duty cycle of a waveform or the phase difference between two waveforms having the same frequency
- Use CCP in capture mode to measure the frequency of an unknown signal
- Use CCP in compare mode to create time delays
- Use CCP in compare mode to trigger pin actions
- Use CCP in compare mode to generate digital waveforms or pulses
- Use CCP in compare mode to create music
- Use CCP in PWM mode to generate digital waveform with certain frequencies and duty cycles
- Use CCP and ECCP in motor control

8.2 Overview of PIC18 Timer Functions

In a digital system, *time* is represented by the *count* of a *timer*. There are many applications that are very difficult or even impossible to implement without the timer function:

- Event arrival time recording and comparison
- Periodic interrupt generation
- Pulse width and period measurement
- Frequency and duty cycle measurement of periodic signals
- Generation of waveforms with certain duty cycles and frequencies
- Time references
- Event counting

A PIC18 microcontroller may have four or five timers. These timers are Timer0, Timer1, Timer2, Timer3, and Timer4.

Timer0, Timer1, and Timer3 are 16-bit timers, whereas Timer2 and Timer4 are 8-bit timers. Whenever a 16-bit (8-bit) timer rolls over from 0xFFFF to 0x0000 (from 0xFF to 0x00), an interrupt will be requested if it is enabled. Both Timer2 and Timer4 use the instruction cycle clock as the clock source, whereas the other timers can also use external clock signal as their clock source.

CCP stands for *capture, compare,* and *pulse-width modulation* (PWM). A PIC18 microcontroller may have one, two, or five CCP modules. An 18-pin or 20-pin PIC18 device has only one CCP channel. The PIC18F8X2X and PIC18F6X2X devices have five CCP channels. Other PIC18 devices (at the time of this writing) have two CCP channels. Each CCP channel can be configured to perform capture, compare, or PWM functions. These three functions share the same signal pin and registers.

When configured as a capture function, the CCP module can be programmed to copy the contents of a timer into a capture register on every falling edge, every rising edge, every fourth rising edge, or every 16th rising edge. This capability can be utilized to measure the pulse width, period, duty cycle, or frequency. The capture function can also be used for timing reference.

When configured as a compare function, the CCP module compares the contents of the 16-bit CCPRx (x = 1 . . . 5) register with the TMR1 (or TMR3) register in every clock cycle. When these two registers match, the CCP module can optionally drive the associated pin to high or low or simply toggled. The compare function can be used to create a time delay, generate a few pulses, or generate periodic waveforms.

When configured as a PWM function, the CCP module can be programmed to generate a waveform with a certain frequency and duty cycle. This function is often used in motor control and light dimming applications.

8.3 Timers

A PIC18 microcontroller may have four or five timers. The PIC18F8720, 18F8621, 18F8620, 18F8525, 18F8520, 18F6720, 18F6620, 18F6621, 18F6525, and 18F6520 have five timers. Timer4 is available only in these devices. A detailed discussion of these timers is given in this section.

8.3.1 Timer0

Timer0 can be configured as an 8-bit or a 16-bit timer or counter. The user can choose the internal instruction cycle clock or the external T0CKI signal as the clock source of Timer0. The user can choose to divide the clock signal by a prescaler before it is connected to the clock input to Timer0. The block diagrams of Timer0 in 8-bit and 16-bit mode are shown in Figures 8.1a and 8.1b.

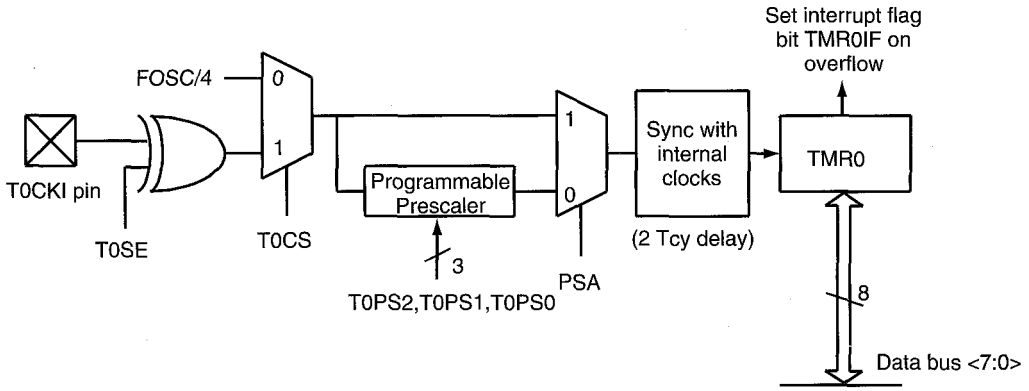

Figure 8.1a ■ Timer0 block diagram in 8-bit mode (redraw with permission of Microchip)

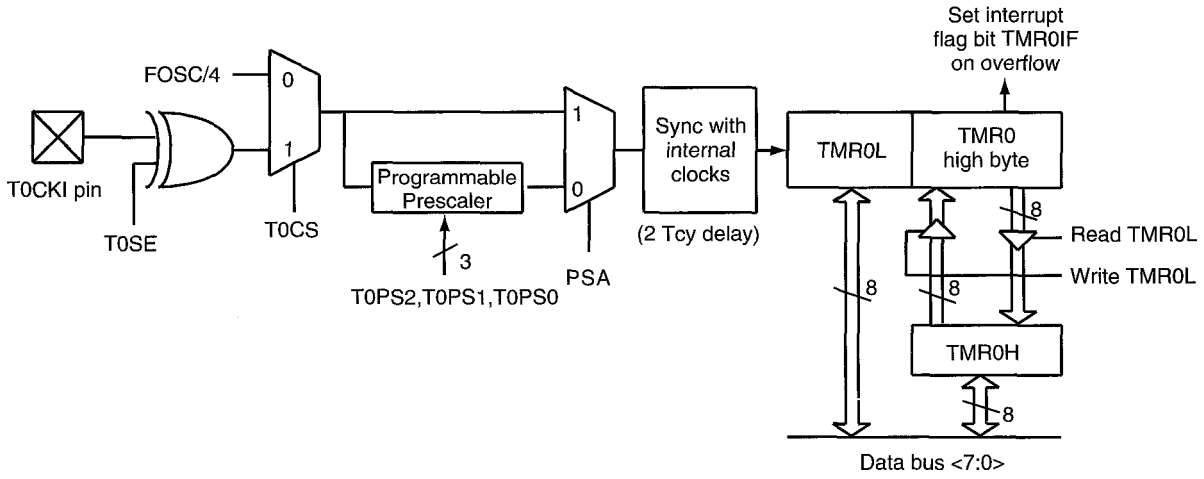

Figure 8.1b ■ Timer0 block diagram in 16-bit mode (redraw with permission of Microchip)

The T0CON register is a readable and writable register that controls the operation parameters of Timer0, including the prescaler selection. The contents of the T0CON register are shown in Figure 8.2.

	7	6	5	4	3	2	1	0
value after reset	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
	1	1	1	1	1	1	1	1

TMR0ON: *Timer0 on/off control bit*
0 = stops Timer0
1 = Enables Timer0
T08BIT: *Timer0 8-bit/16-bit control bit*
0 = Timer0 is configured as a 16-bit timer
1 = Timer0 is configured as an 8-bit timer
T0CS: *Timer0 clock source select*
0 = Instruction cycle clock
1 = Transition on T0CKI pin
T0SE: *Timer0 source edge select bit*
0 = Increment on falling edge transition on T0CKI pin
1 = Increment on rising edge transition on T0CKI pin
PSA: *Timer0 prescaler assignment bit*
0 = Timer0 prescaler is assigned . Timer0 clock input comes from prescaler output.
1 = Timer0 prescaler is not assigned. Timer0 clock input bypasses prescaler.
T0PS2:T0PS0: *Timer0 prescaler select bits*
000 = 1:2 prescaler value
001 = 1:4 prescaler value
010 = 1:8 prescaler value
011 = 1:16 prescaler value
100 = 1:32 prescaler value
101 = 1:64 prescaler value
110 = 1:128 prescaler value
111 = 1:256 prescaler value

Figure 8.2 ■ T0CON register (reprint with permission of Microchip)

Timer0 can operate as a timer or as a counter. When the clock source is the instruction cycle clock, Timer0 operates as a timer. When the clock source comes from the T0CKI pin, Timer0 operates as a counter. By setting the T0CS bit of the T0CON register to 0, the internal instruction cycle clock is used as the clock source to Timer0. Otherwise, the signal from the T0CKI pin will be selected as the clock source, and Timer0 will operate as a counter. When the T0CKI signal is selected, the user has the choice of using either the rising or the falling edge of the T0CKI signal to increment the counter register pair TMR0L and the TMR0 high byte.

As shown in Figure 8.1b, the 8-bit register TMR0H acts as a buffer between the TMR0 high byte and the internal data bus. When the CPU reads the TMR0L register, the content of the TMR0 high byte is transferred to TMR0H. When the MCU reads the high byte of Timer0, it reads from TMR0H. Since the PIC18 microcontroller is 8-bit, it can read only eight bits at a time. Because of this feature, the 16-bit count value will be the value when the CPU reads the TMR0L register. Otherwise, the user may read an invalid value because of the rollover between successive reads of the high and low bytes. This feature is not available in the PIC16 and PIC17 devices.

Example 8.1

Suppose Timer0 does not have the buffer register TMR0H. What would be the value read from Timer0 with the following instruction sequence, assuming that the count value is 0x35FE when the first byte is read?

```
movff   TMR0L,PRODL
movff   TMR0H,PRODH
```

Solution: The **movff** instruction takes two clock cycles to execute. When the second instruction is executed, Timer0 has rolled over to 0x3600. Therefore, the value transferred to PRODH will be 0x36. Since the value transferred to PRODL is 0xFE, the value copied to the PRODH..PRODL pair becomes 0x36FE and is larger than the valid value by 256.

This problem cannot be solved by reversing the order of reading. The reason for this problem is left for you as an exercise problem.

Creating time delays is an important application for Timer0. By setting the prescaler properly, a wide range of delays can be created. The following example illustrates the procedure.

Example 8.2

Write a subroutine to create a time delay that is equal to 100 ms times the contents of the PRODL register, assuming that the crystal oscillator is running at 32 MHz.

Solution: The instruction clock frequency is 8 MHz and has a period of 125 ns. The 100-ms time delay can be created as follows:

1. Place the value 15535 into the TMR0 high byte and the TMR0L register so that Timer0 will overflow in 50000 clock cycles.
2. Choose internal instruction cycle clock as the clock source and set the prescaler to 16 so that Timer0 will roll over in 100 ms.
3. Enable Timer0.
4. Wait until Timer0 overflows.

The subroutine that implements this procedure is as follows:

```
delay   movlw   0x83            ; enable TMR0, select internal clock,
        movwf   T0CON           ; set prescaler to 16
loopd   movlw   0x3C            ; load 15535 into TMR0 so that it will
        movwf   TMR0H           ; roll over in 50000 clock cycles
        movlw   0xAF            ; "
        movwf   TMR0L           ; "
        bcf     INTCON,TMR0IF   ; clear the TMR0IF flag
wait    btfss   INTCON,TMR0IF   ;
        bra     wait            ; wait until 100 ms is over
        decfsz  PRODL,F
        bra     loopd
        return
```

In C language, this function can be written as follows:

```
void delay (char cx)
{
    int i;
    T0CON = 0x83;          /* enable TMR0, select instruction clock, prescaler set to 16 */
    for (i = 0; i < cx; i++) {
        TMR0 = 15535;      /* load 15535 into TMR0 so that it rolls */
                           /* over in 50000 clock cycles */
        INTCONbits.TMR0IF = 0;
        while(!(INTCONbits.TMR0IF)); /* wait until TMR0 rolls over */
    }
    return;
}
```

8.3.2 Timer1

Timer1 is a 16-bit timer/counter, depending on the clock source. The 16-bit timer/counter (TMR1H/TMR1L) is readable and writable. An interrupt will be requested whenever Timer1 rolls over from 0xFFFF to 0x0000. The block diagram of Timer1 is shown in Figure 8.3. Timer1 can be reset when the CCP module is configured in compare mode to generate a *special event trigger*. An A/D conversion will also be started by this reset.

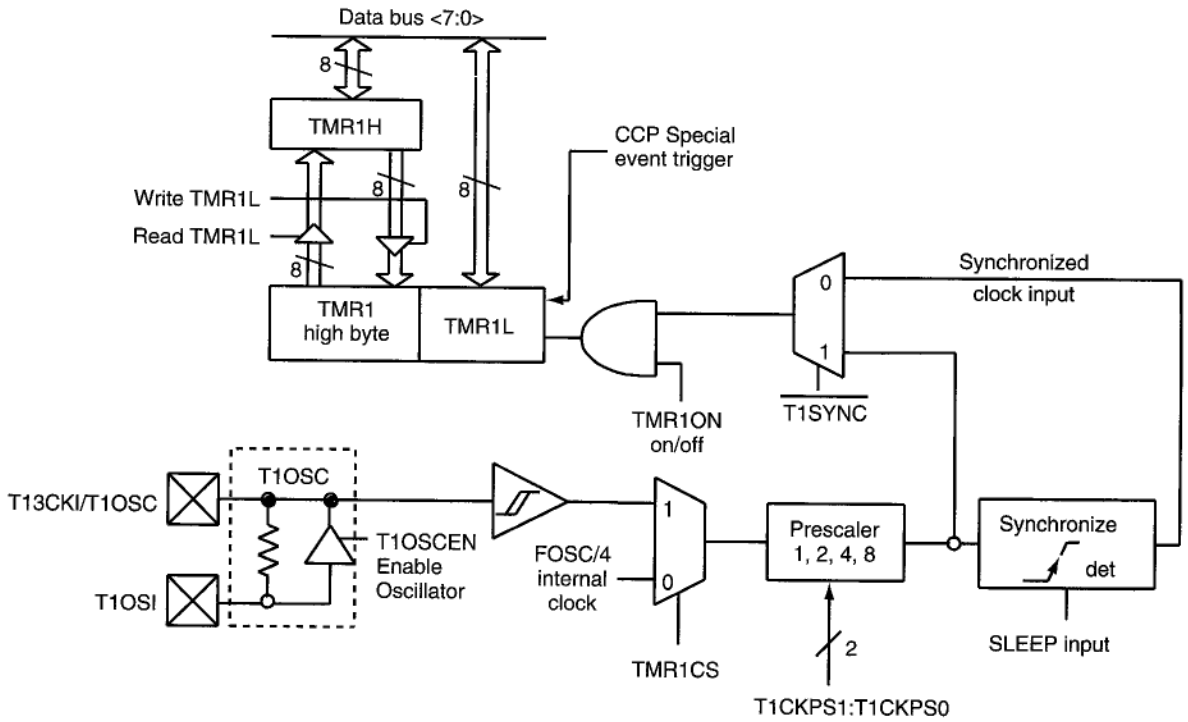

Figure 8.3 ■ Timer1 block diagram: 16-bit mode (redraw with permission of Microchip)

Timer1 operation is controlled by the T1CON register. The contents of the T1CON register are shown in Figure 8.4.

	7	6	5	4	3	2	1	0
value after reset	RD16	--	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
	0	0	0	0	0	0	0	0

RD16: 16-bit read/write mode enable bit
0 = Enables read/write of Timer1 in two 8-bit operations
1 = Enable read/write of Timer1 in 16-bit operation
T1CKPS1:T1CKPS0: Timer1 input clock prescale select bits
00 = 1:1 prescale value
01 = 1:2 prescale value
10 = 1:4 prescale value
11 = 1:8 prescale value
T1OSCEN: Timer1 oscillator enable bit
0 = Timer1 oscillator is shut off
1 = Timer1 oscillator is enabled
$\overline{\text{T1SYNC}}$: Timer1 external clock input synchronization select bit
When TMR1CS = 1
0 = Synchronize external clock input
1 = Do not synchronize external clock input
When TMR1CS = 0
This bit is ignored.
TMR1CS: Timer1 clock source select bit
0 = Instruction cycle clock (FOSC/4)
1 = External clock from pin RC0/T1OSO/T13CKI
TMR1ON: Timer1 on bit
0 = Stop Timer1
1 = Enables Timer1

Figure 8.4 ■ T1CON contents (redraw with permission of Microchip)

When the clock source to Timer1 is the instruction cycle clock, it works as timer. When the external clock signal is selected, Timer1 can be a synchronous or an asynchronous counter, depending on the setting of the T1SYNC bit in the T1CON register. When this bit is clear, the external clock is synchronized with the instruction cycle clock.

When the TMRCS1 bit of the T1CON register is set, Timer1 increments on the rising edge of the external clock input (RC0/T1OSO/T1CKI pin) or the Timer1 oscillator, if enabled. When the Timer1 oscillator is enabled, the RC1/T1OSI and RC0/T1OSO/T1CKI pins become inputs. The TRISC<1:0> value has no effect on the pin direction under this condition. The Timer1 oscillator is intended primarily for a 32-KHz crystal.

To take advantage of the feature of the special event trigger, Timer1 must be configured in compare mode. This event will reset Timer1 but will not set the interrupt flag TMR1IF.

Like Timer0, the high byte of Timer1 is double buffered. When reading TMR1L, the high byte of TMR1 will be loaded into the TMR1H register in Figure 8.3. This ensures that the values read from TMR1H and TMR1L belong together.

Like Timer0, Timer1 can also be used to create time delays. A more interesting application is *frequency measurement*. The method is to use one of the timers to create a 1-second time delay and another timer as a counter to count the incoming clock cycles within 1 second. The following example illustrates this idea.

Example 8.3

Use Timer0 as a timer to create a 1-second delay and use Timer1 as a counter to count the rising (or falling) edges of an unknown signal (at the T0CKI pin) arriving in 1 second, which would measure the frequency of the unknown signal. Write a program to implement this idea, assuming that the PIC18 microcontroller is running with a 32-MHz crystal oscillator.

Solution: A 1-second delay can be created by loading 10 into the PRODL register and calling the subroutine in Example 8.2. Since Timer1 is only 16 bits, it may overflow many times in 1 second. If the unknown signal has the same frequency as the PIC18 microcontroller instruction cycle clock, it may overflow 122 times. Therefore, it is adequate to use one byte to keep track of the number of times that Timer1 overflows. Timer1 **overflow count** can be incremented by using interrupts.

Timer0 should be configured as follows:

- 16-bit mode
- Using instruction cycle clock as the clock source
- Set prescaler to 16
- Enabled to count

Timer1 should be configured as follows:

- 16-bit mode
- Prescaler value set to 1
- Disable oscillator
- Do not synchronize external clock input
- Select external T1CKI pin signal as clock source

The setting of interrupt is as follows:

- Enable priority interrupt
- Place Timer1 interrupt at high priority
- Enable only Timer1 rollover interrupt

The assembly program that implements the frequency measurement algorithm and settings is as follows:

```
            #include <p18F452.inc>
t1ov_cnt    set     0x00            ; Timer1 rollover interrupt count
freq        set     0x01            ; to save the contents of Timer1 at the end
            org     0x00
            goto    start
```

```
; high priority interrupt service routine
        org     0x08
        btfss   PIR1,TMR1IF     ; skip if Timer1 roll-over interrupt occurs
        retfie                  ; return if not Timer1 interrupt
        bcf     PIR1,TMR1IF     ; clear the interrupt flag
        incf    t1ov_cnt,F      ; increment Timer1 roll-over count
        retfie
; dummy low priority interrupt service routine
        org     0x18
        retfie
start   clrf    t1ov_cnt        ; initialize Timer1 overflow cnt to 0
        clrf    freq            ; initialize frequency to 0
        clrf    freq+1          ; "
        clrf    TMR1H           ; initialize Timer1 to 0
        clrf    TMR1L           ; "
        clrf    PIR1            ; clear all interrupt flags
        bsf     RCON,IPEN       ; enable priority interrupt
        movlw   0x01            ; set TMR1 interrupt to high priority
        movwf   IPR1            ; "
        movwf   PIE1            ; enable Timer1 roll-over interrupt
        movlw   0x87            ; enable Timer1, select external clock, set
        movwf   T1CON           ; prescaler to 1, disable crystal oscillator
        movlw   0xC0            ; enable global and peripheral interrupt
        movwf   INTCON          ; "
        movlw   0x0A
        movwf   PRODL           ; prepare to call delay to wait for 1 second
        call    delay           ; Timer1 overflow interrupt occur in this second
        movff   TMR1L,freq      ; save frequency low byte
        movff   TMR1H,freq+1    ; save frequency high byte
        bcf     INTCON,GIE      ; disable global interrupt
forever goto    forever
; ***************************
; include the delay subroutine here.
; ***************************
```

The C language version of the program is as follows:

```
#include  <p18F452.h>
unsigned  int t1ov_cnt;
unsigned  short long freq;
void high_ISR(void);
void low_ISR(void);
#pragma code high_vector = 0x08     // force the following statement to
void high_interrupt (void)          // start at 0x08
{
        _asm
        goto high_ISR
        _endasm
}
```

```
#pragma code low_vector = 0x18      //force the following statements to start at
void low_interrupt (void)           //0x18
{
        _asm
        goto    low_ISR
        _endasm
}
#pragma code                        //return to the default code section
#pragma interrupt high_ISR
void high_ISR (void)
{
        if(PIR1bits.TMR1IF){
          PIR1bits.TMR1IF = 0;
          t1ov_cnt ++;
        }
}
#pragma code
#pragma interrupt low_ISR
void low_ISR (void)
{
        _asm
        retfie 0
        _endasm
}
void delay (char cx);               /* prototype declaration */
void main (void)
{
        char t0_cnt;
        char temp;
        t1ov_cnt = 0;
        freq     = 0;
        TMR1H    = 0;               /* force Timer1 to count from 0 */
        TMR1L    = 0;               /* " */
        PIR1     = 0;               /* clear Timer1 interrupt flag */
        RCONbits.IPEN = 1;          /* enable priority interrupt */
        IPR1     = 0x01;            /* set Timer1 interrupt to high priority */
        PIE1     = 0x01;            /* enable Timer1 roll-over interrupt */
        T1CON    = 0x83;            /* enable Timer1 with external clock, prescaler 1 */
        INTCON   = 0xC0;            /* enable global and peripheral interrupts */
        delay (10);                 /* create one-second delay and wait for interrupt */
        INTCONbits.GIE = 0;         /* disable global interrupt */
        temp     = TMR1L;
        freq     = t1ov_cnt * 65536 + TMR1H * 256 + temp;
}
void delay (char cx)
{
        int i;
        T0CON = 0x83;               /* enable TMR0, select instruction clock, prescaler set to 16 */
```

```
        for (i = 0; i < cx; i++) {
        TMR0H = 0x3C;              /* load 15535 into TMR0 so that it rolls */
        TMR0L = 0xAF;              /* over in 50000 clock cycles */
        INTCONbits.TMR0IF = 0;
        while(!(INTCONbits.TMR0IF)); /* wait until TMR0 rolls over */
        }
        return;
}
```

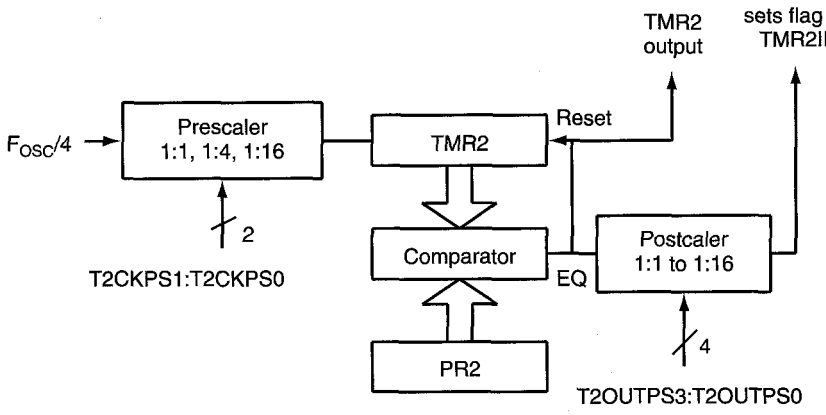

Figure 8.5 ■ Timer2 block diagram (redraw with permission of Microchip)

8.3.3 Timer2

Timer2 has an 8-bit timer (TMR2) and an 8-bit period register (PR2). Both registers are readable and writable. The block diagram of Timer2 is shown in Figure 8.5.

As shown in Figure 8.5, the clock source to TMR2 is $F_{OSC}/4$ divided by a programmable prescale factor. TMR2 is counting up and comparing with the value in the PR2 register in every clock cycle. When these two registers are equal (for one clock cycle only), the EQ signal will reset TMR2 to 0. The output of the comparator is divided by a programmable postscale factor. The equal signal (EQ) goes through this postscale circuit to generate a TMR2 interrupt. The output of TMR2 (before the postscaler) is fed to the synchronous serial port module, which may optionally use it to generate the shift clock.

The prescaler and postscaler counters are cleared when any of the following occurs:

- A write to the TMR2 register
- A write to the T2CON register
- Any device RESET

The operation parameters of Timer2 are configured by the T2CON register. The contents of T2CON are shown in Figure 8.6. Both the prescale and the postscale factors are programmed via this register.

	7	6	5	4	3	2	1	0
value after reset	--	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
	0	0	0	0	0	0	0	0

TOUTPS3:TOUTPS0: Timer2 output postscale select bits
0000 = 1:1 postscale
0001 = 1:2 postscale
.
.
.
1111 = 1:16 postscale
TMR2ON: Timer2 on bit
0 = Timer2 is off
1 = Timer2 is on
T2CKPS1: T2CKPS0: Timer2 clock prescale select bits
00 = prescaler is 1
01 = prescaler is 4
1x = prescaler is 16

Figure 8.6 ■ T2CON control register (redraw with permission of Microchip)

Because TMR2 is reset whenever it is equal to the PR2 register, it can be used to generate periodic interrupts. The next example illustrates this application.

Example 8.4

Assume that the PIC18F8680 is running with a 32-MHz crystal oscillator. Write an instruction sequence to generate periodic interrupts every 8 ms with high priority.

Solution: We need to compute the value to be written into the PR2 register to make this happen. Suppose we set the prescaler and postscaler to 16. Then the value to be written into PR2 register is

$$PR2 = \text{number of counts in 8 ms} = 8 \times 10^{-3} \times 32 \times 10^{6} \div 4 \div 16 \div 16 - 1 = 249$$

The following instruction sequence will configure Timer2 to generate periodic interrupts every 8 ms:

```
movlw   D'249'          ; load 249 into PR2 so that TMR2 counts up
movwf   PR2,A           ; to 249 and reset
bsf     RCON,IPEN,A     ; enable priority interrupt
bsf     IPR1,TMR2IP,A   ; place TMR2 interrupt at high priority
bcf     PIR1,TMR2IF,A   ;
movlw   0xC0
movwf   INTCON,A        ; enable global interrupt
movlw   0x7E            ; enable TMR2, set prescaler to 16, set
movwf   T2CON,A         ; postscaler to 16
bsf     PIE1,TMR2IE,A   ; enable TMR2 overflow interrupt
```

8.3.4 Timer3

Timer3 consists of two readable and writable 8-bit registers (TMR3H and TMR3L). Timer3 can choose to use either the internal (instruction cycle clock) or external signal (T13CKI) as its clock source. The block diagram of Timer3 is quite similar to that of Timer3 and is shown in Figure 8.7.

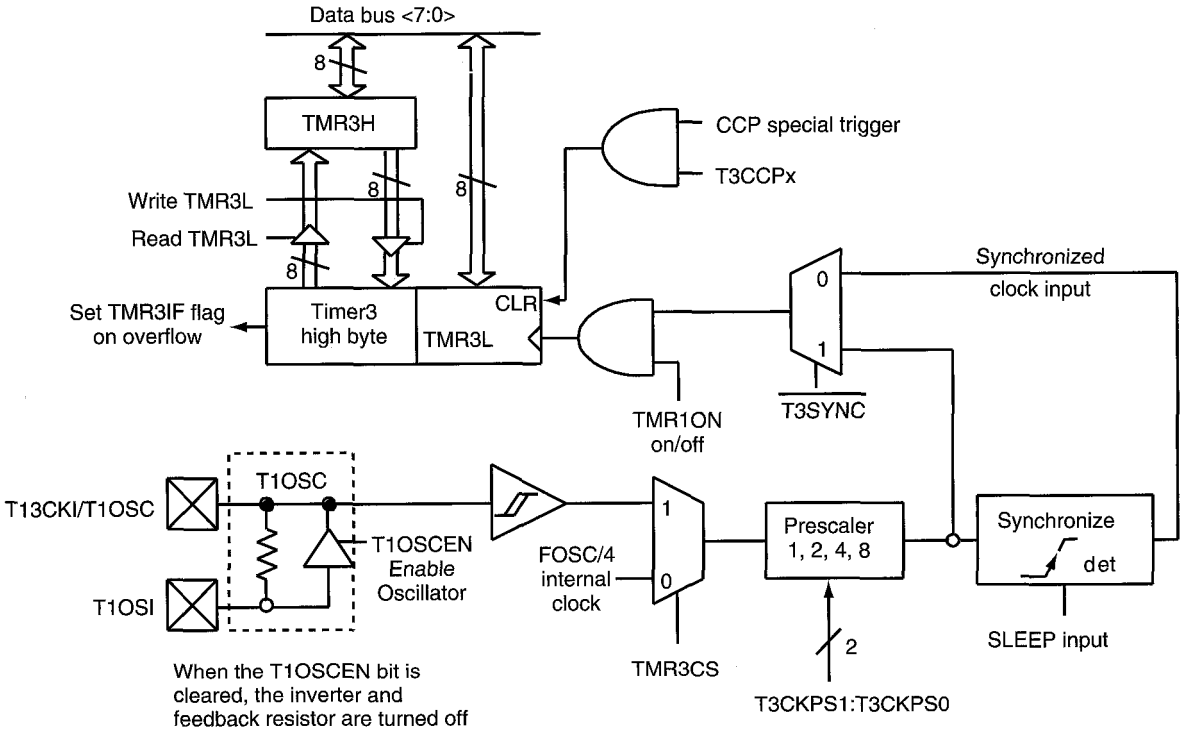

Figure 8.7 ■ Timer3 block diagram: 16-bit mode (redraw with permission of Microchip)

Timer3 can operate in three modes:

- Timer
- Synchronous counter
- Asynchronous counter

When the clock source is the instruction cycle clock, Timer3 operates as a timer. When the external clock source is selected, Timer3 operates as a counter. The external clock source could be the T13CKI signal or the crystal oscillator connected to the T1OSO and T1OSI pins. Timer3 also has an internal reset input. This RESET signal can be generated by the CCP module.

Timer3 may optionally generate an interrupt to the CPU when it rolls over from 0xFFFF to 0x0000. When rollover occurs, the TMR3IF flag (in PIR2 register) will be set to 1. Timer3 operation parameters are configured via the T3CON register. The contents of the T3CON register are shown in Figure 8.8.

	7	6	5	4	3	2	1	0
value after reset	RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON
	0	0	0	0	0	0	0	0

RD16: 16-bit read/write mode enable bit
- 0 = Enables read/write of Timer3 in two 8-bit operations
- 1 = Enables read/write of Timer3 in 16-bit operation

T3CCP2:T3CCP1: Timer3 and Timer1 to CCPx enable bits
- 00 = Timer1 and Timer2 are the clock sources for CCP1 through CCP5
- 01 = Timer3 and Timer4 are the clock sources for CCP2 through CCP5; Timer1 and Timer2 are the clock sources for CCP1
- 10 = Timer3 and Timer4 are the clock sources for CCP3 through CCP5; Timer1 and Timer2 are the clock sources for CCP1 and CCP2
- 11 = Timer3 and Timer4 are the clock sources for CCP1 through CCP5

T3CKPS1:T3CKPS0: Timer3 input clock prescale select bits
- 00 = 1:1 prescale value
- 01 = 1:2 prescale value
- 10 = 1:4 prescale value
- 11 = 1:8 prescale value

$\overline{\text{T3SYNC}}$: Timer3 external clock input synchronization select bit

When TMR3CS = 1
- 0 = Synchronizes external clock input
- 1 = Do not synchronize external clock input

When TMR3CS = 0
- This bit is ignored.

TMR3CS: Timer3 clock source select bit
- 0 = Instruction cycle clock (FOSC/4)
- 1 = External clock from pin RC0/T1OSO/T13CKI

TMR3ON: Timer3 on bit
- 0 = Stops Timer3
- 1 = Enables Timer3

Figure 8.8 ■ T3CON contents (redraw with permission of Microchip)

Reading the TMR3L register will transfer the high byte of Timer3 into the TMR3H register. Writing to the TMR3L register will transfer the value of TMR3H into the high byte of Timer3. This feature enables the 16-bit value written into and read (in two separate operations) from the Timer3 register to belong together.

8.3.5 Timer4

Timer4 is available only in the PIC18F8X20 and PIC18F6X20 devices. It is an 8-bit timer and has an 8-bit period register. The block diagram of Timer4 is shown in Figure 8.9.

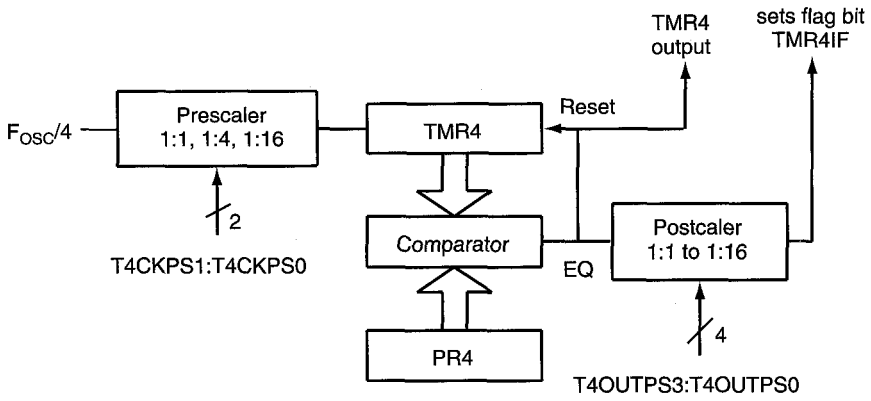

Figure 8.9 ■ Timer4 block diagram (redraw with permission of Microchip)

The output of Timer4 is used mainly as a PWM base timer for the CCP module (to be discussed later). The content of the T4CON register is identical to that of T2CON.

Timer4 increments from 0x00 until it matches the value of the PR4 register and then resets to 0x00 in the next increment cycle. On reset, the PR4 register is initialized to 0xFF. The input clock is divided by a programmable prescaler. The match output of TMR4 goes through a 4-bit postscaler to generate a TMR4 interrupt (when enabled).

The prescaler and postscaler counters are cleared when any of the following occurs:

- A write to the TMR4 register
- A write to the T4CON register
- Any device RESET (power-on reset, MCLR reset, watchdog timer reset, or brown-out reset)

8.3.6 C Library Functions for Timers

Microchip C18 compiler provides library functions for configuring, disabling, and reading from and writing into the timers. One needs to include the header file **timers.h** in order to use these library functions.

The library functions for disabling timers are the following:

void **CloseTimer0** (void);

void **CloseTimer1** (void);

void **CloseTimer2** (void);

void **CloseTimer3** (void);

void **CloseTimer4** (void);

The library functions for configuring timers are the following:

void **OpenTimer0** (unsigned char **config**);

void **OpenTimer1** (unsigned char **config**);

void **OpenTimer2** (unsigned char **config**);

void **OpenTimer3** (unsigned char **config**);

void **OpenTimer4** (unsigned char **config**);

The argument to the **OpenTimer0** function is a bit mask that is created by performing a bit-wise AND operation with a value from each of the categories listed here:

Enable Timer0 Interrupt	
TIMER_INT_ON	enable interrupt
TIMER_INT_OFF	disable interrupt
Timer Width	
T0_8BIT	8-bit mode
T0_16BIT	16-bit mode
Clock Source	
T0_SOURCE_EXT	external clock source
T0_SOURCE_INT	internal clock source
External Clock Trigger	
T0_EDGE_FALL	External clock on falling edge
T0_EDGE_RISE	External clock on rising edge
Prescale Value	
T0_PS_1_n	1: n prescale (n = 1, 2, 4, 8, 16, 32, 64, 128, or 256)

The arguments to other timers are similar and are described in Appendix D. An example of a call to OpenTimer0 is as follows:

```
OpenTimer0 (TIMER_INT_ON & T0_8BIT & T0_SOURCE_INT & T0_PS_1_32);
```

The prototype definitions of library functions for reading timer values are the following:

```
unsigned int    ReadTimer0 (void);
unsigned int    ReadTimer1 (void);
unsigned char   ReadTimer2 (void);
unsigned int    ReadTimer3 (void);
unsigned char   ReadTimer4 (void);
```

The use of these functions is straightforward. The following example reads the 16-bit value of Timer1:

```
unsigned int time_val;
time_val = ReadTimer1();
```

The prototype definitions of library functions for writing values into timers are the following:

```
void WriteTimer0 (unsigned int timer);
void WriteTimer1 (unsigned int timer);
void WriteTimer2 (unsigned char timer);
void WriteTimer3 (unsigned int timer);
void WriteTimer4 (unsigned char timer);
```

where the parameter **timer** is the value to be written into the specified timer register. The following statement will write the value of 0x1535 into Timer0:

```
writeTimer0 (0x1535);
```

8.4 Capture/Compare/PWM Modules

A PIC18 device may have one or two or five capture/compare/PWM (CCP) modules. A CCP module contains a 16-bit capture register, a 16-bit compare register, or a PWM master/slave duty cycle register. A CCP module can be configured to operate in capture or compare or PWM mode. Each CCP module requires the use of timer resources. In capture or compare mode, the CCP module may need to use either Timer1 or Timer3 to operate. In PWM mode, either Timer2 or Timer4 may be needed.

The operations of all CCP modules are identical, with the exception of the special event trigger present on CCP1 and CCP2. The operation mode of the CCP module is configured by the CCPxCON (x = 1, 2, . . ., 5) register. The contents of these registers are shown in Figure 8.10.

	7	6	5	4	3	2	1	0
value after reset	--	--	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0
	0	0	0	0	0	0	0	0

DCxB1:DCxB0: PWM duty cycle bit 1 and bit 0 for CCP module x
capture mode:
unused
compare mode:
unused
PWM mode:
These two bits are the lsbs (bit 1 and bit 0) of the 10-bit PWM duty cycle.
CCPxM3:CCPxM0: CCP module x mode select bits
0000 = capture/compare/PWM disabled (resets CCPx module)
0001 = reserved
0010 = compare mode, toggle output on match (CCPxIF bit is set)
0100 = capture mode, every falling edge
0101 = capture mode, every rising edge
0110 = capture mode, every 4th rising edge
0111 = capture mode, every 16th rising edge
1000 = compare mode, initialize CCP pin low, on compare match force CCP pin high (CCPxIF bit is set)
1001 = compare mode, initialize CCP pin high, on compare match force CCP pin low (CCPxIF bit is set)
1010 = compare mode, generate software interrupt on compare match (CCP pin unaffected, CCPxIF bit is set).
1011 = compare mode, trigger special event (CCPxIF bit is set)
For CCP1 and CCP2: Timer1 or Timer3 is reset on event
For all other modules: CCPx pin is unaffected and is configured as an I/O port.
11xx = PWM mode

Figure 8.10 ■ CCPxCON register (x = 1,..,5) (redraw with permission of Microchip)

8.4.1 CCP Module Configuration

Each capture/compare/PWM module is associated with a control register (CCPxCON) and a data register (CCPRx). The data register in turn is comprised of two 8-bit registers: CCPRxL (low byte) and CCPRxH (high byte).

The CCP modules utilize Timer1, Timer2, Timer3, or Timer4, depending on the mode selected. Timer1 and Timer3 are available to modules in capture or compare mode, while Timer2 and Timer4 are available for modules in PWM mode.

The assignment of a particular timer to a module is determined by the timer-to-CCP enable bits (bit 6 and bit 3) in the T3CON register. Depending on the selected configuration, up to four timers may be active at once, with modules in the same configuration sharing the same timer resources. The possible configurations are shown in Figure 8.11.

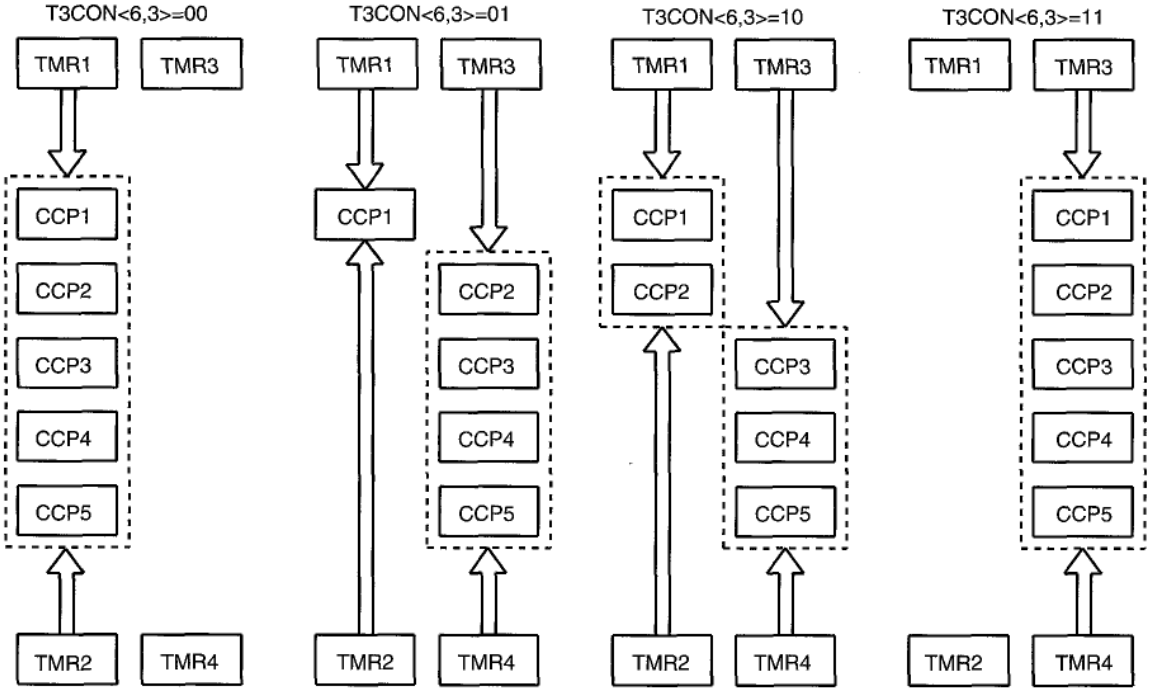

Figure 8.11 ■ CCP and Timer interconnect configurations (redraw with permission of Microchip)

In Figure 8.11,

when T3CON<6,3> = 00,

Timer1 is used for all capture and compare operations for all CCP modules. Timer2 is used for PWM operations for all CCP modules.

when T3CON<6,3> = 01,

Timer1 is used for the capture and compare operations for the CCP1 module, and Timer2 is used for the PWM operation for the CCP1 module. Timer3 is used for the

capture and compare operations for the remaining four CCP modules, and Timer4 is used for the PWM operations for the remaining four CCP modules.

when T3CON<6,3> = 10,

Timer1 is used for the capture and compare operations for the CCP1 and CCP2 modules, whereas Timer2 is used for the PWM operations for the CCP1 and CCP2 modules. Timer3 is used for the capture and compare operations for the CCP3, CCP4, and CCP5 modules. Timer4 is used for the PWM operations for CCP3-CCP5.

when T3CON<6,3> = 11,

Timer3 is used for the capture and compare operations for all CCP modules, whereas Timer4 is used for the PWM operations for all CCP modules.

8.5 CCP in Capture Mode

Some applications need to know the arrival times of events. In a computer, *physical time* is represented by the count value in a counter, whereas the occurrence of an *event* is represented by a signal edge (can be a rising or a falling edge). The time when an event occurs can be recorded by latching the count value when a signal edge arrives, as illustrated in Figure 8.12. The capture mode of the PIC18 CCP module is designed for this type of application. As shown in Figure 8.13,

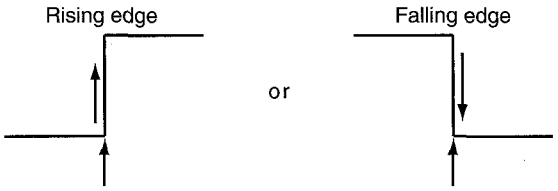

Figure 8.12 ■ Events represented by signal edges

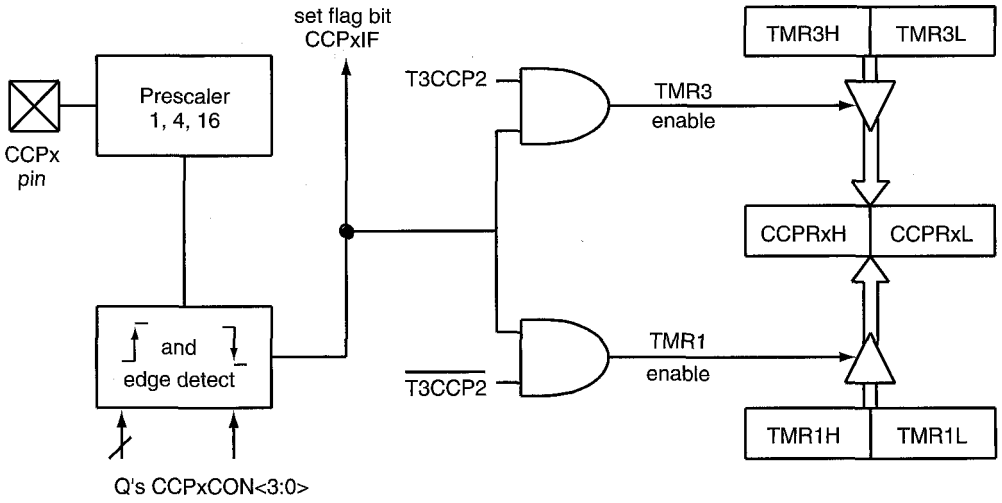

Figure 8.13 ■ Capture mode operation block diagram (redraw with permission of Microchip)

the 16-bit value of the TMR1 or the TMR3 register pair is captured into the register pair CCPRxH: CCPRxL when an event occurs on the CCPx pin. An event can be one of the following:

- Every falling edge
- Every rising edge
- Every 4th rising edge
- Every 16th rising edge

8.5.1 Capture Operation

When the capture is made, the interrupt request flag bit CCPxIF is set, which must be cleared in software. If another capture occurs before the value in the register CCPRx is read, the old value will be overwritten by the new captured value.

In capture mode, the CCPx pin must be configured as an input. The timers that are to be used with the capture operation (Timer1 and/or Timer3) must be running in the timer mode or synchronous counter mode. In the asynchronous counter mode, the capture operation may not work.

When the capture mode is changed, a false capture interrupt may be generated. The user should clear the CCPxIE bit prior to changing the capture mode to avoid false interrupts and should clear the flag bit CCPxIF following any such change in the operation mode.

There are four prescaler settings (one for falling edge capture and three for rising edge capture). Whenever the CCP module is turned off or the CCP module is not in the capture mode, the prescaler counter is cleared. Any reset will also clear the prescaler counter.

Switching from one capture prescaler to another may generate an interrupt. In addition, the prescaler counter will not be cleared during this switching. Therefore, the first capture may be from a nonzero prescaler. To prevent this, the user should turn off the CCP module before switching to a new capture prescaler.

8.5.2 Microchip C Library Functions for CCP in Capture Mode

Microchip MCC18 C compiler provides three functions (shown in Table 8.1) to support each CCP channel. The header file **capture.h** must be included in the user program to use these library functions.

Function	Description
CloseCapture*x*	Disable capture channel *x*
OpenCapture*x*	Configure capture channel *x*
ReadCapture*x*	Read a value from CCP channel *x*

Table 8.1 ■ MCC18 C library functions for CCP peripheral

The prototype declarations and parameter definitions are as follows:

void **CloseCapture1** (void);

void **CloseCapture2** (void);

void **CloseCapture3** (void);

void **CloseCapture4** (void);
void **CloseCapture5** (void);
void **OpenCapture1** (unsigned char **config**);
void **OpenCapture2** (unsigned char **config**);
void **OpenCapture3** (unsigned char **config**);
void **OpenCapture4** (unsigned char **config**);
void **OpenCapture5** (unsigned char **config**);

There are two values to be set for the parameter **config:** interrupt enabling and the edge to capture. There are two possible values for the CCP interrupt capture mode:

CAPTURE_INT_ON : interrupt enabled
CAPTURE_INT_OFF : interrupt disabled

There are four possible edges to be captured:

Cx_EVERY_FALL_EDGE : capture on every falling edge
Cx_EVERY_RISE_EDGE : capture on every rising edge
Cx_EVERY_4_RISE_EDGE : capture on every 4th rising edge
Cx_EVERY_16_RISE_EDGE : capture on every 16th rising edge

The following five functions read the values from the specified capture channel:

unsigned int **ReadCapture1** (void);
unsigned int **ReadCapture2** (void);
unsigned int **ReadCapture3** (void);
unsigned int **ReadCapture4** (void);
unsigned int **ReadCapture5** (void);

The following example configures CCP2 in capture mode that disables interrupt, capture on every 16th rising edge:

```
OpenCapture2 (CAPTURE_INT_OFF & C2_EVERY_16_RISE_EDGE);
```

8.5.3 Applications of Capture Mode

The capture mode of CCP module has many applications. Some examples are the following:

- *Event arrival time recording.* Some applications (e.g., a swimming competition) need to compare the arrival times of several different swimmers. The capture function is very suitable for this application. The number of events that can be compared is limited by the number of capture channels.
- *Period measurement.* To measure the period of an unknown signal, the capture function should be configured to capture the timer values corresponding to two consecutive rising or falling edges, as illustrated in Figure 8.14.

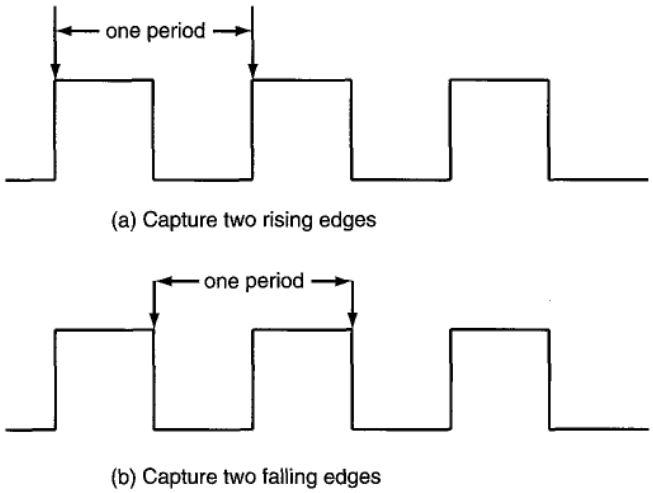

Figure 8.14 ■ Period measurment by capturing two consecutive edges

- *Pulse-width measurement.* To measure the width of a pulse, two adjacent rising and falling edges are captured, as shown in Figure 8.15.

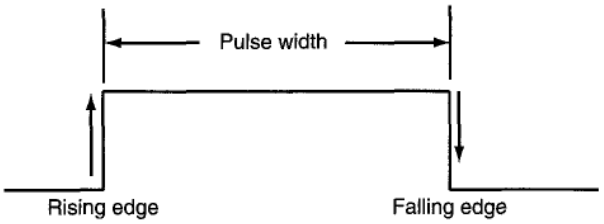

Figure 8.15 ■ Pulse-width measurement using input capture

- *Interrupt generation.* All capture inputs can serve as edge-sensitive interrupt sources. Once enabled, interrupt will be generated on the selected edge.
- *Event counting.* An event can be represented by a signal edge. A CCP channel in capture mode can be used in conjunction with a timer or another CCP channel in compare mode to count the number of events that occur during an interval. This application is similar to using a timer in counter mode.
- *Time reference.* In this application, a CCP channel in capture mode is used in conjunction with another CCP channel in compare mode. For example, if the user wants to activate a pulse at certain number of clock cycles after detecting an input event, the CCP channel in capture mode can be used to record the time at which the edge is detected. A number corresponding to the desired delay would be added to this captured value and stored in the CCPRx register of the CCPx channel in compare mode. This application is illustrated in Figure 8.16.

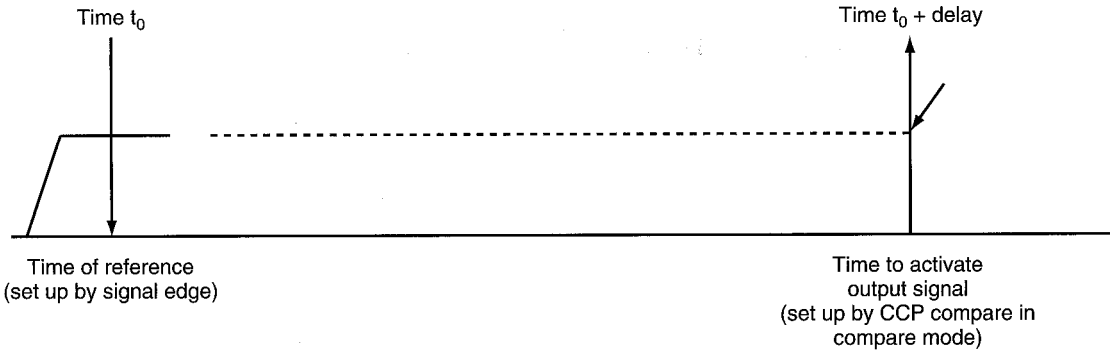

Figure 8.16 ■ A time reference application

- *Duty cycle measurement.* The duty cycle is the percentage of time that the signal is high within a period in a periodic digital signal. The measurement of duty cycle is illustrated in Figure 8.17.

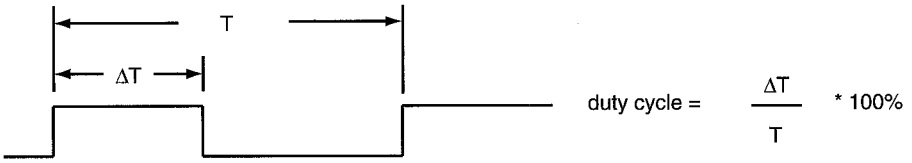

Figure 8.17 ■ Definition of duty cycle

The unit used in most of the measurements is the number of clock cycles. When it is desirable, the unit should be converted to an appropriate unit such as seconds.

Example 8.5

Period measurement. Use the CCP channel 1 in capture mode to measure the period of an unknown signal, assuming that the PIC18 microcontroller is running with a 16-MHz crystal oscillator. Use the number of clock cycles as the unit of period. The period of the unknown signal is shorter than 65536 clock cycles.

Solution: The user needs to capture either two consecutive rising or falling edges and calculate their difference in order to measure the period. The required settings are as follows:

- CCP1 (RC2) pin: Configured for input.
- Timer1: 16-bit operation, use instruction cycle clock as clock source, prescaler set to 1. Write the value 0x81 into the T1CON register.
- Timer3: Select Timer1 as the base timer for the CCP1 capture mode. Write the value 0x81 into the T3CON register.
- CCP1: Capture on every rising edge. Write the value 0x05 into the CCP1CON register.
- Disable the CCP1 interrupt. Clear the CCP1IE bit of the PIE1 register.

The assembly program that measures the period (placed in PRODH:PRODL) using this setting is as follows:

```
        #include <p18F8720.inc>
        org     0x00
        goto    start
        org     0x08
        retfie
        org     0x18
        retfie
start   bsf     TRISC,CCP1      ; configure CCP1 pin for input
        movlw   0x81            ; use Timer1 as the time base
        movwf   T3CON           ; of CCP1 capture
        bcf     PIE1,CCP1IE     ; disable CCP1 capture interrupt
        movlw   0x81            ; enable Timer1, prescaler set to 1,
        movwf   T1CON           ; 16-bit, use instruction cycle clock
        movlw   0x05            ; set CCP1 to capture on every rising edge
        movwf   CCP1CON         ; "
        bcf     PIR1,CCP1IF     ; clear the CCP1IF flag
edge1   btfss   PIR1,CCP1IF     ; wait for the first edge to arrive
        goto    edge1           ; "
        movff   CCPR1H,PRODH    ; save the first edge
        movff   CCPR1L,PRODL    ; "
        bcf     PIR1,CCP1IF     ; clear the CCP1IF flag
edge2   btfss   PIR1,CCP1IF     ; wait for the second edge to arrive
        goto    edge2           ; "
        clrf    CCP1CON         ; disable CCP1 capture
        movf    PRODL,W
        subwf   CCPR1L,W        ; subtract first edge from 2nd edge
        movwf   PRODL           ; and leave the period in PRODH:PRODL
        movf    PRODH,W         ; "
        subwfb  CCPR1H,W        ; "
        movwf   PRODH           ; "
forever goto    forever         ;
        end
```

The C language version of the program is as follows:

```
#include <p18F8720.h>
void main (void)
{
        unsigned int period;
        TRISCbits.TRISC2 = 1;           /* configure CCP1 pin for input */
        T3CON = 0x81;                   /* use Timer1 as the time base for CCP1 capture */
        PIE1bits.CCP1IE = 0;            /* disable CCP1 capture interrupt */
        PIR1bits.CCP1IF = 0;            /* clear the CCP1IF flag */
        T1CON = 0x81;                   /* enable 16-bit Timer1, prescaler set to 1 */
        CCP1CON = 0x05;                 /* capture on every rising edge */
        while (!(PIR1bits.CCP1IF));     /* wait for 1st rising edge */
        PIR1bits.CCP1IF = 0;
        period = CCPR1;                 /* save the first edge (CCPR1 is accessed as a 16-bit value) */
```

```
    while (!(PIR1bits.CCP1IF));        /* wait for the 2nd rising edge */
    CCP1CON = 0x00;                    /* disable CCP1 capture */
    period = CCPR1 - period;
}
```

The clock period of an unknown signal could be much longer than 2^{16} clock cycles. In this situation, the user will need to keep track of the number of times that the timer overflows. Each timer overflow adds 2^{16} clock cycles to the period. Let

ovcnt = timer overflow count
diff = the difference of two edges
edge1 = the captured time of the first edge
edge2 = the captured time of the second edge

The signal period can be calculated by the following equations:

Case 1: edge2 ≥ edge1
period = ovcnt × 2^{16} + diff
Case 2: edge1 > edge2
period = (ovcnt - 1) × 2^{16} + diff

In case 2, the timer overflows at least once even if the pulse width is shorter than $2^{16} - 1$ clock cycles. Therefore, we need to subtract 1 from the timer overflow count in order to get the correct result. The period is obtained by appending the difference of the two captured edges to the timer overflow count.

Example 8.6

Write a program to measure the period of a signal connected to the CCP1 (RC2) pin, assuming that the instruction clock is running at 5 MHz. Make the program more general so that it can also measure the period of a signal with very low frequency.

Solution: The logic flow of the program is illustrated in Figure 8.18. The assembly program that implements the algorithm shown in Figure 8.18 is as follows:

```
            #include <p18F8720.inc>
ov_cnt      set     0x00        ; timer overflow count
per_hi      set     0x01        ; high byte of edge difference
per_lo      set     0x02        ; low byte of edge difference
            org     0x00
            goto    start
            org     0x08
            goto    hi_pri_ISR  ; go to the high-priority service routine
            org     0x18
            retfie
start       clrf    ov_cnt      ; initialize overflow count by 1
            bcf     INTCON,GIE  ; disable all interrupts
            bsf     RCON,IPEN   ; enable priority interrupt
            bcf     PIR1,TMR1IF ; clear the TMR1IF flag
            bsf     IPR1,TMR1IP ; set Timer1 interrupt to high priority
            bsf     TRISC,CCP1  ; configure CCP1 pin for input
            movlw   0x81        ; use Timer1 as the time base
```

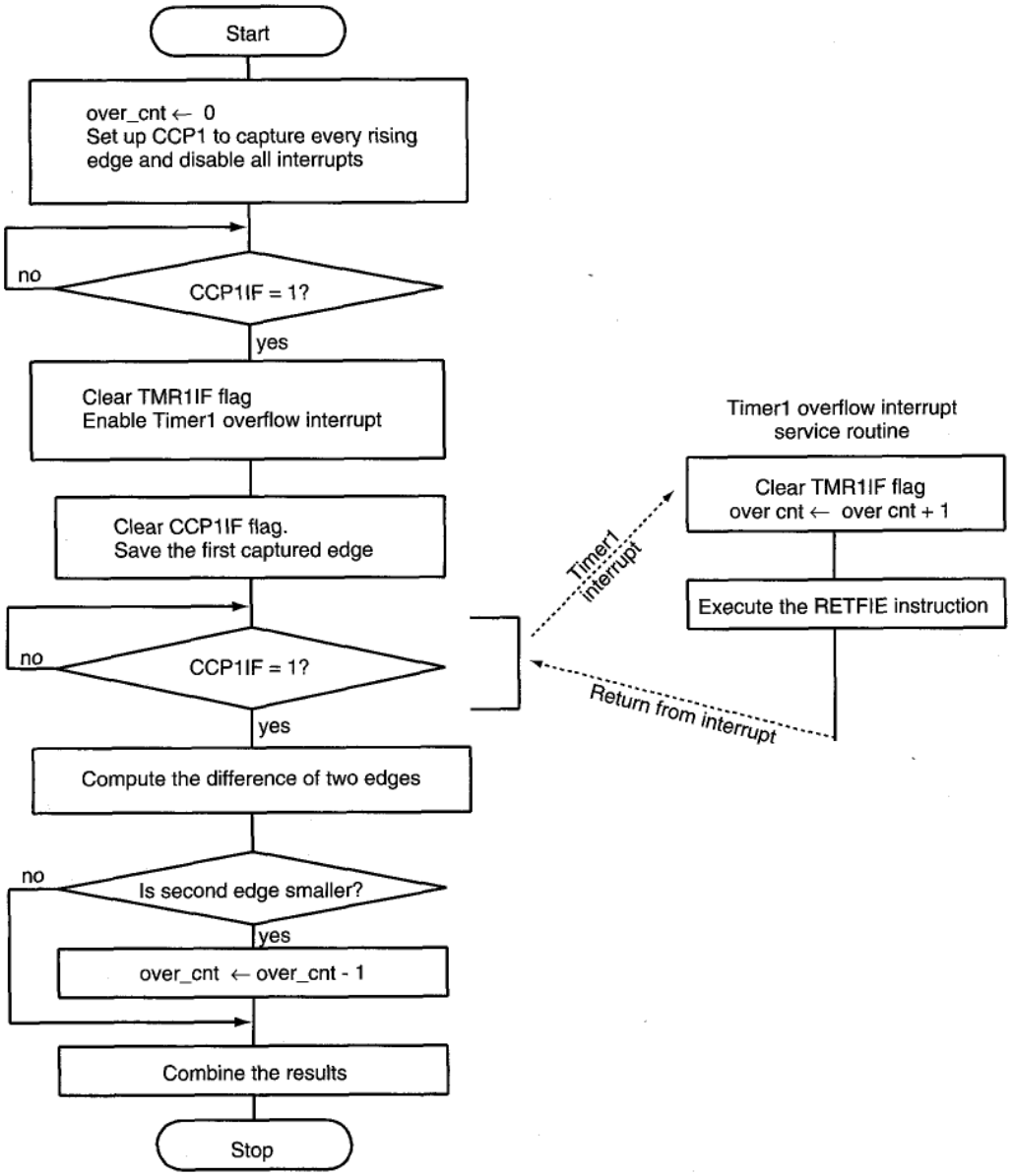

Figure 8.18 ■ Logic flow for measuring period of slow signals

```
movwf   T3CON           ; of CCP1 capture
bcf     PIE1,CCP1IE     ; disable CCP1 capture interrupt
movlw   0x81            ; enable Timer1, prescaler set to 1,
movwf   T1CON           ; 16-bit mode, use instruction cycle clock
movlw   0x05            ; set CCP1 to capture on every rising edge
movwf   CCP1CON         ; "
bcf     PIR1,CCP1IF     ; clear the CCP1IF flag
```

```
edge1     btfss    PIR1,CCP1IF      ; wait for the first edge to arrive
          goto     edge1            ; "
          movff    CCPR1H,per_hi    ; save the high byte of captured edge
          movff    CCPR1L,per_lo    ; save the low byte of captured edge
          bcf      PIR1,TMR1IF
          movlw    0xC0
          iorwf    INTCON,F         ; enable global interrupts
          bsf      PIE1,TMR1IE      ; enable Timer1 overflow interrupt
edge2     btfss    PIR1,CCP1IF      ; wait for the 2nd edge to arrive
          goto     edge2
          movf     per_lo,W
          subwf    CCPR1L,W
          movwf    per_lo           ; save the low byte of edge difference
          movf     per_hi,W
          subwfb   CCPR1H,W
          movwf    per_hi,A         ; save the high byte of edge difference
          btfsc    STATUS,C
          goto     normal
          decf     ov_cnt           ; 1st edge is larger, so decrement overflow count
normal    nop
forever   goto     forever
hi_pri_ISR btfss   PIR1,TMR1IF      ; high priority interrupt service routine
          retfie                    ; not Timer1 interrupt, so return
          incf     ov_cnt
          bcf      PIR1,TMR1IF      ; clear Timer1 overflow interrupt flag
          retfie
          end
```

Using one byte to keep track of the times that Timer1 has rolled over can measure a period of length up to $2^{24} \times 10^{-6} \times 0.2 = 3.355$ seconds. To measure an even slower signal, two bytes are needed to keep track of the Timer1 rollover count.

The C language version of the program is as follows:

```
#include <p18F8720.h>
#include <timers.h>
#include <capture.h>
unsigned int ov_cnt, temp;
unsigned short long period;              /* 24-bit period value */
void high_ISR(void);
void low_ISR(void);
#pragma code high_vector = 0x08          // force the following statement to
void high_interrupt (void)               // start at 0x08
{
        _asm
        goto high_ISR
        _endasm
}
#pragma code low_vector = 0x18           //force the following statements to start at
void low_interrupt (void)                //0x18
{
        _asm
        goto        low_ISR
```

```
        _endasm
}
#pragma code                        //return to the default code section
#pragma interrupt high_ISR
void high_ISR (void)
{
        if(PIR1bits.TMR1IF){
                PIR1bits.TMR1IF = 0;
                ov_cnt ++;
        }
}
#pragma code
#pragma interrupt low_ISR
void low_ISR (void)
{
        _asm
        retfie 0
        _endasm
}

void main (void)
{
        unsigned int temp1;
        ov_cnt = 0;
        INTCONbits.GIE = 0;         /* disable global interrupts */
        RCONbits.IPEN = 1;          /* enable priority interrupts */
        PIR1bits.TMR1IF = 0;
        IPR1bits.TMR1IP = 1;        /* promote Timer1 rollover interrupt to high priority */
        TRISCbits.TRISC2 = 1;       /* configure CCP1 pin for input */
        OpenTimer1 (TIMER_INT_ON & T1_16BIT_RW & T1_PS_1_1 &
                T1_OSC1EN_OFF & T1_SYNC_EXT_OFF &
                T1_SOURCE_INT);
        OpenTimer3 (TIMER_INT_OFF & T3_16BIT_RW & T3_PS_1_1 &
                T3_SOURCE_INT & T3_PS_1_1 & T3_SYNC_EXT_ON & T12_SOURCE_CCP);
                /* turn on Timer3 and appropriate parameters */
        OpenCapture1 (CAPTURE_INT_OFF & C1_EVERY_RISE_EDGE);
        PIE1bits.CCP1IE = 0;        /* disable CCP1 capture interrupt */
        PIR1bits.CCP1IF = 0;
        while(!(PIR1bits.CCP1IF));
        temp = ReadCapture1( );     /* save the first captured edge */
        PIR1bits.CCP1IF = 0;
        PIR1bits.TMR1IF = 0;
        INTCON |= 0xC0;             /* enable global interrupts */
        PIE1bits.TMR1IE = 1;        /* enable Timer1 rollover interrupt */
        while(!(PIR1bits.CCP1IF));
        CloseCapture1();            /* disable CCP1 capture */
        temp1 = ReadCapture1();
        if (temp1 < temp)
                ov_cnt--;
        period = ov_cnt * 65536 + temp1 - temp;
}
```

The algorithm for measuring the pulse width is similar to that for measuring the duty cycle and hence is left for you as an exercise problem.

When the signal frequency is very high, it is likely that the next edge will arrive even before the captured edge has been saved. In this situation, the user can choose to capture every 4th or even every 16th rising edge in order to measure the period of the unknown signal. The actual period is simply the difference of two captured edges divided by 4 or 16.

8.6 CCP in Compare Mode

In compare mode, the 16-bit CCPRx register value is constantly compared against either the TMR1 register pair value or the TMR3 register pair value. When a match occurs, one of the following actions may occur on the associated CCPx pin:

- Driven high
- Driven low
- Toggle output (high to low or low to high)
- Remains unchanged

8.6.1 Compare Mode Operation

The circuit related to the CCP compare mode operation is shown in Figure 8.19. In order to use the CCP in compare mode, the user must do the following:

1. Make a copy of the 16-bit timer value
2. Add to this copy a **delay count**
3. Store the sum in the CCPRxH:CCPRxL register pair

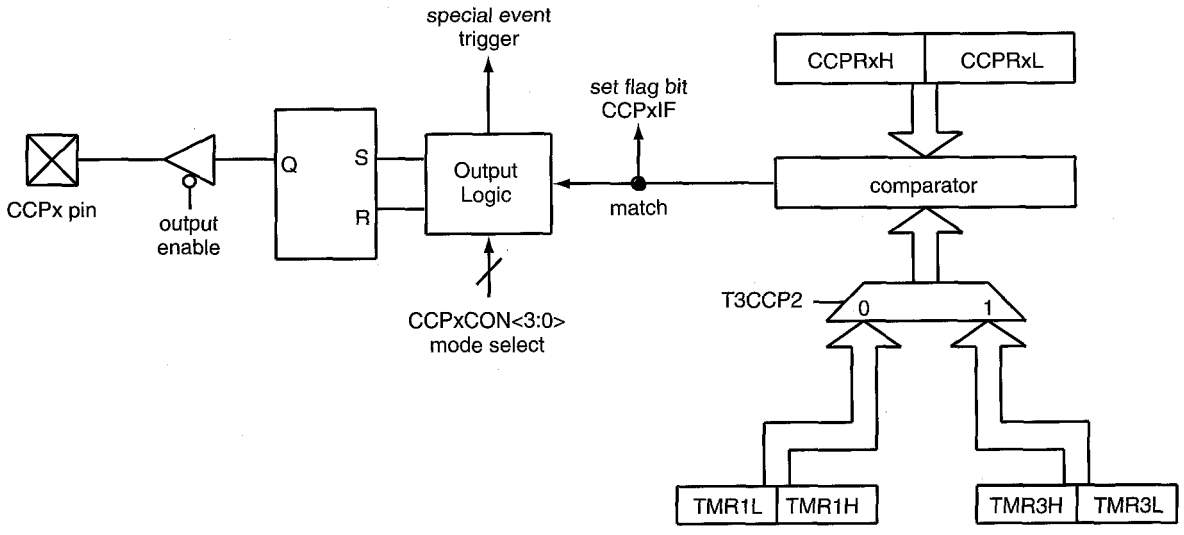

Figure 8.19 ■ Circuit for CCP in compare mode (redraw with permission of Microchip)

The CCPx pin must be configured for output and Timer1 and/or Timer3 must be running in timer mode or synchronized counter mode.

The CCP1 and the CCP2 modules in compare mode can also generate the special event trigger. The output of either the CCP1 or the CCP2 module resets the TMR1 or TMR3 register pair, depending on which timer resource is currently selected. This allows the CCPR1 register to effectively be a 16-bit programmable period register for Timer1 or Timer3. The CCP2 special event trigger will also start an A/D conversion if the A/D module is enabled.

8.6.2 Applications of CCP in Compare Mode

CCP in compare mode can be used to perform a variety of functions. Generation of a single pulse, a train of pulses, a periodic waveform with a certain duty cycle, and a specific time delay are among the most popular applications.

Example 8.7

Use CCP1 to generate a periodic waveform with 40% duty cycle and 1-KHz frequency, assuming that the instruction cycle clock frequency is 4 MHz.

Solution: The waveform of a 1-KHz digital signal with a 40% duty cycle is shown in Figure 8.20.

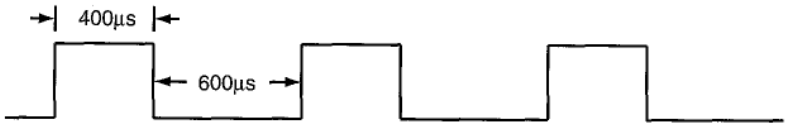

Figure 8.20 ■ 1KHz 40% duty cycle waveform

The logic flow of this problem is illustrated in Figure 8.21.Suppose we use Timer3 as the base timer and set the prescale factor of Timer3 to 1. The number of clock cycles that the signal is high and low will be 1600 and 2400, respectively.

The assembly program to implement the algorithm illustrated in Figure 8.21 is as follows:

```
        #include <p18F8720.inc>
hi_hi   equ     0x06        ; number (1600) of clock cycles that signal
hi_lo   equ     0x40        ; is high
lo_hi   equ     0x09        ; number (2400) of clock cycles that signal
lo_lo   equ     0x60        ; is low
        org     0x00
        goto    start
        org     0x08
        retfie
        org     0x18
        retfie
start   bcf     TRISC,CCP1  ; configure CCP1 pin for output
        movlw   0xC9        ; Configure Timer3 for 16-bit timer, prescaler 1,
        movwf   T3CON       ; use Timer3 for CCP1 time base, enable Timer3
        bcf     PIR1,CCP1IF ; clear the CCP1IF flag
        movlw   0x09        ; CCP1 pin set high initially and
        movwf   CCP1CON     ; pull low on match
```

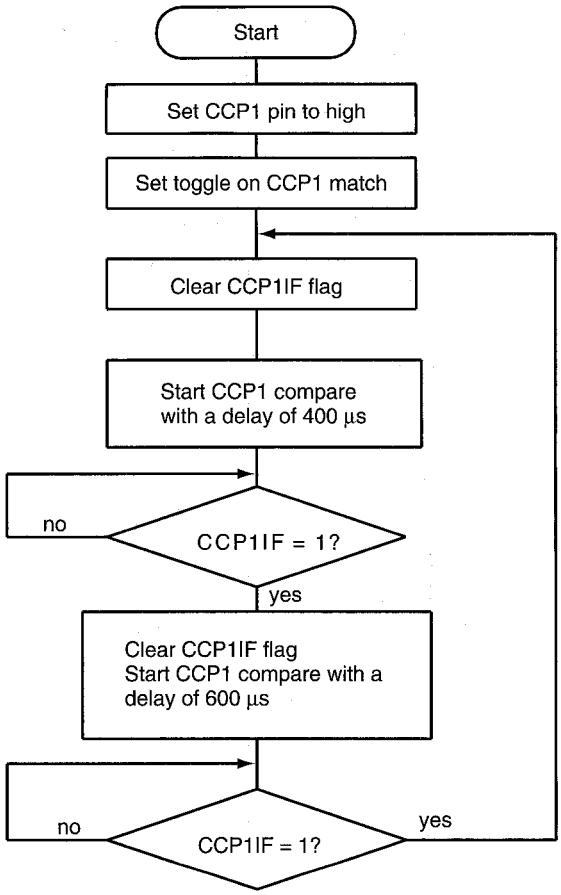

Figure 8.21 ■ The program logic flow for digital waveform generation

```
;
; CCPR1 <-- TMR3 + 1600; start a new compare operation
;
            movlw     hi_lo
            addwf     TMR3L,W
            movwf     CCPR1L          ; "
            movlw     hi_hi           ; "
            addwfc    TMR3H,W         ; "
            movwf     CCPR1H          ; save it in CCPR1 register pair
            bcf       PIR1,CCP1IF     ; clear the CCP1IF flag
hi_time     btfss     PIR1,CCP1IF
            bra       hi_time
            bcf       PIR1,CCP1IF
            movlw     0x08            ; CCP1 pin set low initially and
            movwf     CCP1CON         ; pull high on match
```

```
;
; CCPR1 <-- CCPR1 + 2400; start another compare operation
;
            movlw     lo_lo
            addwf     CCPR1L,F        ;
            movlw     lo_hi
            addwfc    CCPR1H,F
lo_time     btfss     PIR1,CCP1IF
            bra       lo_time
            bcf       PIR1,CCP1IF
            movlw     0x09            ; CCP1 pin set high initially
            movwf     CCP1CON         ; and pull low on match
            movlw     hi_lo
            addwf     CCPR1L,F
            movlw     hi_hi
            addwfc    CCPR1H,F
            bra       hi_time
            end
```

The C language version of the program is as follows:

```
#include <p18F8720.h>
void main (void)
{
      TRISCbits.TRISC2 = 0;              /* configure CCP1 pin for output */
      T3CON = 0xC9;                      /* turn on TMR3 in 16-bit mode, prescaler is 1 */
      CCP1CON = 0x09;                    /* configure CCP1 pin to set high initially but pull low on match*/
      CCPR1 = TMR3 + 0x0640;             /* start CCP1 compare with delay equals 1600*/
      PIR1bits.CCP1IF = 0;               /* clear CCP1IF flag */
      while (1) {
            while (!(PIR1bits.CCP1IF));
            PIR1bits.CCP1IF = 0;
            CCP1CON = 0x08;
            CCPR1 += 0x0960;             /* start CCP1 compare with delay equals 2400*/
            while (!(PIR1bits.CCP1IF));
            PIR1bits.CCP1IF = 0;
            CCP1CON = 0x09;
            CCPR1 += 0x0640;             /* start CCP1 compare with delay equals 1600*/
      }
}
```

In this example, the CPU is tied up in generating the waveform. A better approach is to use interrupt. After starting the first pulse, the user program enables the CCP interrupt. The CCP interrupt service routine simply clears the interrupt flag and restarts the next compare operation. By using this approach, the CPU can still perform other useful operations. The following example illustrates this idea.

Example 8.8

Use the interrupt-driven approach to generate the waveform specified in Example 8.6.

Solution: This program will use a flag to select either 1600 (= 0) or 2400 (= 1) as the delay count for the compare operation. We will configure the CCP1 interrupt to high priority. The CCP1

interrupt service routine will check to make sure that the interrupt is caused by the compare match. If not, the service routine simply returns. If the interrupt is caused by the CCP1 compare match, the service routine will start a new compare operation with 1600 (when flag = 0) or 2400 (when flag = 1) as the delay. The assembly program is as follows:

```
#include <p18F8720.inc>
hi_hi     equ     0x06          ; number (1600) of clock cycles that signal
hi_lo     equ     0x40          ; is high
lo_hi     equ     0x09          ; number (2400) of clock cycles that signal
lo_lo     equ     0x60          ; is low
flag      set     0x00          ; select 1600 (=0) or 2400(=1) as delay
          org     0x00
          goto    start
          org     0x08
          goto    hi_ISR        ; go to the actual interrupt service routine
          org     0x18
          retfie
start     bcf     TRISC,CCP1    ; configure CCP1 pin for output
          movlw   0xC9          ; Enable Timer3 in 16-bit mode, prescaler 1,
          movwf   T3CON         ; use Timer3 as the base timer of CCP1
          bsf     PORTC,CCP1    ; pull the CCP1 pin to high
          movlw   0x09          ; configure CCP1 pin to set high initially and
          movwf   CCP1CON       ; pull low on match
;
; start a compare operation so that the CCP1 pin stay high for 400 us
          movlw   hi_lo         ; "
          addwf   TMR3L,W       ; "
          movwf   CCPR1L        ; "
          movlw   hi_hi         ; "
          addwfc  TMR3H,W       ; "
          movwf   CCPR1H        ; "
          bcf     PIR1,CCP1IF   ; clear the CCP1IF flag
hi_LST    btfss   P1R1,CPP1IF
          bra     hi_lst
          bcf     P1R1,CCP1IF
          movlw   0x02          ; CCP1 pin
          movwf   CCP1CON       ; toggle on match
          movlw   lo_lo         ; start next
          addwf   CCPR1L,F      ; compare
          movlw   lo_hi         ; operation
          addwfc  CCPR1H,F      ; "
          bsf     IPR1,CCP1IP   ; configure CCP1 interrupt to high priority
          movlw   0x00          ; clear the flag so that 1600 is the next delay
          movwf   flag          ; for CCP compare operation
          movlw   0xC0
          iorwf   INTCON,F      ; enable CCP1 interrupt
          bsf     PIE1,CCP1IE   ; "
forever   nop                   ; wait for interrupt
          goto    forever       ; "
hi_ISR    btfss   PIR1,CCP1IF   ; is the interrupt caused by CCP1?
          retfie
```

```
            bcf     PIR1,CCP1IF
            btfsc   flag,0          ; prepare to add 1600 if flag is 0
            goto    add_2400
            movlw   hi_lo           ; start a new compare operation
            addwf   CCPR1L,F        ; which will keep CCP1 pin high for
            movlw   hi_hi           ; 1600 clock cycles
            addwfc  CCPR1H,F        ; "
            btg     flag,0          ; toggle the flag
            retfie
add_2400    movlw   lo_lo           ; start a new compare operation which
            addwf   CCPR1L,F        ; will keep CCP1 pin low for 2400
            movlw   lo_hi           ; clock cycles
            addwfc  CCPR1H,F        ; "
            btg     flag,0          ; toggle the flag
            retfie
            end
```

The C language version of the program is straightforward and hence is left for you as an exercise problem.

The CCP in compare mode can be used to create a time delay. The following example illustrates this application.

Example 8.9

Assume that there is a PIC18 demo board (e.g., SSE8720) running with a 16-MHz crystal oscillator. Write a function that uses CCP1 in compare mode to create a time delay that is equal to 10 ms multiplied by the contents of PRODL.

Solution: The function uses a program loop to perform the number (specified in the PRODL register) of compare operations on CCP1. A 10-ms delay can be created by the following:

- Choosing instruction cycle clock as the clock source
- Setting the prescaler of Timer3 to 1
- Using 40000 as the delay count of the compare operation
- Using Timer1 as the base timer for CCP1 through CCP5

The assembly language version of the function is as follows:

```
; ************************************************************
; The following function creates a delay that is equal to the content of PRODL
; multiplied by 10 ms assuming that the PIC18 MCU uses a 16 MHz crystal oscillator.
; ************************************************************
dly_by10ms  movlw   0x81            ; enable Timer3 in 16-bit mode, 1:1 prescaler
            movwf   T3CON,A         ; use Timer1 as base times for CCP1
            movwf   T1CON,A         ; enable Timer1 in 16-bit mode with 1:1 prescaler
            movlw   0x0A            ; configure CCP1 to generate software
            movwf   CCP1CON,A       ; interrupt on compare match
            movf    TMR1L,W,A       ; to perform a CCP1 compare with
            addlw   0x40            ; 40000 cycles of delay
            movwf   CCPR1L,A        ; "
```

```
                movlw   0x9C            ; "
                addwfc  TMR1H,W,A       ; "
                movwf   CCPR1H,A        ; "
                bcf     PIR1,CCP1IF,A
loop            btfss   PIR1,CCP1IF,A   ; wait until 40000 cycles are over
                bra     loop
                dcfsnz  PRODL,F,A       ; is loop count decremented to zero yet?
                return  0               ; delay is over, return
                bcf     PIR1,CCP1IF,A   ; clear the CCP1IF flag
                movlw   0x40            ; start the next compare operation
                addwf   CCPR1L,F,A      ; with 40000 cycles delay
                movlw   0x9C            ; "
                addwfc  CCPR1H,F,A      ; "
                bra     loop
```

The C language version of the function is as follows:

```
/* ------------------------------------*/
/* The following function creates a delay equals to 10ms */
/* times kk.                                              */
/* ------------------------------------*/
void dly_by10ms (unsigned char kk)
{
        CCP1CON         = 0x0A; /* configure CCPR1 to generate software interrupt */
        T3CON           = 0x81; /* enables Timer3 and select Timer1 as base timer for CCP1 */
        T1CON           = 0x81; /* enables Timer1 in 16-bit mode with 1:1 as the prescaler */
        CCPR1           = TMR1 + 40000; /* start compare operation with 40000 delay */
        PIR1bits.CCP1IF = 0;
        while (kk){
                while (!PIR1bits.CCP1IF);
                PIR1bits.CCP1IF = 0;
                kk--;
                CCPR1 += 40000;
        }
        return;
}
```

Using the CCP module (in compare mode) to make sound is easy. The basic idea is to create a digital waveform of appropriate frequency and use it to drive a speaker. A small speaker of 8-Ω resistance that consumes between 10 mW and 20 mW can be driven directly by the CCP output. The user can purchase such a speaker from Radio Shack or any electronic retailer. The next example illustrates how to use CCP1 in compare mode to generate a siren.

Example 8.10

Describe the circuit connection for siren generation and write a program that uses CCP1 in compare mode to generate a siren that oscillates between 440 Hz and 880 Hz.

Solution: A simple 8-Ω speaker has two terminals: one terminal is for signal input, whereas the other terminal is for ground connection. The circuit connection for siren generation is shown in Figure 8.22.

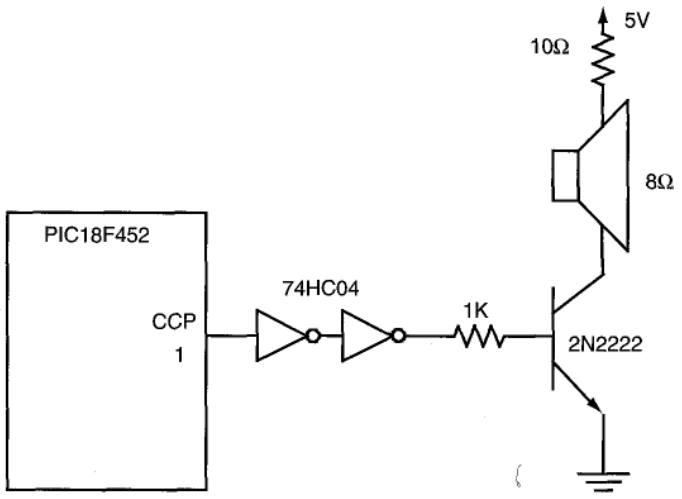

Figure 8.22 ■ Circuit connection for siren generation

The algorithm for generating the siren is as follows:

Step 1
Configure an appropriate CCP channel (CCP1 in this example) to operate in the compare mode and toggle output on match.

Step 2
Start a compare operation and enable its interrupt with a delay equal to half the period of the sound of the siren.

Step 3
Wait for certain amount of time (say, half of a second). During the waiting period, interrupts will be requested by the CCP compare match many times. The interrupt service routine simply clears the CCP flag and starts the next compare operation and then returns.

Step 4
At the end of the delay, choose a different delay time for the compare operation so that the siren sound with different frequency can be generated.

Step 5
Wait for the same amount of time as in Step 3. Again, interrupts caused by CCP compare match will be requested many times.

Step 6
Go to Step 2.

The assembly program that implements this algorithm for a PIC18F452 running with a 32-MHz crystal oscillator is as follows:

```
        #include <p18F452.inc>
hi_hi   equ     0x02        ; delay count to create 880 Hz sound
hi_lo   equ     0x38        ; "
lo_hi   equ     0x04        ; delay count to create 440 Hz sound
lo_lo   equ     0x70        ; "
        org     0x00
        goto    start
        org     0x08
```

```
            goto        hi_ISR
            org         0x18
            retfie
start       bcf         TRISC,CCP1      ; configure CCP1 pin for output
            movlw       0x81            ; Enable Timer3 for 16-bit mode, use
            movwf       T3CON           ; Timer1 as the base timer of CCP1
            movlw       0xB1            ; enables Timer1 for 16-bit, prescaler set to 1:8
            movwf       T1CON           ; "
            movlw       0x02            ; CCP1 pin toggle on match
            movwf       CCP1CON         ; "
            bsf         RCON,IPEN       ; enable priority interrupt
            bsf         IPR1,CCP1IP     ; configure CCP1 interrupt to high priority
            movlw       hi_hi           ; load delay count for compare operation
            movwf       PRODH           ; into PRODH:PRODL register pair
            movlw       hi_lo           ; "
            movwf       PRODL           ; "
            movlw       0xC0
            iorwf       INTCON,F        ; set GIE & PIE bits
            movf        PRODL,W         ; start a new compare operation with
            addwf       TMR1L,W         ; delay stored in PRODH:PRODL
            movwf       CCPR1L          ; "
            movf        PRODH,W         ; "
            addwfc      TMR1H,W         ; "
            movwf       CCPR1H          ; "
            bcf         PIR1,CCP1IF     ; clear CCP1IF flag
            bsf         PIE1,CCP1IE     ; enable CCP1 interrupt
forever     call        delay_hsec      ; stay for half second in one frequency
            movlw       lo_hi           ; switch to different frequency
            movwf       PRODH,A         ; "
            movlw       lo_lo           ; "
            movwf       PRODL,A         ; "
            call        delay_hsec      ; stay for half second in another frequency
            movlw       hi_hi           ; switch to different frequency
            movwf       PRODH,A         ; "
            movlw       hi_lo           ; "
            movwf       PRODL,A         ; "
            goto        forever
hi_ISR      bcf         PIR1,CCP1IF     ; clear the CCP1IF flag
            movf        PRODL,W         ; start the next compare operation
            addwf       CCPR1L,F        ; using the delay stored in PRODH:PRODL
            movf        PRODH,W         ; "
            addwfc      CCPR1H,F        ; "
            retfie      fast
;*****************************************************************
; The following routine uses Timer 0 to create a delay of half a second.
; By setting the prescale factor to 64 and let Timer0 to count up from 3035 (0x0BDB),
; Timer0 will overflow in 62500 clock cycles (0.5 second).
;*****************************************************************
delay_hsec  movlw       0x0B
            movwf       TMR0H
```

```
        movlw   0xDB
        movwf   TMR0L
        movlw   0x85            ; enable TMR0, select instruction clock,
        movwf   T0CON           ; prescaler set to 64
        bcf     INTCON,TMR0IF
loopw   btfss   INTCON,TMR0IF
        bra     loopw           ; wait for a half second
        return
        end
```

The C language version of the program is straightforward and hence is left for you as an exercise problem. The previous program can be modified to be run on a demo board with a crystal oscillator running at different frequencies.

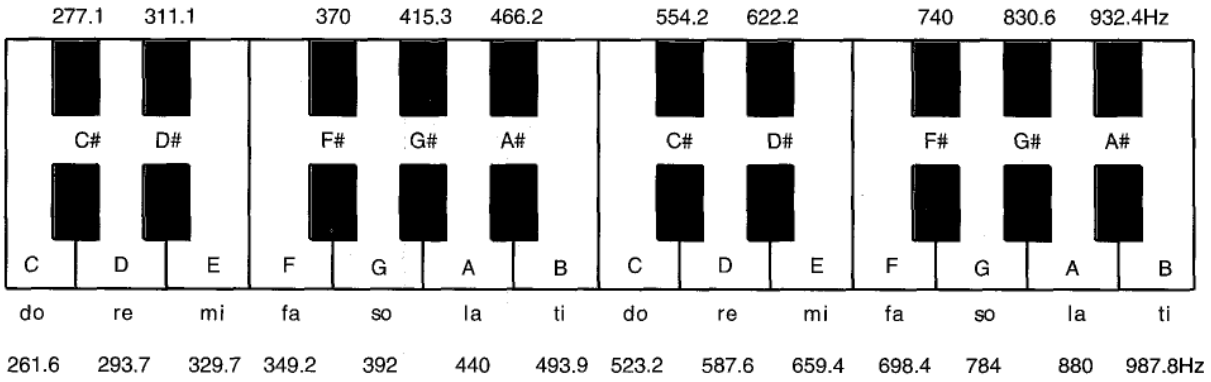

Figure 8.23 ■ Frequencies of music notes.

A siren can be considered as a song with only two notes. The frequencies of sounds over two octaves along with the black keys on a piano are shown in Figure 8.23. In general, a quarter note lasts for about 0.4 seconds (150 quarter notes per minute). The duration of other notes can be derived proportionally.

To play a song from the speaker, one places the frequencies and durations of all the notes in a music score in a table. For every note, the user program uses the CCP in compare mode to generate the digital waveform with the specified frequency and duration. The following example uses CCP1 in compare mode to play a song using a speaker. The frequencies of all the notes available on the piano are listed in Appendix E.

Example 8.11

For the circuit shown in Figure 8.22, write a C program to generate a simple song assuming that the demo board is running with a 4 MHz crystal oscillator.

Solution: The sample song to be played is a folk song. The following C program places in two tables (1) numbers to be added to the CCPR1 register to generate the waveform with the desired frequency for each note and (2) numbers that select the duration for each note:

```
#include    <P18F452.h>
#define     base    3125        /* counter count to create 0.1s delay */
```

```
#define     NOTES     38              /* 38 notes in the song to be played */
unsigned int half_cyc = 0;
unsigned rom int per_arr[38] = {
                    0x777, 0x470, 0x4FC, 0x470, 0x598, 0x777, 0x777, 0x3BC,
                    0x3F4, 0x3BC, 0x470, 0x470, 0x598, 0x353, 0x353, 0x353,
                    0x3BC, 0x470, 0x3BC, 0x3BC, 0x3F4, 0x470, 0x3F4, 0x3BC,
                    0x470, 0x598, 0x353, 0x2CC, 0x353, 0x3BC, 0x470, 0x3BC,
                    0x3BC, 0x3F4, 0x470, 0x3F4, 0x3BC, 0x470};

unsigned rom char wait[38] = {
                    3, 5, 3, 3, 5, 3, 3, 5,
                    3, 3, 5, 3, 3, 5, 3, 3,
                    5, 3, 3, 3, 2, 2, 3, 3,
                    6, 3, 5, 3, 3, 5, 3, 3,
                    3, 2, 2, 3, 3, 6};
void delay (unsigned char xc);
void high_ISR(void);
void low_ISR(void);
#pragma code high_vector = 0x08         // force the following statement to
void high_interrupt (void)              // start at 0x08
{
        _asm
        goto high_ISR
        _endasm
}
#pragma code low_vector = 0x18          //force the following statements to start at
void low_interrupt (void)               //0x18
{
        _asm
        goto    low_ISR
        _endasm
}
#pragma code                            //return to the default code section
#pragma interrupt high_ISR
void high_ISR (void)
{
        if(PIR1bits.CCP1IF){
                PIR1bits.CCP1IF = 0;
                CCPR1 + = half_cyc;
        }
}
#pragma code
#pragma interrupt low_ISR
void low_ISR (void)
{
        _asm
        retfie 0
        _endasm
}

void main (void)
```

```
{
        int i, j;
        TRISCbits.TRISC2 = 0; /* configure CCP1 pin for output */
        T3CON = 0x81; /* enables Timer3 in 16-bit mode, Timer1 for CCP1 time base */
        T1CON = 0x81; /* enable Timer1 in 16-bit mode */
        CCP1CON = 0x02; /* CCP1 compare mode, pin toggle on match */
        IPR1bits.CCP1IP = 1; /* set CCP1 interrupt to high priority */
        PIR1bits.CCP1IF = 0; /* clear CCP1IF flag */
        PIE1bits.CCP1IE = 1; /* enable CCP1 interrupt */
        INTCON |= 0xC0; /* enable high priority interrupt */
        for (j = 0; j < 3; j++) { /*play the song for times */
                i = 0;
                half_cyc = per_arr[0];
                CCPR1 = TMR1 + half_cyc;
                while (i < NOTES) {
                        half_cyc = per_arr[i]; /* get the cycle count for half period of the note */
                        delay (wait[i]);      /* stay for the duration of the note */
                        i++;
                }
                INTCON &= 0x3F; /* disable interrupt */
                delay (5);
                delay (6);
                INTCON |= 0xC0; /* re-enable interrupt */
        }
}
/* ------------------------------------------------------------*/
/* The following function runs on a PIC18 demo board running with a 4 MHz crystal  */
/* oscillator. The parameter xc specifies the amount of delay to be created        */
/* ------------------------------------------------------------*/
void delay (unsigned char xc)
{
        switch (xc){
        case 1:         /* create 0.1 second delay (sixteenth note) */
                T0CON = 0x84; /* enable TMR0 with prescaler set to 32 */
                TMR0 = 0xFFFF - base; /* set TMR0 to this value so it overflows in 0.1 second */
                INTCONbits.TMR0IF = 0;
                while (!INTCONbits.TMR0IF);
                break;
        case 2:         /* create 0.2 second delay (eighth note) */
                T0CON = 0x84; /* set prescaler to Timer0 to 32 */
                TMR0 = 0xFFFF - 2*base; /* set TMR0 to this value so it overflows in 0.2 second */
                INTCONbits.TMR0IF = 0;
                while (!INTCONbits.TMR0IF);
                break;
        case 3:         /* create 0.4 seconds delay (quarter note) */
                T0CON = 0x84; /* set prescaler to Timer0 to 32 */
                TMR0 = 0xFFFF - 4*base; /* set TMR0 to this value so it overflows in 0.4 second */
                INTCONbits.TMR0IF = 0;
                while (!INTCONbits.TMR0IF);
```

```
            break;
        case 4:     /* create 0.6 s delay (3 eighths note) */
            TOCON = 0x84; /* set prescaler to Timer0 to 32 */
            TMR0 = 0xFFFF - 6*base; /* set TMR0 to this value so it overflows in 0.6 second */
            INTCONbits.TMR0IF = 0;
            while (!INTCONbits.TMR0IF);
            break;
        case 5:     /* create 0.8 s delay (half note) */
            TOCON = 0x84; /* set prescaler to Timer0 to 32 */
            TMR0 = 0xFFFF - 8*base; /* set TMR0 to this value so it overflows in 0.8 second */
            INTCONbits.TMR0IF = 0;
            while (!INTCONbits.TMR0IF);
            break;
        case 6:     /* create 1.2 second delay (3 quarter note) */
            TOCON = 0x84; /* set prescaler to Timer0 to 32 */
            TMR0 = 0xFFFF - 12*base; /* set TMR0 to this value so it overflows in 1.2 second */
            INTCONbits.TMR0IF = 0;
            while (!INTCONbits.TMR0IF);
            break;
        case 7:     /* create 1.6 second delay (full note) */
            TOCON = 0x84; /* set prescaler to Timer0 to 32 */
            TMR0 = 0xFFFF - 16*base; /* set TMR0 to this value so it overflows in 1.6 second */
            INTCONbits.TMR0IF = 0;
            while (!INTCONbits.TMR0IF);
            break;
            default:
            break;
    }
}
```

Other songs can be created by changing the tables **per_arr[]** and **wait[]** to appropriate values.

8.7 CCP in PWM Mode

In PWM mode, the CCPx pin can output a 10-bit resolution periodic digital waveform with programmable period and duty cycle. To operate in PWM mode, a CCPx pin must be configured for output. A simplified block diagram of the CCP module in PWM mode is shown in Figure 8.24.

The duty cycle of the waveform to be generated is a 10-bit value of which the upper eight bits are stored in the CCPRxH register, whereas the lowest two bits are stored in bit 5 and bit 4 of the CCPxCON register. The duty cycle value is compared with TMRy (y = 2 or 4) cascaded with the 2-bit Q clocks in every instruction clock cycle. Whenever these two values are equal, the CCPx pin is pulled to low. The TMRy register is also compared with the PRy register in every clock cycle. Whenever these two registers are equal, the following three events occur on the next increment cycle:

- The CCPx pin is pulled high (exception: if PWM duty cycle is 0%, the CCPx pin will not be set).

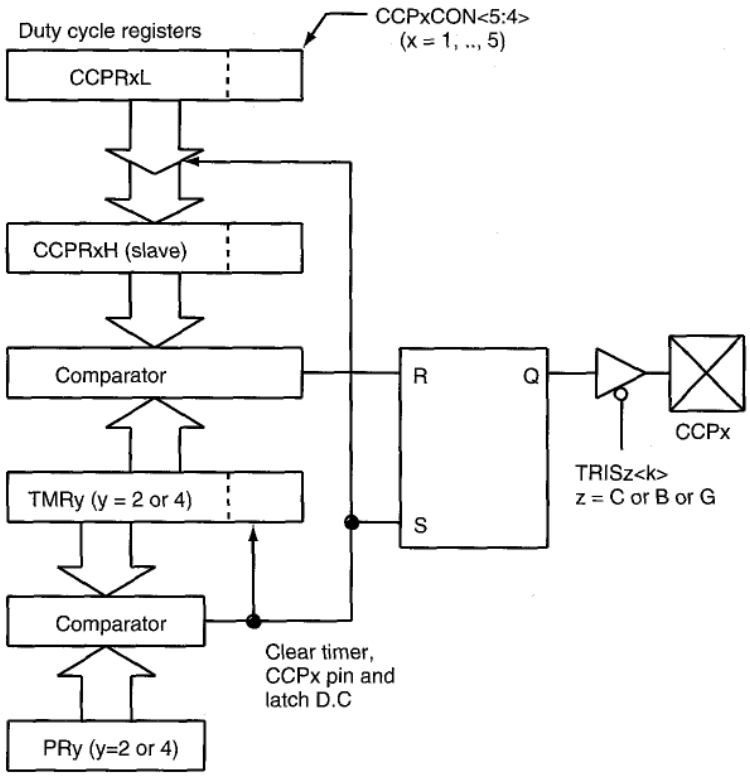

Figure 8.24 ■ Simplified PWM block diagram (redraw with permission of Microchip)

- TMRy register is cleared.
- The PWM duty cycle is latched from CCPRxL into CCPRxH.

The PWM period can be calculated using the following formula:

PWM period = $[(PRy) + 1] \times 4 \times T_{OSC} \times$ (TMRy prescale factor)

The PWM duty cycle can be calculated using the following formula:

PWM duty cycle = (CCPRxL:CCPxCON<5:4>) $\times T_{OSC} \times$ (TMRy prescale factor)

The following steps should be taken when configuring the CCP module for PWM operation:

1. Set the PWM period by writing to the PRy (y = 2 or 4) register.
2. Set the PWM duty cycle by writing to the CCPRxL register and CCPxCON<5:4> bits.
3. Make the CCPx pin an output.
4. Set the TMRy prescale value and enable TMRy by writing to TyCON register.
5. Configure the CCPx module for PWM operation.

The user should be aware that the time unit used in period calculation is *instruction cycle clock*, whereas the time unit used in duty cycle calculation is *crystal oscillator cycle*. Using different time units can improve the accuracy of the waveform to be generated when multiplying the period value by the duty cycle is not an integer. The lowest two bits of the duty cycle can

be used to represent the fractional part (0.0, 0.25, 0.5, or 0.75) of the duty cycle value. For example, if one wants to create a waveform with 62.5% duty cycle from CCP1 pin, then one can place 99 in the PR2 register and 62 in the CCPR1L and set bit 5 and bit 4 of the CCP1CON register to 10. However, since the user can place any value (from 0 to 255) in the CCPRxL register, the lowest two bits may not always represent the fractional part of the duty cycle.

Example 8.12

Configure CCP1 in PWM mode to generate a digital waveform with 40% duty cycle and 10-KHz frequency assuming that the PIC18 microcontroller is running with a 32-MHz crystal oscillator.

Solution: A possible setting is as follows:

1. Use Timer2 as the base timer of CCP1 through CCP5 for PWM mode.
2. Enable Timer3 in 16-bit timer mode with 1:1 prescaler.
3. Set prescaler to Timer2 to 1:4.

The value to be written into PR2 is calculated as follows:

$PR2 = 32 \times 10^6 \div 4 \div 4 \div 10^4 - 1 = 199$

The value to be written into the CCPR1L (and CCPR1H) register is

$CCPR1L = 200 \times 40\% = 80$

The lowest two bits (bits 5 and 4) of the duty cycle value will be cleared to 0. The following instruction sequence will configure CCP1 in PWM mode as desired:

```
movlw   0xC7          ; set period value to 199
movwf   PR2,A         ; "
movlw   0x50          ; set duty cycle value to 80
movwf   CCPR1L,A      ; "
movwf   CCPR1H,A      ; "
bcf     TRISC,CCP1,A  ; configure CCP1 pin for output
movlw   0x81          ; enable Timer3 in 16-bit mode and use
movwf   T3CON,A       ; Timer2 as time base for PWM1 thru PWM5
clrf    TMR2,A        ; force TMR2 to count from 0
movlw   0x05          ; enable Timer2 and set its prescaler to 4
movwf   T2CON,A       ; "
movlw   0x0C          ; enable CCP1 PWM mode
movwf   CCP1CON,A     ; "
```

Function	Description
ClosePWMx	Disable PWM channel x
OpenPWMx	Configure PWM channel x
SetDCPWMx	Write a new duty cycle value to PWM channel x
SetOutputPWMx	Sets the PWM output configuration bits for ECCP

Table 8.2 ■ Microchip C library functions for PWM

8.7.1 PWM C Library Functions

The Microchip PIC18 C compiler provides library functions (listed in Table 8.2) for configuring PWM parameters, disabling PWM, and setting duty cycles. The header file **pwm.h** must be included in order to use these functions.

The prototype declarations and parameter definitions are as follows:

void **ClosePWM1** (void);
void **ClosePWM2** (void);
void **ClosePWM3** (void);
void **ClosePWM4** (void);
void **ClosePWM5** (void);

void **OpenPWM1** (char **period**);
void **OpenPWM2** (char **period**);
void **OpenPWM3** (char **period**);
void **OpenPWM4** (char **period**);
void **OpenPWM5** (char **period**);

Each of the **OpenPWM** functions enables the specified CCP module for PWM mode and configures its period. The user can choose either Timer2 or Timer4 as the base timer of the PWM function. This can be done by configuring the T3CON register to select the desired base timer for PWM operation. The parameter **period** can be any value between 0x00 and 0xFF. This value determines the PWM frequency by using the following formula:

PWM period = [period + 1] × 4 × T_{OSC} × TMRy (y = 2 or 4) prescaler

void **SetDCPWM1** (unsigned int **dutycycle**);
void **SetDCPWM2** (unsigned int **dutycycle**);
void **SetDCPWM3** (unsigned int **dutycycle**);
void **SetDCPWM4** (unsigned int **dutycycle**);
void **SetDCPWM5** (unsigned int **dutycycle**);

These functions write a new duty cycle value to the specified PWM channel duty cycle registers. The value of **dutycycle** can be any 10-bit number:

void **SetOutputPWM1** (unsigned char **outputconfig,** unsigned char **outputmode**);

The value of **outputconfig** can be any one of the following values:

SINGLE_OUT	Single output
FULL_OUT_FWD	Full-bridge output forward
HALF_OUT	Half-bridge output
FULL_OUT_REV	Full-bridge reverse

The value of **outputmode** can be any one of the following values:

PWM_MODE_1	P1A and P1C active high, P1B and P1D active high
PWM_MODE_2	P1A and P1C active high, P1B and P1D active low
PWM_MODE_3	P1A and P1C active low, P1B and P1D active high

PWM_MODE_4	P1A and P1C active low, P1B and P1D active low

This function can be used only with the PIC18 members that come with the enhanced CCP (ECCP) modules. ECCP is discussed in Section 8.8 and is placed in the complimentary CD.

Example 8.13

Write a set of C statements to configure CCP4 to generate a digital waveform with 5-KHz frequency and 70% duty cycle, assuming that the PIC18F8720 is running with a 16-MHz crystal oscillator. Use Timer4 as the base timer.

Solution: To create a waveform with 5-KHz frequency, the user can set the timer prescaler to 8 and set the period value to 100 (99 to be written as the period value). The duty cycle value to be written is $100 \times 70\% \times 4 = 280$.

The following C statements will configure CCP4 to generate the desired waveform:

```
TRISGbits.TRISG3 = 0; /* configure CCP4 pin for output */
OpenTimer3 (TIMER_INT_OFF & T3_16BIT_RW & T3_SOURCE_INT & T3_PS_1_1 & T34_SOURCE_CCP &
            T3_OSC1EN_OFF);
OpenTimer4 (TIMER_INT_OFF & T4_PS_1_8 & T4_POST_1_1);
SetDCPWM4 (280);
OpenPWM4 (99);
```

8.7.2 PWM Applications

There are some applications that respond to the average input voltage rather than the instantaneous input voltage. Lamp dimming and direct-current (DC) motor control are two examples. Both the brightness of a lamp and the speed of a DC motor are proportional to the average applied voltage rather than the instantaneous voltage value.

A PIC18 I/O pin can be used to switch the lightbulb through a PNP or an NPN transistor as long as the lightbulb power consumption is moderate. Two circuit examples are shown in Figure 8.25.

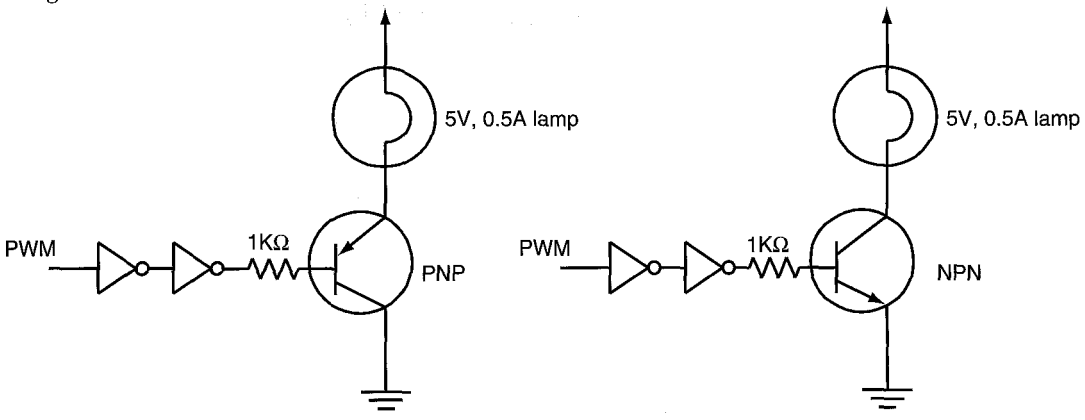

Figure 8.25 ■ Using PWM to control the brightness of a lightbulb

The collector current of the BJT transistor drives the lamp. A PIC18 I/O pin can sink or source up to 25 mA. The collector current is equal to h_{fe} times this current. The parameter of h_{fe} varies from around 15 to around 800. The typical value of h_{fe} is 100. However, one must keep in mind that the larger the collector current gets, the smaller the h_{fe} parameter becomes. The circuit shown in Figure 8.25 is not suitable for applications that require current larger than a few hundred milliamps. If an application requires larger current to drive, then it would be necessary to use some type of power MOSFETs. In Figure 8.25, a buffer is added to protect the microcontroller from being damaged by accidental overload of the transistor.

Example 8.14

Assume that PWM1 is being used to control the brightness of a lamp. The circuit connection is shown in Figure 8.25. Write a program to dim the lamp to 10% brightness in 5 seconds. Assume that the PIC18 microcontroller is running with a 32-MHz crystal oscillator.

Solution: We will dim the lamp as follows:

1. Set duty cycle to 100% at the beginning. Load 99 and 400 as the initial period and duty cycle register values, respectively.
2. Dim the lamp by 10% in the first second by reducing the brightness in 10 steps. This can be achieved by decrementing the duty cycle value by 4 in each step (or decrementing the CCPR1L register by 1).
3. Dim the lamp down to 10% brightness in the next 4 seconds in 40 steps. This is achieved by decrementing the CCPR1L register by 2.

The assembly program that implements this algorithm is as follows:

```
            #include <p18F452.inc>
            org     0x00
            goto    start
            org     0x08
            retfie
            org     0x18
            retfie
start       bcf     TRISC,CCP1      ; configure CCP1 pin for output
            movlw   0x81            ; Use Timer2 as the base timer for PWM1
            movwf   T3CON           ; and enable Timer3 in 16-bit mode
            movlw   0x63            ; set 100 as the period of the digital
            movwf   PR2             ; waveform
            movlw   0x64            ; set 100 as the duty cycle
            movwf   CCPR1L          ; "
            movwf   CCPR1H          ; "
            movlw   0x05            ; enable Timer2 and set its prescaler to 4
            movwf   T2CON           ; "
            movlw   0x0C            ; enable PWM1 operation and set the lowest
            movwf   CCP1CON         ; two bits of duty cycle to 0
            movlw   0x0A            ; use PRODL as the loop count
            movwf   PRODL           ; "
loop_1s     call    delay           ; call "delay" to delay for 100 ms
            decf    CCPR1L,F        ; decrement the duty cycle value by 1
```

```
                decfsz      PRODL,F         ; check to see if loop index expired
                goto        loop_1s
                movlw       0x28            ; repeat the next loop 40 times
                movwf       PRODL           ; "
loop_4s         call        delay           ; call "delay" to delay for 100 ms
                decf        CCPR1L,F        ; decrement duty cycle value by 2
                decf        CCPR1L,F        ; "
                decfsz      PRODL,F         ; is loop index expired?
                goto        loop_4s
forever         nop
                goto        forever
;*********************************************************
; The following function creates 100 ms delay for a 32 MHz crystal oscillator.
;*********************************************************
delay           movlw       0x3C            ; load 15535 into TMR0 so that it will overflow
                movwf       TMR0H           ; in 50000 clock cycles
                movlw       0xAF            ; "
                movwf       TMR0L           ; "
                movlw       0x83            ; set prescaler of TMR0 to 16
                movwf       T0CON,A         ; "
                bcf         INTCON,TMR0IF   ; clear TMR0IF flag
wait_3          btfss       INTCON,TMR0IF   ; wait until the delay time is over
                goto        wait_3
                return
                end
```

The C language version of the program is as follows:

```
#include <p18F452.h>
#include <pwm.h>
#include <timers.h>
void delay (void);
void main (void)
{
                int i;
                TRISCbits.TRISC2 = 0;           /* configure CCP1 pin for output */
                T3CON = 0x81;                   /* use Timer2 as base timer for CCP1 */
                OpenTimer2 (TIMER_INT_OFF & T2_PS_1_4 & T2_POST_1_1);
                SetDCPWM1 (400);                /* set duty cycle to 100% */
                OpenPWM1 (99);                  /* enable PWM1 with period equals 100 */
                for(i = 0; i < 10; i++) {
                                delay();
                                CCPR1L--;       /* decrement duty cycle value by 1 */
                }
                for(i = 0; i < 40; i++) {
                                delay();
                                CCPR1L -= 2;    /* decrement duty cycle value by 2 */
                }
}
void delay (void)
```

```
{
        TMR0 = 0x3CAF; /* load 15535 into TMR0 so that it overflows in 50000 clock cycles */
        OpenTimer0 (TIMER_INT_OFF & T0_16BIT & T0_SOURCE_INT & T0_PS_1_16);
                /* enable Timer0 in 16-bit mode and set prescaler to 16 */
        INTCONbits.TMR0IF = 0; /* clear TMR0IF flag */
        while(!INTCONbits.TMR0IF); /* wait for 100 ms */
}
```

8.7.3 DC Motor Control

A DC motor is devised to convert electrical power into mechanical power. In a DC motor, electrical energy is converted into mechanical energy through the interaction of two magnetic fields. One field is produced by a permanent magnet assembly (on the *stator*), and the other field is produced by an electrical current flowing in the motor winding (on the *rotor*). These two fields result in a torque that tends to rotate the rotor. As the rotor turns, the current in the windings is commutated to produce a continuous torque output. Instead of using permanent magnets to produce permanent magnetic fields, some DC motors use coils.

The rotation of magnetic field is achieved by switching current between coils within the motor. This action is called *commutation*. Most DC motors have built-in commutation in which mechanical brushes automatically commutate coils on the rotor as the motor rotates. Besides these *brush-type* DC motors, there is another DC motor type: *brushless*. A brushless DC motor relies on the external power drive to perform the commutation of stationary copper windings on the stator. This changing stator field makes the permanent magnet rotor rotate. A brushless permanent magnet motor is the highest-performing motor in terms of torque versus weight or efficiency. Brushless motors are usually the most expensive type of motor. Electronically commutated, brushless motors are widely used as drives for blowers and fans in electronics, telecommunications, and industrial equipment applications. There are a wide variety of different brushless motors for various applications. Some are designed to rotate at constant speed (such as those used in disk drives), and the speed of some can be controlled by varying the voltage applied to them.

DC motor speed is controlled by controlling its driving voltage. The higher the voltage, the higher the motor speed. In many applications, a simple voltage regulation would cause lots of power loss in the control circuit, so a PWM method is used in many DC motor-controlling applications. In the basic PWM method, the operating power to the motors is turned on and off to modulate the current to the motor. The ratio of on time to off time is what determines the speed of the motor.

Sometimes the rotation direction needs to be changed. In normal permanent magnet motors, the reversal of motor direction is implemented by changing the polarity of the operating power.

A PWM circuit can be implemented by using discrete components. However, this approach cannot provide the desired flexibility and controllability and is expensive. A better implementation method for PWM circuitry is to use the PWM functions available in many microcontrollers today. Most of the PIC18 devices have PWM functions.

The PIC18 microcontroller can interface with a DC motor through a driver, as shown in Figure 8.26. This circuit takes up only three I/O pins. The pin that controls the direction can be an ordinary I/O pin, but the pin that controls the speed must be a PWM pin. The pin that receives the feedback must be a CCP pin configured in capture mode.

Although some DC motors can operate at 5 V or less, the PIC18 microcontroller cannot supply the necessary current to drive a motor directly. The minimum current required by any practical DC motor is much larger than any microcontroller can supply. Depending on the size and rating of the motor, a suitable driver must be selected to take control signals from the PIC18 microcontroller and deliver the necessary voltage and current to the motor.

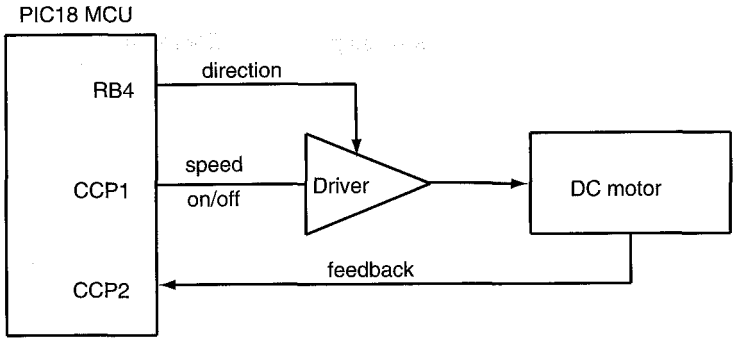

Figure 8.26 ■ A simplified circuit for DC motor control

DC Motor Drivers

Standard motor drivers are available in many current and voltage ratings. Examples are the L292 and L293 made by SGS Thompson Inc. The L293 has four channels and can deliver up to 1 A of current per channel with a supply of 36 V. It has a separate logic supply and takes a logic input (1 or 0) to enable or disable each channel. The L293D also includes clamping diodes needed to drain the *kickback* current generated from the inductive load during the motor reversal. The pin assignment and block diagram of the L293 are shown in Figure 8.27. There are two supply voltages: Vss and Vs. Vss is the logic supply voltage, which can be from 4.5 V to 36 V (normally 5.0 V). Vs is the analog supply voltage and can be as high as 36 V.

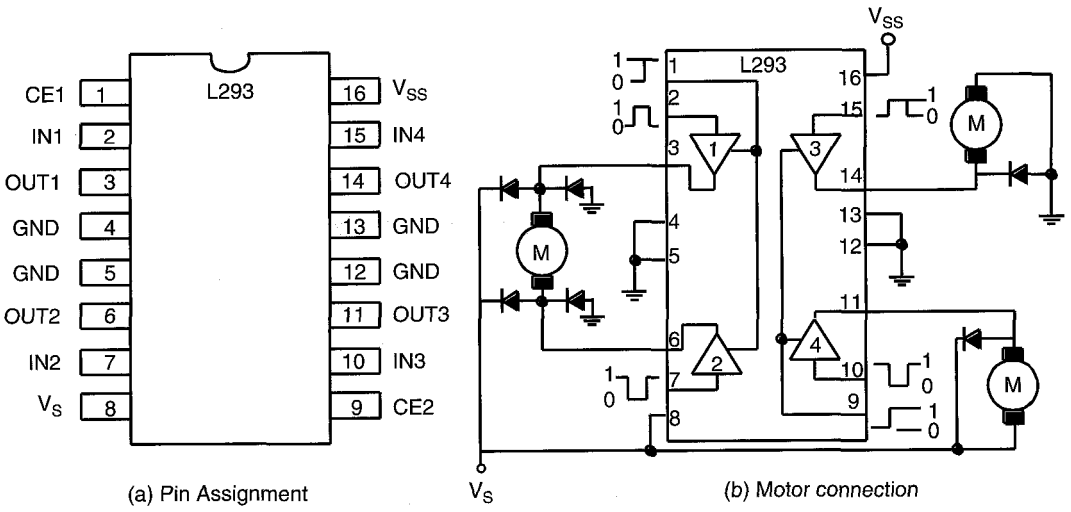

Figure 8.27 ■ Motor driver L293 pin assignment and motor connection

Feedback

The DC motor controller needs information to adjust the voltage output to the motor driver circuit. The most important information is the speed of the motor that must be fed back from the motor by a sensing device. The sensing device may be an optical encoder, infrared detector,

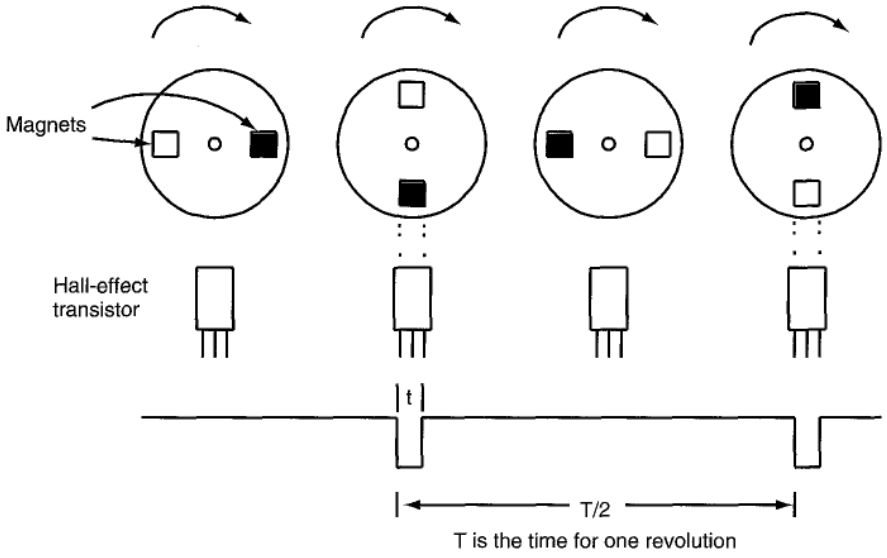

Figure 8.28 ■ The output waveform of the Hall-effect transistor

Hall-effect sensor, and so on. Whatever the means of sensing, the result is a signal that is fed back to the microcontroller. The microcontroller can use the feedback to determine the speed and position of the motor. Then it can make adjustments to increase or decrease the speed, reverse the direction, or stop the motor.

Assume that a Hall-effect transistor is mounted on the shaft (rotor) of a DC motor and that two magnets are mounted on the armature (stator). As shown in Figure 8.28, every time the Hall-effect transistor passes by the magnet, it generates a pulse. One can use the CCP module to capture the passing time of the pulse. The time between two captures is half a revolution. Thus, the motor speed can be calculated. By storing the value of the capture register each time, the controller can constantly measure and adjust the speed of the motor. A motor can therefore be run at a precise speed or synchronized with another event.

The schematic of a motor control system is illustrated in Figure 8.29. The PWM output from the CCP1 pin is connected to one end of the motor, whereas the RB4 pin is connected to the other end. The circuit is connected in such a way that the motor will rotate clockwise when the voltage of the RB4 pin is zero while the PWM output is positive. The direction of the motor rotation is illustrated in Figure 8.30. By applying appropriate voltages on the RB4 and CCP1 pins, the motor can rotate clockwise or counterclockwise or even stop. The CCP2 module in capture mode is used to capture the feedback from the Hall-effect transistor.

When a DC motor is first turned on, it cannot reach a steady speed immediately. A certain amount of startup time should be allowed for the motor to get to speed. A smaller motor usually can reach steady speed faster than a larger one. It is desirable for the motor speed to be a constant for many applications. However, when a load is applied to the motor, it will be slowed down. To keep the speed constant, the duty cycle of the voltage applied to the motor should be increased. When the load gets lighter, the motor will accelerate and run faster than desired. To slow down the motor, the duty cycle of the applied voltage should be reduced.

The response time will be slow if the change to the duty cycle is small. However, a large variation in the duty cycle tends to cause it to overreact and causes oscillation. There are control algorithms that you can find discussed in textbooks on control.

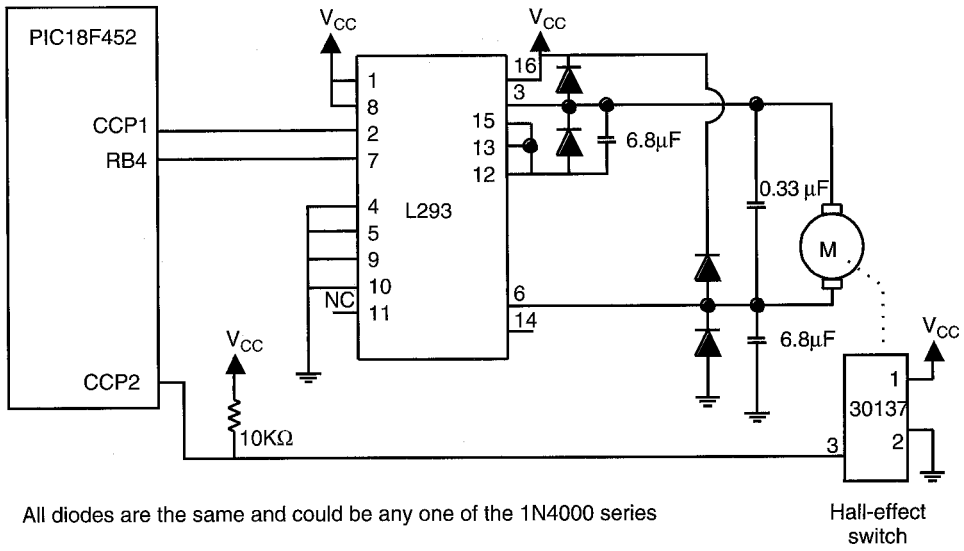

Figure 8.29 ■ Schematic of a PIC18-based motor-control system

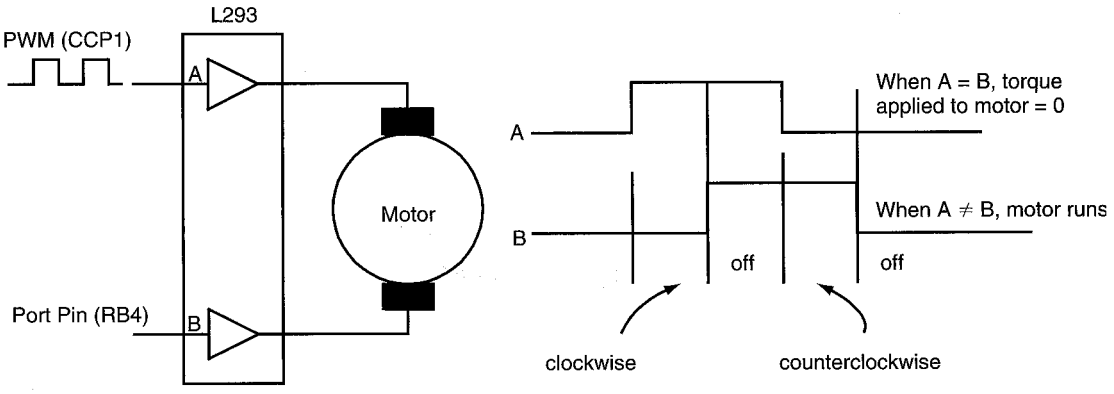

Figure 8.30 ■ The L293 motor driver

A DC motor cannot respond to the change of duty cycle instantaneously because of its inertia. A certain amount of time should be allowed for the motor to speed up or slow down before the effect of the change of duty cycle is measured.

Electrical Braking

Once a DC motor is running, it picks up speed. Turning off the voltage to the motor does not make it stop immediately because the momentum of the motor will keep it rotating. After the voltage is turned off, the momentum will gradually wear out because of friction. If the application does not require an abrupt stop, then the motor can be brought to a gradual stop by removing the driving voltage.

An abrupt stop may be required by certain applications in which the motor must run a few turns and stop quickly at a predetermined point. This can be achieved by electrical braking. Electrical braking is done by reversing the voltage applied to the motor. The length of time that the reversing voltage is applied must be precisely calculated to ensure a quick stop while not starting it in the reverse direction. A discussion of good motor braking algorithms is outside the scope of this textbook.

In a closed-loop system, the feedback can be used to determine where and when to start and stop braking and when to discontinue. In Figure 8.30, the motor can be braked by (1) reducing the PWM duty cycle to 0 and (2) setting the RB4 pin's output to high for an appropriate amount of time.

Example 8.15

For the circuit shown in Figure 8.29, write a function in C language to measure the motor speed (in RPM), assuming that the PIC18 microcontroller is running with a 20-MHz crystal oscillator.

Solution: The motor speed can be measured by capturing two consecutive rising or falling edges. Let the difference of two captured edges and the clock frequency of Timer1 be **diff** and **f**, respectively. Then the motor speed (in RPM) is

$$\text{Speed} = 60 \times f \div (2 \times \text{diff})$$

The C function that measures the motor speed is as follows:

```
unsigned int motor_speed(void)
{
  unsigned int edge1, diff, rpm;
  long unsigned temp;
  T3CON = 0x81;              /* enables Timer3 in 16-bit mode and use Timer1 and Timer2 for CCP1 thru CCP2
                                operations */
  OpenTimer1(TIMER_INT_OFF & T1_16BIT_RW & T1_SOURCE_INT & T1_PS_1_4);
          /* set Timer1 prescaler to 1:4 */
  PIR2bits.CCP2IF = 0;
  OpenCapture2(CAPTURE_INT_OFF & C2_EVERY_RISE_EDGE);
  while (!PIR2bits.CCP2IF);
  edge1 = CCPR2;             /* save the first rising edge */
  PIR2bits.CCP2IF = 0;
  while (!PIR2bits.CCP2IF);
  CloseCapture2();
  diff = CCPR2 - edge1;      /* compute the difference of two rising consecutive edges */
  temp = 1250000ul/(2 * diff);
  rpm = temp * 60;
  return rpm;
}
```

Example 8.16

Write a subroutine to perform the electrical braking.

Solution: Electrical braking is implemented by setting the duty cycle to 0% and setting the voltage on the RB4 pin to high for certain amount of time. The following subroutine will perform the desired electrical braking:

```
brake   bsf     PORTB,RB4,A     ; reverse the applied voltage to motor
        movlw   0x00            ; "
        movwf   CCPR1L,A        ; set PWM1 duty cycle to 0
        call    brake_time      ; wait for certain amount of time
        bcf     PORTB,RB4,A     ; stop braking
        return
```

The subroutine **brake_time** will wait for certain amount of time to allow for the motor to stop. It is similar to the delay routine that we learned earlier.

Instead of calling the **brake_time** function, the brake function can also keep monitoring the motor speed until it drops to a very low value (say, 5%) and then stop the PWM function. This method is also straightforward and hence is left for you as an exercise problem.

The motor braking can be invoked by using external interrupt. Using this approach, the brake function would be implemented as an interrupt service routine.

8.8 Enhanced CCP Module

The PIC18F448/458, PIC18F1220/1320, and PIC18F4220/4320 have incorporated one enhanced CCP (ECCP) module. The PIC18F8621/8525 and PIC6621/6525 incorporate three ECCP modules. Many future PIC18 members will also have the enhanced ECCP module. The ECCP module is implemented as a standard CCP module with enhanced PWM capabilities with the intention to simplify the support of motor-control applications.

8.8.1 ECCP Pins

Depending on the operating mode, one ECCP may have up to four outputs. These outputs, designated P1A through P1D, are multiplexed with I/O pins on PORTB (18F1x20) or PORTC and PORTD (18F458) or PORTD (18F4x20). The pin assignments are summarized in Table 8.3. These four pins must be configured for output. Depending on the device, either RC2 or RD4 is used as the CCP1/P1A pin. The pin assignments for P1A–P1D, P2A–P2D, and P3A–P3D for other PIC18 devices can be found in appropriate datasheets.

ECCP mode	CCP1CON Configuration	RB3 or RC2 or RD4	RB2 or RD5	RB6 or RD6	RB7 or RD7
Compatible CCP	00xx11xx	CCP1	I/O pin	I/O pin	I/O pin
Dual PWM	10xx11xx	P1A	P1B	I/O pin	I/O pin
Quad PWM	x1xx11xx	P1A	P1B	P1C	P1D

Table 8.3 ■ Pin assignments for various ECCP modes

8.8.2 ECCP Registers[1]

The ECCP module differs from the CCP module with the addition of an enhanced PWM mode, which allows for two or four output channels, user-selectable polarity, deadband control, and automatic shutdown and restart. The control register for the CCP1 module is shown in Figure 8.31. The control registers for the ECCP2 and ECCP3 modules are identical and are called CCP2CON and CCP3CON, respectively.

[1]Section 8.8 can be skipped without affecting one's understanding of later chapters.