Table 2.1 A basic set of CCS C components

Compiler Directives

`#include` *`source files`*	Include another source code or header file
`#use` *`functions(parameters)`*	Include library functions

C Blocks

`main(`*`condition`*`) {statements }`	Main program block
`while(`*`condition`*`) {statements }`	Conditional loop
`if(`*`condition`*`) {statements }`	Conditional sequence
`for(`*`condition`*`) {statements }`	Preset loop

C Functions

`delay_ms(nnn)`	Delay in milliseconds
`delay_us(nnn)`	Delay in microseconds
`output_x(n)`	Output 8-bit code at Port X
`output_high(PIN_nn)`	Set output bit high
`output_low(PIN_nn)`	Set output bit low
`input(PIN_nn)`	Get input

2.3 PIC16 C Data Operations

- Variable types
- Floating point numbers
- Characters
- Assignment operators

A main function of any computer program is to carry out calculations and other forms of data processing. Data structures are made up of different types of numerical and character variables, and a range of arithmetical and logical operations are needed.

Microcontroller programs do not generally need to process large volumes of data, but processing speed is often important.

Table 2.1 Integer Variables

Name	Type	Min	Max
int1	1 bit	0	1
unsigned int8	8 bits	0	255
signed int8	8 bits	-127	+127
unsigned int16	16 bits	0	65525
signed int16	16 bits	-32767	+32767
unsigned int32	32 bits	0	4294967295
signed int32	32 bits	-2147483647	+2147483647

Floating Point Number Format

Table 2.2 Microchip/CCS Floating Point Number Format

Exponent	Sign	Mantissa
xxxx xxxx	x	xxx xxxx xxxx xxxx xxxx xxxx
8 bits	1	23 bits

Table 2.4 Example of 32-bit floating point number conversion

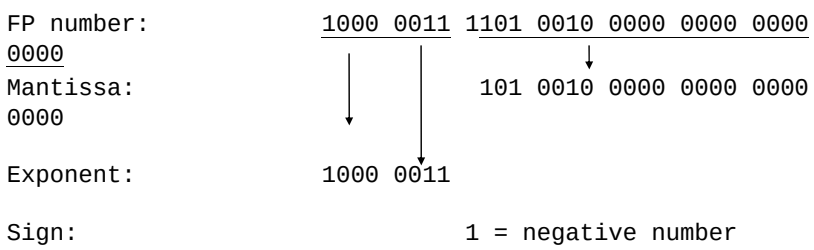


Figure 2.5 Variable Types

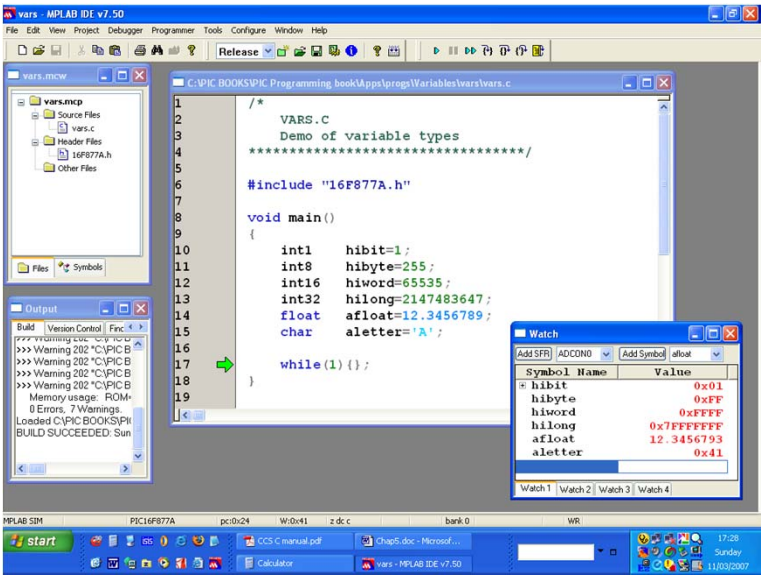


Table 2.5 ASCII Codes

Low Bits	High Bits					
	0010	0011	0100	0101	0110	0111
0000	Space	0	@	P	`	p
0001	!	1	A	Q	a	q
0010	"	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	'	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K	[	k	{
1100	,	<	L	\	l	|
1101	-	=	M	]	m	}
1110	.	>	N	^	n	~
1111	/	?	O	_	o	Del

Table 2.6 Arithmetic and Logical Operations

OPERATION	OPERATOR	DESCRIPTION	SOURCE CODE	EXAMPLE	RESULT
Single operand					
Increment	++	*Add one to integer*	`result = num1++;`	0000 0000	0000 0001
Decrement	--	*Subtract one from integer*	`result = num1--;`	1111 1111	1111 1110
Complement	~	*Invert all bits of integer*	`result = ~num1;`	0000 0000	1111 1111
Arithmetic Operation					
Add	+	*Integer or Float*	`result = num1 + num2;`	0000 1010 + 0000 0011	0000 1101
Subtract	-	*Integer or Float*	`result = num1 - num2;`	0000 1010 - 0000 0011	0000 0111
Multiply	*	*Integer or Float*	`result = num1 * num2;`	0000 1010 * 0000 0011	0001 1110
Divide	/	*Integer or Float*	`result = num1 / num2;`	0000 1100 / 0000 0011	0000 0100
Logical Operation					
Logical AND	&	*Integer Bitwise*	`result = num1 & num2;`	1001 0011 & 0111 0001	0001 0001
Logical OR	|	*Integer Bitwise*	`result = num1 | num2;`	1001 0011 | 0111 0001	1111 0011
Exclusive OR	^	*Integer Bitwise*	`result = num1 ^ num2;`	1001 0011 ^ 0111 0001	1110 0010

Figure 2.6 Variable Operations

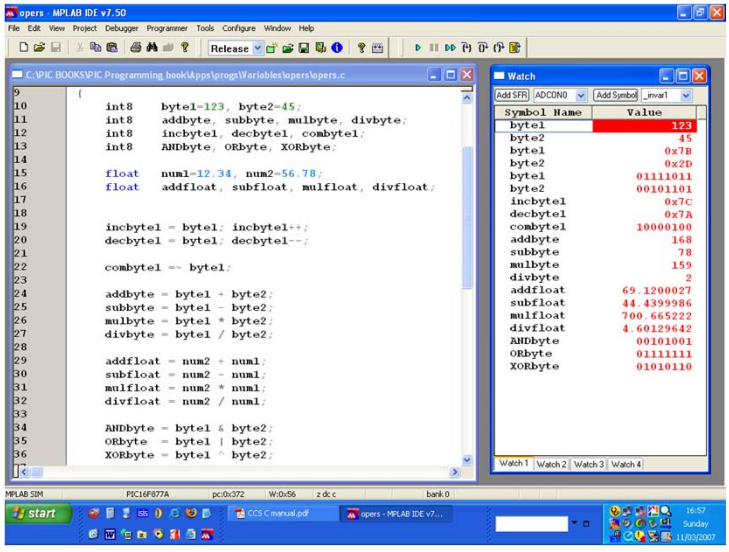


Table 2.7: Conditional Operators

Operation	Symbol	EXAMPLE
Equal to	==	`if(a == 0) b=b+5;`
Not equal to	!=	`if(a != 1) b=b+4;`
Greater than	>	`if(a > 2)  b=b+3;`
Less than	<	`if(a < 3)  b=b+2;`
Greater than or equal to	>=	`if(a >= 4) b=b+1;`
Less than or equal to	<=	`if(a <= 5) b=b+0;`

2.4 PIC16 C Sequence Control

- While loops
- Break, continue, goto
- If, else, switch

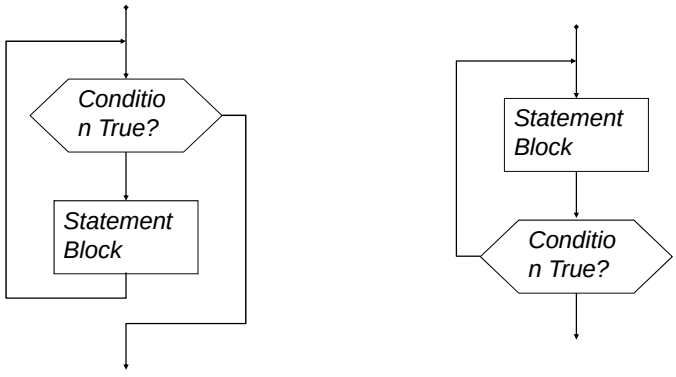
- Conditional branching operations are a basic feature of any program.
- These must be properly organized so that the program structure is maintained and confusion avoided.
- The program then is easy to understand and more readily modified and upgraded.

While Loops

The basic *while(condition)* provides a logical test at the start of a loop, and the statement block is executed only if the condition is true. It may, however, be desirable that the loop block be executed at least once, particularly if the test condition is affected within the loop. This option is provided by the *do..while(condition)* syntax. The difference between these alternatives is illustrated in Figure 2.7 . The WHILE test occurs before the block and the DO WHILE after.

The program DOWHILE shown in Listing 2.9 includes the same block of statements contained within both types of loop. The WHILE block is not executed because the loop control variable has been set to 0 and is never modified. By contrast, ‘ count ’ is incremented within the DO WHILE loop before being tested, and the loop therefore is executed.

Figure 2.3.1 Comparison of While and Do..While Loop



(a) While loop

(b) Do..While loop

Listing 2.9 DOWHILE.C contains both types of 'while' loop

```
//  DOWHILE.C
//  Comparison of WHILE and DO WHILE loops
#include "16F877A.H"
main()
{
int outbyte1=0;
           int outbyte2=0;
           int count;
           count=0;                        // This loop is not
    while (count!=0)                       // executed
    {      output_C(outbyte1);
           outbyte1++;
           count--;
           }
           count=0;                        // This loop is
           do                              // executed
           {      output_C(outbyte2);
           outbyte2++;
           count--;
           } while (count!=0);
           while(1){};
}
```

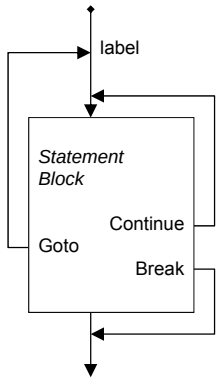

Break, Continue, and Goto

It may sometimes be necessary to *break* the execution of a loop or block in the middle of its sequence (Figure 2.8). The block must be exited in an orderly way, and it is useful to have the option of restarting the block (*continue*) or proceeding to the next one (*break*).

Occasionally, an unconditional jump may be needed, but this should be regarded as a last resort, as it tends to threaten the program stability. It is achieved by assigning a label to the jump destination and executing a *goto*..label.

The use of these control statements is illustrated in Listing 2.10 . The events that trigger break and continue are asynchronous (independent of the program timing) inputs from external switches, which allows the counting loop to be quit or restarted at any time.

Figure 2.8 Break, continue and goto



Listing 2.10 Continue, Break & Goto

```
//     CONTINUE.C
//     Continue, break and goto jumps
#include "16F877A.H"
#use delay(clock=4000000)
main()
{
    int outbyte;
    again: outbyte=0;                    // Goto destination
    while(1)
    {
        output_C(outbyte);        // Loop operation
        delay_ms(10);
        outbyte++;
        if (!input(PIN_D0)) continue;    // Restart loop
        if (!input(PIN_D1)) break;       // Terminate loop
        delay_ms(100);
        if (outbyte==100) goto again;    // Unconditional jump
    }
}
```

Figure 2.9 Comparison of If and If..Else

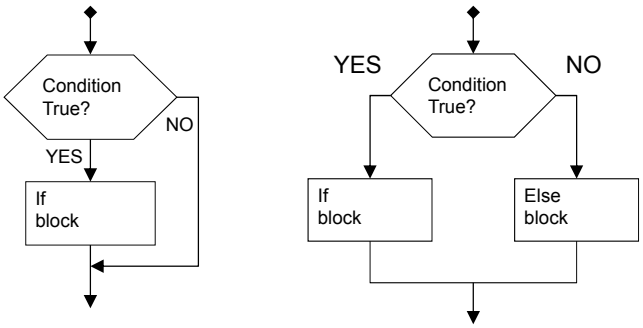
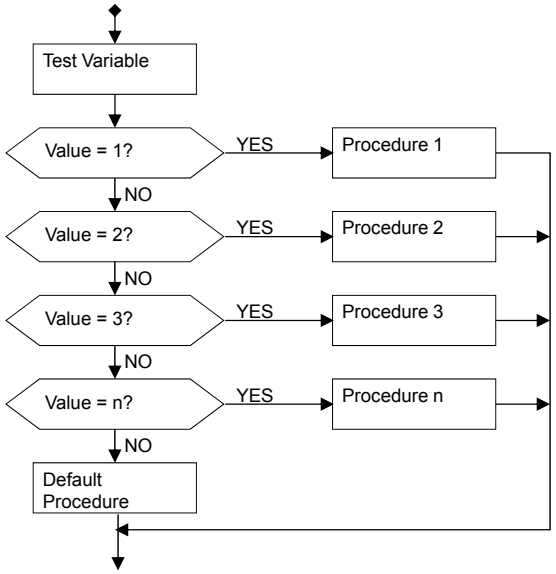


Figure 2.10 Switch..case branching structure



Listing 2.11 Comparison of Switch and If..Else control

```
//      SWITCH.C
//      Switch and if..else sequence control
//      Same result from both sequences
#include "16F877A.h"
void main()
{
    int8 inbits;
    while(1)
    {
        inbits = input_D();               // Read input byte
// Switch..case option.....................................................
        switch(inbits)                    // Test input byte
        {
            case 1: output_C(1);          // Input = 0x01, output = 0x01
                    break;                // Quit block
            case 2: output_C(3);          // Input = 0x02, output = 0x03
                    break;                // Quit block
            case 3: output_C(7);          // Input = 0x03, output = 0x07
                    break;                // Quit block
            default:output_C(0);          // If none of these, output = 0x00
        }

// If..else option.....................................................
        if (input(PIN_D0))  output_C(1);           // Input RD0 high
        if (input(PIN_D1))  output_C(2);           // Input RD1 high
        if (input(PIN_D0) && input(PIN_D1)) output_C(7);    // Both high
        else output_C(0);                 // If none of these, output = 0x00
    }
}
```


Class Assignments

1. Write a C function to convert a BCD code to a common-anode 7-segment LED code
2. Write a C program to read 8-bit value from Port B, then add 5 to them and output the result to Port D.
3. Write a C statement to convert numbers 0 to 9 to their ASCII hex code.
4. Write a function to detect a button press and button release with de-bouncing ability
5. Write a C program to create a chasing LED effect with 8 single LEDs at port D.

Basic Functions

- A simple program using a function is shown in FUNC1.C, Listing 2.12 . The main block is very short, consisting of the function call out() and a while statement, which provides the wait state at the end of main().
- In this case, the variables are declared *before the main block. This makes them global in scope; that is, they are recognized* throughout the whole program and within all function blocks. The function out() is also defined *before main() , so that, when it is called, the function name is recognized. The* function starts with the keyword void , which indicates that no value is returned by the function. The significance of this is explained shortly.
- The function itself simply increments Port C from 0 to 255. It contains a for loop to provide a delay, so that the output count is visible. This is a simple alternative to the built-in delay functions seen in previous examples and is used here to avoid the inclusion of such functions while we study user-defined functions. It simply counts up to a preset value to waste time. The delay time is controlled by this set value.

Listing 2.12 Basic function call

```
//  FUNC1.C
//  Function call structure
#include "16F877A.H"
int8    outbyte=1;
int16   n;
 void out()                    // Start of function block
{
   while (outbyte!=0)  // Start loop, quit when output =0
   {
     output_C(outbyte);// Output code 1 – 0xFF
     outbyte++;         // Increment output
     for(n=1;n<500;n++);      // Delay so output is visible
   }
}
main()
{
    out();              // Function call
    while(1);           // Wait until reset
}
```

Global and Local Variables

- Now, assume that we wish to pass a value to the function for local. The simplest way is to define it as a global variable, which makes it available throughout the program. In program FUNC2.C, Listing 2.13 , the variable count, holding the delay count, hence the delay time, is global.
- If there is no significant restriction on program memory, global variables may be used. However, microcontrollers, by definition, have limited memory, so it is desirable to use local variables whenever possible within the user functions. This is because local variables exist only during function execution, and the locations used for them are freed up on completion of function call. This can be confirmed by watching the values of C program variables when the program is executed in simulation mode — the local ones become undefined once the relevant function block is terminated.
- If only global variables are used and the functions do not return results to the calling block, they become procedures. Program FUNC3.C, Listing 2.14 , shows how local variables are used.

Listing 2.13 Passing a parameter to the function

```
//  FUNC2.C
#include "16F877A.H"
int8    outbyte=1;         // Declare global variables
int16   n,count;

    void out()                    // Function block
    {
        while (outbyte!=0)
           {  output_C(outbyte);
              outbyte++;
              for(n=1;n<count;n++);
           }
    }
main()
{
    count=2000;
    out();                  // Call function
    while(1);
}
```

Listing 2.14 Local variables

```
//  FUNC3.C
//  Use of local variables

#include "16F877A.H"

int8    outbyte=1;            // Declare global variables
int16   count;

int out(int16 t)              // Declare argument types
{
    int16 n;                  // Declare local variable

    while (input(PIN_D0))     // Run output at speed t
    {     outbyte++;
          for(n=1;n<t;n++);
    }
    return outbyte;           // Return output when loop stops
}

main()
{
    count=50000;
    out(count);               // Pass count value to function
    output_C(outbyte);        // Display returned value
    while(1);
}
```


2.6 PIC16 C More Data Types

- Arrays and strings
- Pointers and indirect addressing
- Enumeration

The data in a C program may be most conveniently handled as sets of associated variables. These occur more frequently as the program data becomes more complex, but only the basics are mentioned here.

3. Timer and Interrupt

- PIC16F877 has 14 interrupt sources

No	Interrupt Label	Interrupt Source
1	INT_EXT	External interrupt detect on RB0
2	INT_RB	Change on Port B detect
3	INT_TIMER0 (INT_RTCC)	Timer 0 overflow
4	INT_TIMER1	Timer 1 overflow
5	INT_TIMER2	Timer 2 overflow
6	INT_CCP1	Timer 1 capture or compare detect
7	INT_CCP2	Timer 2 capture or compare detect

3. Timer and Interrupt

- PIC16F877 has 14 interrupt sources

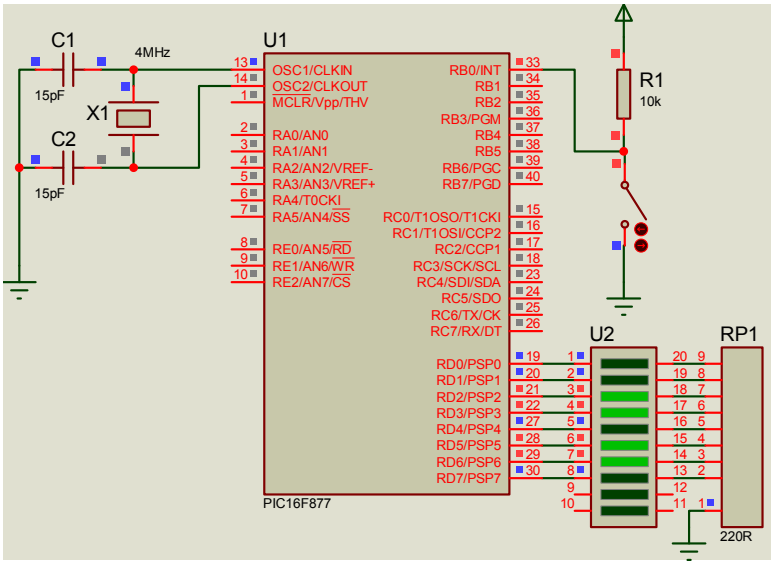
No	Interrupt Label	Interrupt Source
8	INT_TBE	USART transmit data done
9	INT_RDA	USART receive data ready
10	INT_SSP	Serial data received at SPI or I2C
11	INT_BUSCOL	I2C collision detected
12	INT_PSP	Data ready at parallel serial port
13	INT_AD	Analog-to-digital converter complete
14	INT_EEPROM	EEPROM write completion

C Interrupts

- CCS C Interrupt Functions

Action	Description	Example
INTERRUPT CLEAR	Clears peripheral interrupt	clear_interrupt(int_timer0);
INTERRUPT DISABLE	Disables peripheral interrupt	disable_interrupts(int_timer0);
INTERRUPT ENABLE	Enables peripheral interrupt	enable_interrupts(int_timer0);
INTERRUPT ACTIVE	Checks if interrupt flag set	interrupt_active(int_timer0);
INTERRUPT EDGE	Selects interrupt trigger edge	ext_int_edge(H_TO_L);
INTERRUPT JUMP	Jump to address of ISR	jump_to_isr(isr_loc);

Interrupt example



Interrupt example

```
#include " 16F877A.h "
#use delay(clock = 4000000)
#int_ext // Interrupt name
void isrext() {                          // Interrupt service routine
        output_D(255);                   // ISR action
        delay_ms(1000);
}
void main()  {
        int x;
        enable_interrupts(int_ext);      // Enable named interrupt
        enable_interrupts(global);       // Enable all interrupts
        ext_int_edge(H_TO_L);            // Interrupt signal polarity
        while(1) {                       // Foreground loop
                output_D(x); x + + ;
                delay_ms(100);
        }
}
```


Interrupt statements

- #int_xxx
  - Tells the compiler that the code immediately following is the service routine for this particular interrupt
  - The interrupt name is preceded by #(hash) to mark the start of the ISR definition and to differentiate it from a standard function block.
  - An interrupt name is defined for each interrupt source.
- enable_interrupts(int_ext);
  - Enables the named interrupt by loading the necessary codes into the interrupt control registers

Interrupt statements

- enable_interrupts(level);
  - Enables the interrupt at the given level.
  - Examples:

```
enable_interrupts(GLOBAL);
enable_interrupts(INT_TIMER0);
enable_interrupts(INT_TIMER1);
```

- Disable_interrupts(level)
  - Disable interrupt at the given level
- ext_int_edge(H_TO_L);
  - Enables the edge on which the edge interrupt should trigger. This can be either rising or falling edge.

Class Assignment

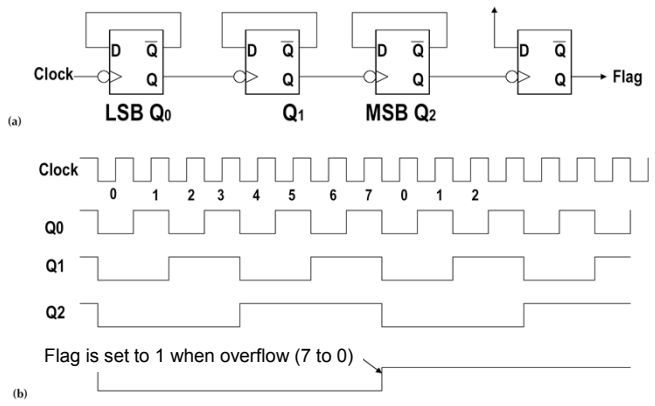
1. Write C code to enable an external interrupt at RB0 with the trigger low to high
2. Write a C program to control 4 output pins RC0-RC3 from 4 input pins RB4-RB7 using port interrupt.

3. PIC16 Hardware Timers

- The PIC 16F877 has three hardware timers built in:
  - Timer0: 8-bit, originally called RTCC, the real-time counter clock
  - Timer1: 16-bit
  - Timer2: 8-bit
- The principal modes of operation
  - **Counters** for external events
  - **Timers** using the internal clock.

Counter/Timer Operation

- A counter/timer register consists of a set of bistable stages (flip-flops) connected in cascade (8, 16, or 32 bits).



- An 8-bit counter counts up from 0x00 to 0xFF

Counter/Timer Operation

- Timer0 is an 8-bit register that can count pulses at RA4; for this purpose, the input is called T0CKI (Timer0 clock input).
- Timer1 is a 16-bit register that can count up to 0xFFFF (65,535) connected to RC0 (T1CKI) .
- The count can be recorded at any chosen point in time; alternatively, an interrupt can be generated on overflow to notify the processor that the maximum count has been exceeded.
- If the register is preloaded with a suitable value, the interrupt occurs after a known count.
- Timer0 has a prescaler that divides by up to 128;
- Timer1 has one that divides by 2, 4, or 8;
- Timer2 has a prescaler and postscaler that divide by up to 16.

Timer Functions

Functions	Description	Examples
Setup_timer_x	Setup timer	setup_timer_0(RTCC_INTERNAL | RTCC_DIV_8);
Set_timerx(value)	Set the value of the timer	Set_timer0(81);
Get_timerx()	Get the value of the timer	int x = get_timer0();
Setup_ccpx(mode)	Set PWM, capture, or compare mode	setup_ccp1(ccp_pwm);
Set_pwmx_duty(value)	Set PWM duty cycle	set_pwm1_duty(512);

Timer Functions

- **Setup_timer_0(mode)**
  - RTCC_INTERNAL, RTCC_EXT_L_TO_H or RTCC_EXT_H_TO_L
  - RTCC_DIV_2, RTCC_DIV_4, RTCC_DIV_8, RTCC_DIV_16, RTCC_DIV_32, RTCC_DIV_64, RTCC_DIV_128, RTCC_DIV_256
- **Setup_timer_1(mode)**
  - T1_DISABLED, T1_INTERNAL, T1_EXTERNAL, T1_EXTERNAL_SYNC
  - T1_CLK_OUT
  - T1_DIV_BY_1, T1_DIV_BY_2, T1_DIV_BY_4, T1_DIV_BY_8
- **Example:**
  - setup_timer_0(RTCC_INTERNAL | RTCC_DIV_8)
  - setup_timer_1 (T1_DISABLED); //disables timer1
  - setup_timer_1 (T1_INTERNAL | T1_DIV_BY_4);
  - setup_timer_1 (T1_INTERNAL | T1_DIV_BY_8);

Timer Functions

- **setup_timer_2 (*mode, period, postscale*)**
  - ***mode*** may be one of T2_DISABLED, T2_DIV_BY_1, T2_DIV_BY_4, T2_DIV_BY_16
  - ***period*** is a int 0-255 that determines when the clock value is reset,
  - ***postscale*** is a number 1-16 that determines how many timer overflows before an interrupt: (1 means once, 2 means twice, and so on).

Timer Functions

- Create delay by timers

  $N = 2^n – (T * F_{clock}) / (4*Prescaler)$

  - N: the count number
  - n: bit number of timer (Timer 0 & 2: n=8, Timer1: n = 16)
  - T: delay time
  - F_{clock}: frequency of crystal
  - Prescaler: prescaler number

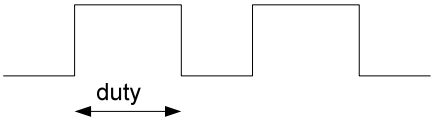

```
#include <16F877A.h>
#device adc=8
#use delay(clock=20000000)
#byte portc = 7
int8 bin2bcd (int8 bin);
void main(){
   int8 count;
   set_tris_a(0x10);
   set_tris_c(0x00);
   setup_timer_0(RTCC_EXT_H_TO_L|RTCC_DIV_1|RTCC_8_BIT);//
   //setup_timer_1(T1_EXTERNAL_SYNC|T1_DIV_BY_1);
   set_timer0(0x00);

   while(1) {
      count = get_timer0();
      if(count == 20) {
         count = 0;
         set_timer0(0x00);
      }
      portc = bin2bcd(count);
   }
}
int bin2bcd(int8 bin){
   int8 bcd;
   bcd = ((bin/10)<<4) + (bin % 10);
   return bcd;
   }
```

The program that carries out the function of a counting circuit counts from 00 to 19 and displays on two 7-seg leds connected to port C

PWM mode

- In Pulse Width Modulation mode, a CCP module can be used to generate a timed output signal.
- This provides an output pulse waveform with an adjustable high (mark) period.
- CCS C functions:
  - Set_pwm1_duty(value);
  - Set_pwm2_duty(value);
  - duty cycle = value / [4 * (PR2 +1)]
  - PR2 is the count value of timer 2



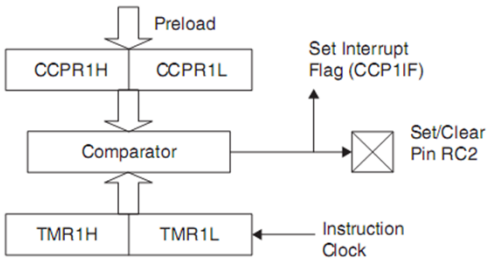
PWM mode - Example

```
#include " 16F877A.h "
void main()
{
   setup_ccp1(ccp_pwm);      // Select timer and mode
   set_pwm1_duty(500);       // Set on time
   setup_timer_2(T2_DIV_BY_16,249,1); // Clock rate & output
                             //period
   while(1) { }              // Wait until reset
}
```

Produce an output at CCP1 of 250Hz (4ms) and a mark-space ratio of 50% with a 4-MHz MCU clock. Explain?

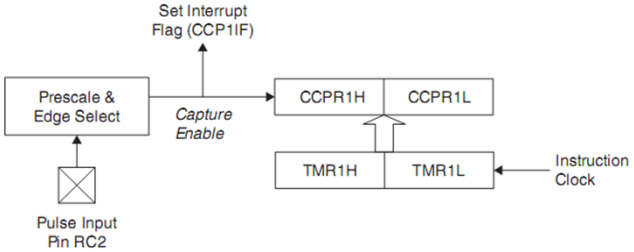
Compare Mode

- Generate a timed output in conjunction with Timer1.
- The 16-bit CCPR register is preloaded with a set value, which is continuously compared with the Timer1 count. When the count matches the CCPR value, the output pin toggles and a CCP interrupt is generated. If this operation is repeated, an interrupt and output change with a known period can be obtained.



Capture mode

- The CCP pin is set to input and monitored for a change of state.
- When a rising or falling edge (selectable) is detected, the timer register is cleared to 0 and starts counting at the internal clock rate.
- When the next active edge is detected at the input, the timer register value is copied to the CCP register. The count therefore corresponds to the period of the input signal. With a 1MHz instruction clock, the count is in microseconds



Exercise – Timer Interrupt

1) Calculate the frequency of the pulse on PIN B0 created by the program on the right figure, given that F_{OSC} = 4MHz

2) Write the program that create a 2Hz pulse on PIN_B1, given that F_{OSC} = 4MHz and dutycycle = 20%

```
// Timer interrupt
#include "16F877A.h"
#use delay(clock=4000000)

#int_timer1                    // Interrupt name
    void isrtimer1()           // Interrupt service routine
    {
      output_toggle(PIN_B0);
      set_timer1(-50000);
    }
void main() {
    int x;

    enable_interrupts(int_timer1);      // Enable named interrupt
    enable_interrupts(global);          // Enable all interrupts
    // ext_int_edge(H_TO_L);      // Interrupt signal polarity
    setup_timer_1(T1_INTERNAL | T1_DIV_BY_8);
    set_timer1(-50000);
    while(1);

}
```


Class Assignment

1. Write a program for PIC16F877 to create rectangle pulses 2KHz at RB1 using interrupt Timer 0.
2. Write a C program for PIC16F877 to create rectangle pulses 0.5KHz and 1KHz at RC0 and RC1 with duty cycle 50%. Use Timer1 interrupt with 4MHz OSC.
3. Write the program that create a 2Hz pulse on PIN_B1, given that F_{OSC} = 4MHz and dutycycle = 20%