Experiment #9 (HW#4)
Capture, Compare and Pulse Width Modulation

1. Prerequisites

ENCS 538, C programming language, PICC compiler, Microchip PIC16F877A datasheet.

2. Objectives

- Getting familiar with Capture, Compare, and Pulse Width Modulation (PWM) module.
- Configuring the CCP module to capture an input signal.
- Generating signals of different frequencies and duty cycles.

3. Background

The abbreviation CCP stands for Capture/Compare/PWM. The CCP module is a peripheral which allows the user to timing and controlling different events.

Capture Mode: allows timing for the duration of an event. This circuit gives insight into the current state of a register which constantly changes its value. In this case, it is the timer TMR1 register.

Compare Mode: compares values contained in two registers at some point. One of them is the timer TMR1 register. This circuit also allows the user to trigger an external event when a predetermined amount of time has expired.

PWM: Pulse Width Modulation can generate signals of varying frequency and duty cycle.

The PIC16F887 microcontroller has two such modules: CCP1 and CCP2.

Both of them are identical in normal mode, with the exception of the Enhanced PWM features available on CCP1 only. Each CCP module has 3 registers. Multiple CCP modules may exist on a single device. Throughout this section we use generic names for the CCP registers. These generic names are shown in Table 1.

Table 1: the CCP1 and CCP2 modules

Generic Name	CCP1	CCP2	Comment
CCPxCON	CCP1CON	CCP2CON	CCP control register
CCPRxH	CCPR1H	CCPR2H	CCP High byte
CCPRxL	CCPR1L	CCPR2L	CCP Low byte
CCPx	CCP1	CCP2	CCP pin

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0
bit 7							bit 0

bit 7:6 **Unimplemented:** Read as '0'

bit 5:4 **DCxB1:DCxB0**: PWM Duty Cycle bit1 and bit0

Capture Mode:

Unused

Compare Mode:

Unused

PWM Mode:

These bits are the two LSbs (bit1 and bit0) of the 10-bit PWM duty cycle. The upper eight bits (DCx9:DCx2) of the duty cycle are found in CCPRxL.

bit 3:0 **CCPxM3:CCPxM0**: CCPx Mode Select bits

0000 = Capture/Compare/PWM off (resets CCPx module)
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 (CCPIF bit is set)
1001 = Compare mode,
Initialize CCP pin High, on compare match force CCP pin Low (CCPIF bit is set)
1010 = Compare mode,
Generate software interrupt on compare match
(CCPIF bit is set, CCP pin is unaffected)
1011 = Compare mode,
Trigger special event (CCPIF bit is set)
11xx = PWM mode

Legend
R = Readable bit **W** = Writable bit, **U** = Unimplemented bit, read as '0' **n** = Value at POR reset

Figure 1: CCPxCON Register

3.1 The capture mode

In Capture mode, CCPRxH:CCPRxL captures the 16-bit value of the TMR1 register when an event occurs on pin CCPx. An event is triggered for an input signal can be on every falling edge, rising edge, 4th rising edge or every 16th rising edge. An event is selected by the control bits CCPxM3:CCPxM0 (CCPxCON<3:0>). When a capture is made, the interrupt request flag bit, CCPxIF, is set. The CCPxIF bit must be cleared in software. If another capture occurs before the value in register CCPRx is read, the previous captured value will be lost.

Note: Timer1 must be running in timer mode or synchronized counter mode for the CCP module to use the capture feature. In asynchronous counter mode, the capture operation may not work.

As can be seen in Figure 2, a capture does not reset the 16-bit TMR1 register. This is so Timer1 can also be used as the time base for other operations. The time between two captures can easily be computed as the difference between the two consecutive capture values. When Timer1 overflows, the TMR1IF bit. An interrupt occurs if and only if the TMR1IE flag was set. In Capture mode, the CCPx (RC2) pin must be configured as an input by setting its corresponding TRIS bit.

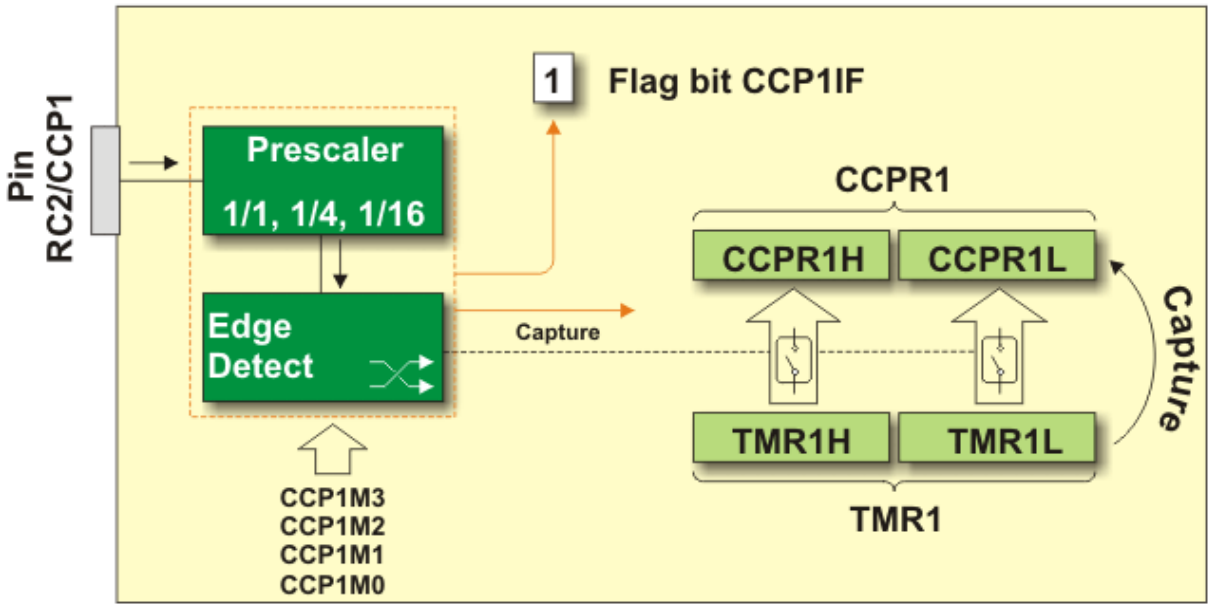

Figure 2: CCP1 in Capture mode

The pre-scaler can be used to get a fine average resolution on a constant input frequency, e.g. if we have a stable input frequency and we set the pre-scaler to 1:16, then the total error for those 16 periods is 1 TCY. This gives an effective resolution of TCY/16, which at 20 MHz is 12.5 ns. This technique is only valid where the input frequency is "stable" over the 16 samples. The flag bit CCP1IF is set when a capture is made. If it happens and if the CCP1IE bit of the PIE register is set, then an interrupt occurs.

When the Capture mode is changed, an undesirable capture interrupts may be generated. In order to avoid that, both a bit enabling CCP1IE interrupt and flag bit CCP1IF should be cleared prior to any change occurring in the control register. Undesirable interrupt may be also generated by switching from one capture pre-scaler to another. To avoid this, the CCP1 module should be temporarily switched off before changing the pre-scaler.

3.2 The compare mode

In this mode, the value in the CCP1 register is constantly compared to the value in the timer register TMR1. When a match occurs, the output pin RC2/CCP1 logic state may be changed, which depends on the state of bits in the control register (CCP1M3 - CCP1M0). The flag-bit CCP1IF will be simultaneously set (See Figure 3).

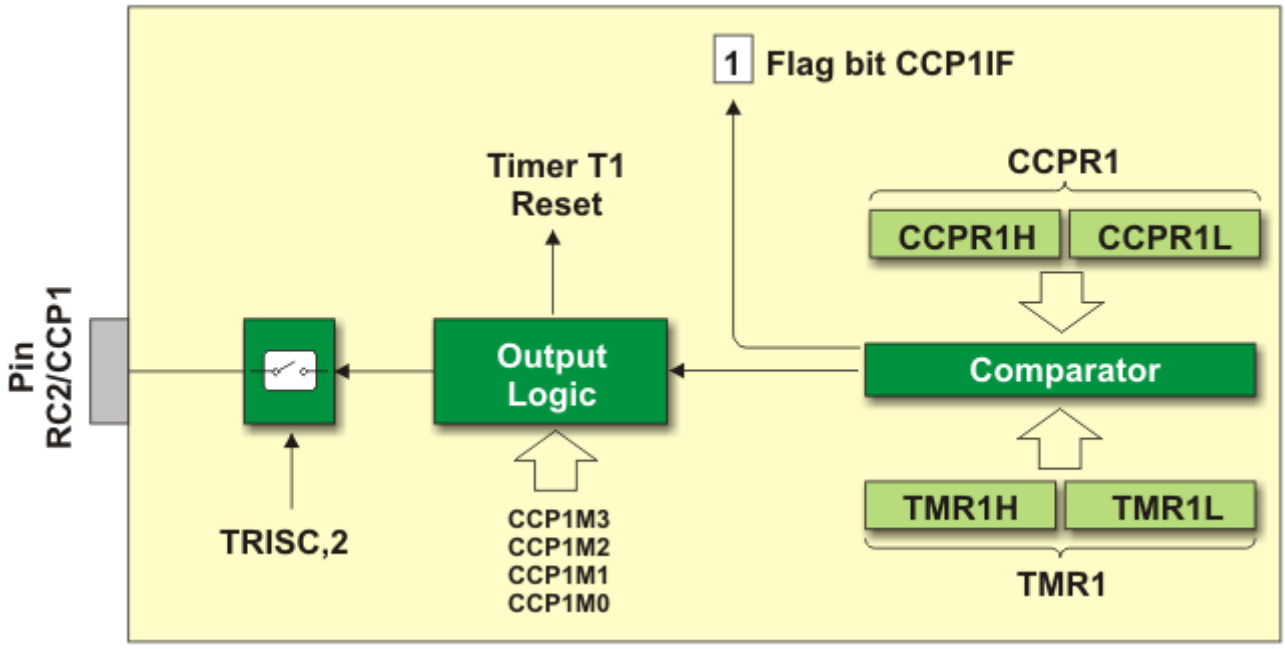

Figure 3: CCP1 in Compare mode

Note: To setup CCP1 module to operate in this mode, two conditions must be met:

1. Pin RC2/CCP1 must be configured as output.
2. Timer TMR1 must be synchronized with internal clock.

3.3 The PWM mode

In Pulse Width Modulation (PWM) mode, the CCPx pin produces up to a 10-bit resolution PWM output. Since the CCPx pin is multiplexed with the PORT data latch, the corresponding TRIS bit must be cleared to make the CCPx pin an output.

Figure 4 shows the block diagram of the CCP1 module setup in PWM mode. In order to generate a pulse of arbitrary form on its output pin, it is necessary to determine only two values, pulse frequency and duration.

3.3.1 PWM period

The output pulse period (T) is specified by the PR2 register of the timer TMR2. The PWM period can be calculated using the following equation:

PWM Period(T) = (PR2 +1) * 4Tosc * TMR2 Prescale Value

If the PWM Period (T) is known then, it is easy to determine the signal frequency F because these two values are related by equation F=1/T.

3.3.2 PWM duty cycle

The PWM duty cycle is specified by using in total of 10 bits: eight MSbs found in the CCPR1L register and two additional LSbs found in the CCP1CON register (DC1B1 and DC1B0). The result is 10-bit number.

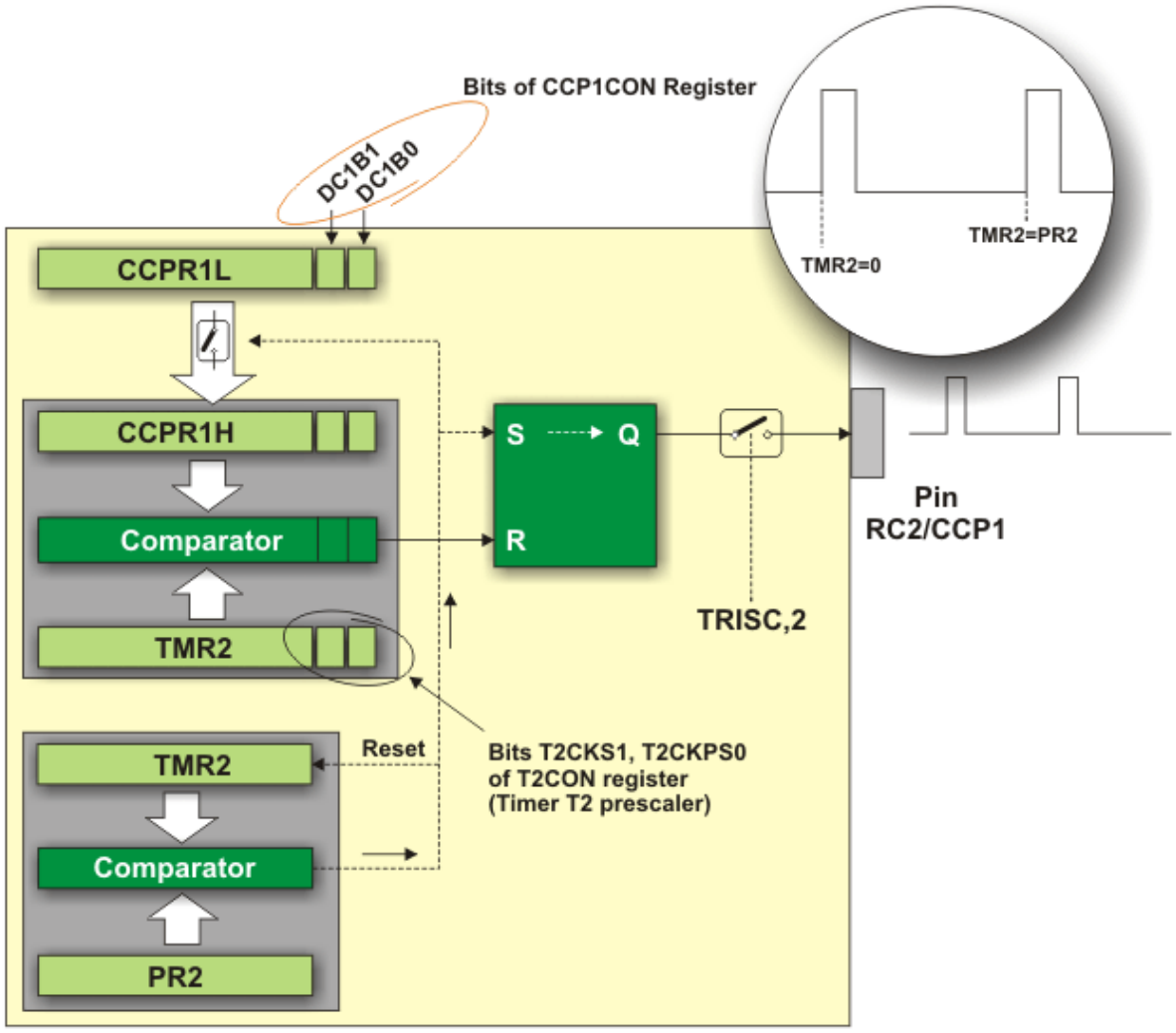

Figure 4: PWM module

Pulse Width (High Part) = (CCPR1L,DC1B1,DC1B0) * Tosc * TMR2 Prescale Value

The following table shows how to generate PWM signals of varying frequency if the microcontroller uses 20 MHz quartz-crystal (Tosc=50nS).

Frequency [KHz]	**1.22**	**4.88**	**19.53**	**78.12**	**156.3**	**208.3**
TMR2 Pre-scaler	16	4	1	1	1	1
PR2 Register	FFh	FFh	FFh	3Fh	1Fh	17h

Note1: Output pin will be constantly set in case the pulse width is by negligence determined to be larger than PWM period; and
Note2: In this application, the timer TMR2 Post-scaler cannot be used for generating longer PWM periods.

3.3.3 PWM resolution

PWM signal is nothing more than the pulse sequence with varying duty cycle. For one specified frequency (number of pulses per second), there is a limited number of duty cycle combinations. This number is called resolution measured by bits. For example, a 10-bit resolution will result in 1024 discrete duty cycles, whereas an 8-bit resolution will result in 256 discrete duty cycles etc. In relation to this microcontroller, the resolution is specified by the PR2 register. The maximal value is obtained by writing number FFh.

PWM frequencies and resolutions (Fosc = 20MHz):

PWM Frequency	**1.22kHz**	**4.88kHz**	**19.53kHz**	**78.12kHz**	**156.3kHz**	**208.3kHz**
Timer Prescale	16	4	1	1	1	1
PR2 Value	FFh	FFh	FFh	3Fh	1Fh	17h
Maximum Resolution	10	10	10	8	7	6

PWM frequencies and resolutions (Fosc = 8MHz):

PWM Frequency	**1.22kHz**	**4.90kHz**	**19.61kHz**	**76.92kHz**	**153.85kHz**	**200.0kHz**
Timer Prescale	16	4	1	1	1	1
PR2 Value	65h	65h	65h	19h	0Ch	09h
Maximum Resolution	8	8	8	6	5	5

3.4 Timer1 module

The Timer1 module is a 16-bit timer/counter consisting of two 8-bit registers (TMR1H and TMR1L) which are readable and writable. The TMR1 Register pair (TMR1H:TMR1L) increments from 0000h to FFFFh and rolls over to 0000h. The Timer1 Interrupt, if enabled, is generated on overflow which is latched in the TMR1IF interrupt flag bit. This interrupt can be enabled/disabled by setting/clearing the TMR1IE interrupt enable bit. Timer1 can operate in one of three modes: as a synchronous timer, synchronous counter, or as an asynchronous counter. The operating mode is determined by clock select bit, TMR1CS (T1CON<1>), and the synchronization bit, T1SYNC (See Figure 5).

In timer mode, Timer1 increments every instruction cycle. In counter mode, it increments on every rising edge of the external clock input on pin T1CKI. Timer1 can be turned on and off using theTMR1ON control bit (T1CON<0>). Timer1 also has an internal “reset input”, which can be generated by a CCP module. Timer1 has the capability to operate off an external crystal. When the Timer1 oscillator is enabled (T1OSCEN is set), the T1OSI and T1OSO pins become inputs. That is, their corresponding TRIS values are ignored.

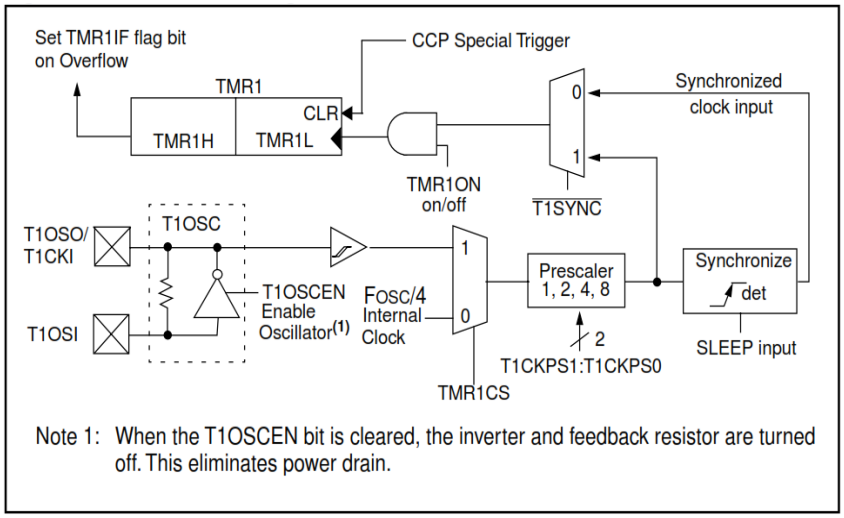

Figure 5: Timer1 Block Diagram

T1CON: Timer1 Control Register

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON
bit 7							bit 0

bit 7:6 **Unimplemented:** Read as '0'

bit 5:4 **T1CKPS1:T1CKPS0**: Timer1 Input Clock Prescale Select bits

11 = 1:8 Prescale value
10 = 1:4 Prescale value
01 = 1:2 Prescale value
00 = 1:1 Prescale value

bit 3 **T1OSCEN**: Timer1 Oscillator Enable bit

1 = Oscillator is enabled
0 = Oscillator is shut off. The oscillator inverter and feedback resistor are turned off to eliminate power drain

bit 2 **T1SYNC**: Timer1 External Clock Input Synchronization Select bit

When TMR1CS = 1:

1 = Do not synchronize external clock input
0 = Synchronize external clock input

When TMR1CS = 0:

This bit is ignored. Timer1 uses the internal clock when TMR1CS = 0.

bit 1 **TMR1CS**: Timer1 Clock Source Select bit

1 = External clock from pin T1OSO/T1CKI (on the rising edge)
0 = Internal clock (FOSC/4)

bit 0 **TMR1ON**: Timer1 On bit

1 = Enables Timer1
0 = Stops Timer1

Legend		
R = Readable bit	W = Writable bit	
U = Unimplemented bit, read as '0'		- n = Value at POR reset

Figure 6: Timer1 Block Diagram

3.5 Timer2 module

Timer2 is an 8-bit timer with a pre-scaler, a post-scaler, and a period register. Using the pre-scaler and post-scaler at their maximum settings, the overflow time is the same as a 16-bit timer. Timer2 is the PWM time-base when the CCP module(s) is used in the PWM mode. Figure 7 shows a block diagram of Timer2. The post-scaler counts the number of times that the TMR2 register matched the PR2 register. This can be useful in reducing the overhead of the
Interrupt service routine on the CPU performance.

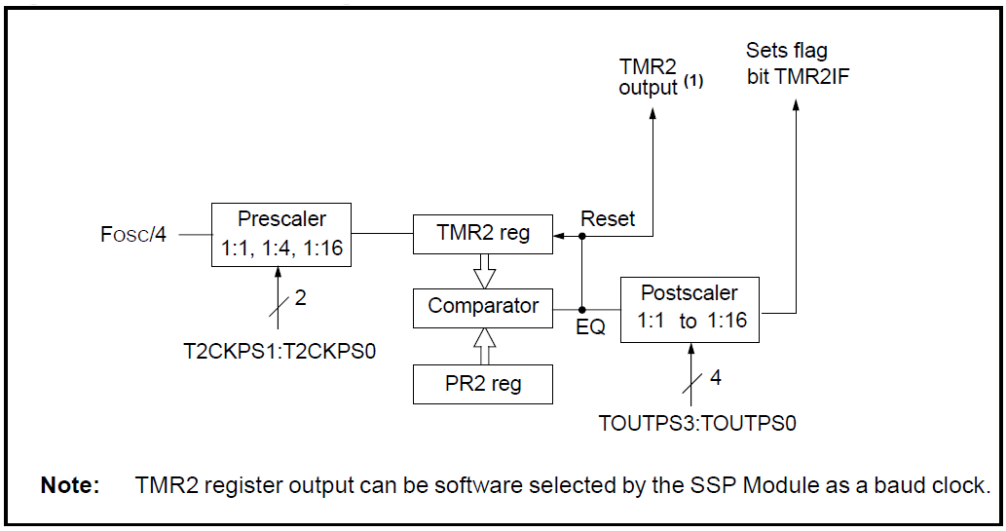

Figure 6: Timer2 Block Diagram

T2CON: Timer2 Control Register

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

bit 7 **Unimplemented:** Read as '0'

bit 6:3 **TOUTPS3:TOUTPS0**: Timer2 Output Postscale Select bits

0000 = 1:1 Postscale
0001 = 1:2 Postscale
•
•
•
1111 = 1:16 Postscale

bit 2 **TMR2ON**: Timer2 On bit

1 = Timer2 is on
0 = Timer2 is off

bit 1:0 **T2CKPS1:T2CKPS0**: Timer2 Clock Prescale Select bits

00 = Prescaler is 1
01 = Prescaler is 4
1x = Prescaler is 16

Legend		
R = Readable bit	W = Writable bit	
U = Unimplemented bit, read as '0'		- n = Value at POR reset

Figure 7: Timer2 Control Register

4. Procedure

4.1 Capturing a signal using capture mode

1. Open **MPLAB IDE**.
2. Make new project, and add the codes **delay.h**, **delay.c, lcd.h**, **lcd.c**, **string.h**, **string.c** and **capture.c** provided in the experiment codes section.
3. Compile the project using **HI-TECH UNVERSAL ANSI COMPILER** for **PIC16F877A**.
4. Open Proteus ISIS.
5. Under components, list the following: **PIC16F877A**, **POT-LN** or **POT-HG**, **RES**, **LM016L**, **CRYSTAL** and **CAP**.
6. Under terminal mode: use the **POWER** and **GND**.
7. Under instruments, pick a **SIGNAL GENERATOR** and **OSCILLOSCOPE**. These devices results usually pop-up at simulation time. In case they didn't, you can always display them by clicking on **Debug**, then ticking the devices listed on the bottom of the list. All this happens only at simulation time.
8. Implement circuit provided in Figure 8.

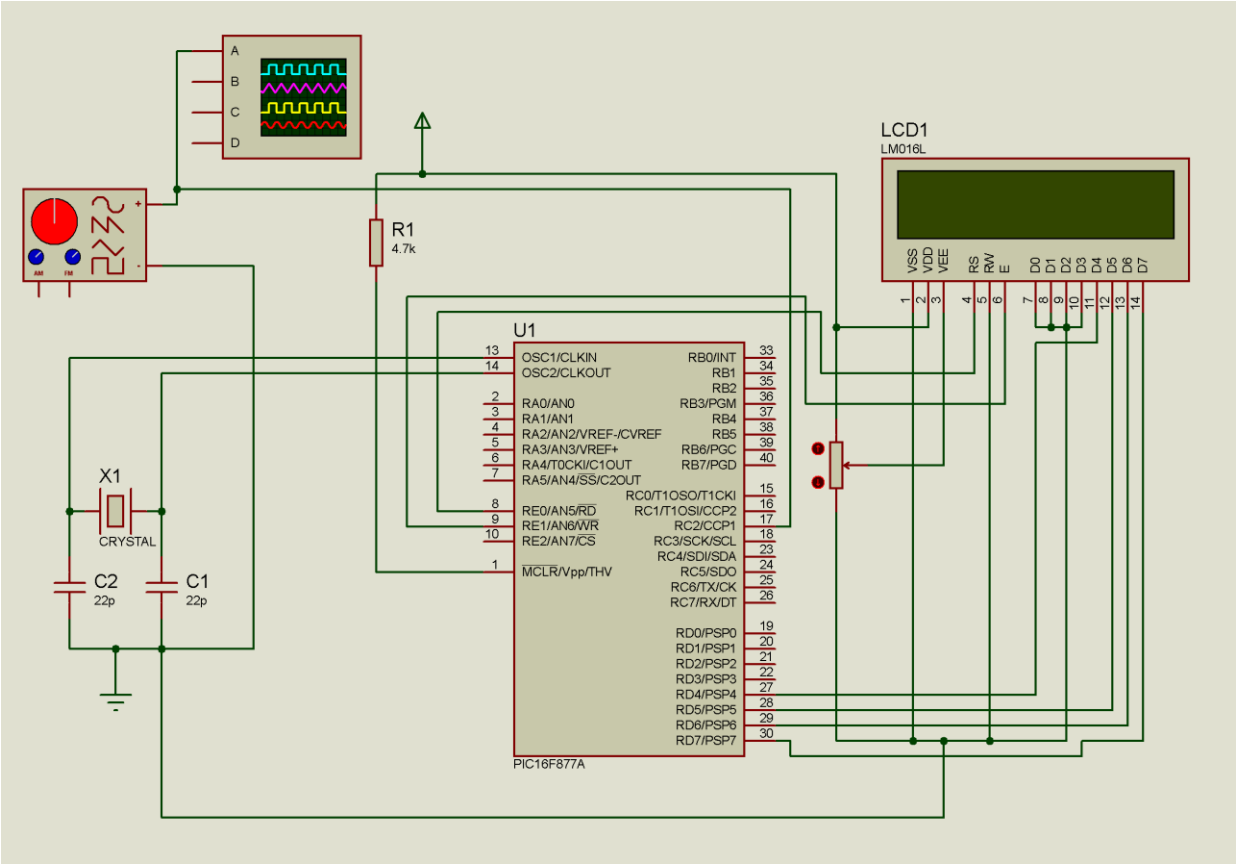

Figure 8: Capture mode connections

9. Load the generated **HEX** onto the micro-controller. Set its external frequency to **4MHz**.

10. Click on **"play"**.
11. Set the values according to the Figure 9. Observe the outputs you get.

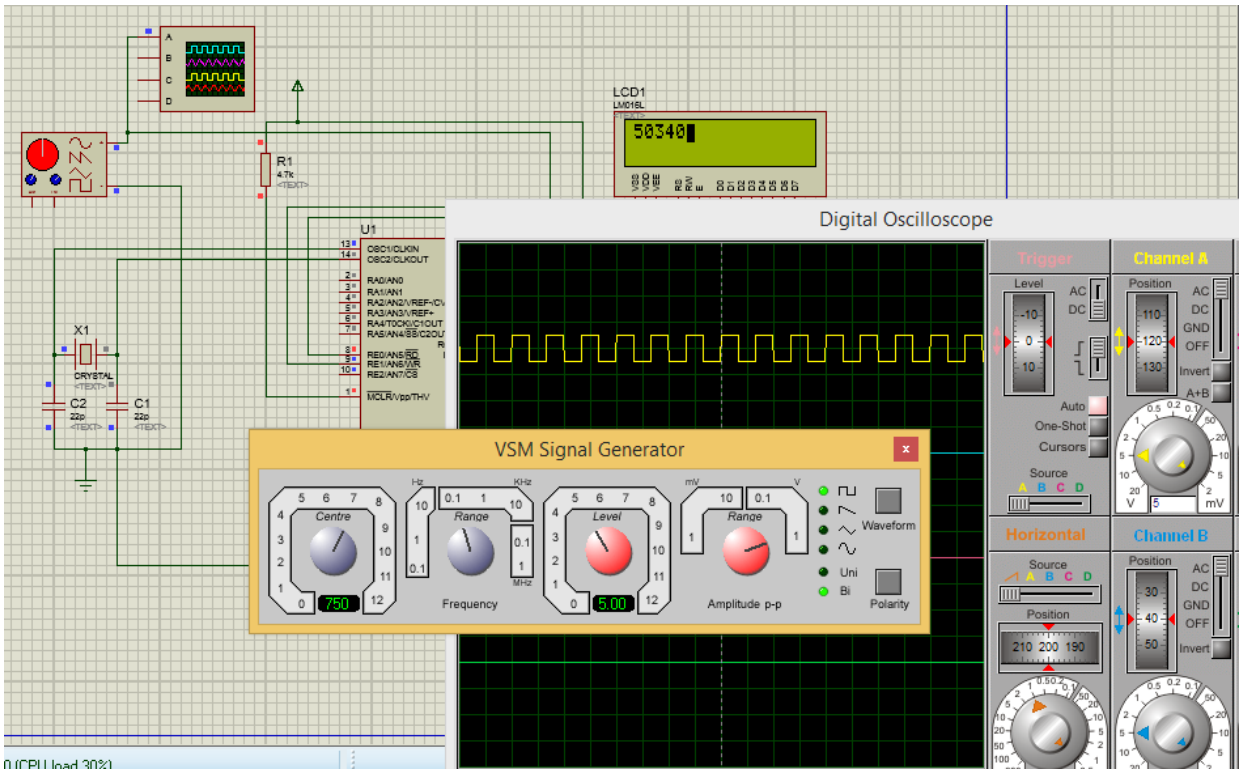

Figure 9: Capture mode results

12. Change the values to ones of your own choice and observe how the outputs change.
13. Download the program on the **PIC16F877A** using **PicKit3**.
14. Apply the connections on hardware.
15. Observe the outputs you get and compare them to the simulated ones.

4.1 Generating 15KHz signal using PWM mode

16. Open **MPLAB IDE**.
17. Make new project, and add the code **pwm.c** provided in the experiment codes section.
18. Compile the project using **HI-TECH UNVERSAL ANSI COMPILER** for **PIC16F877A**.
19. Open Proteus ISIS.
20. Under components, list the following: **PIC16F877A**, **RES**, **CRYSTAL** and **CAP**.
21. Under terminal mode: use the **POWER** and **GND**.
22. Under instruments, pick an **OSCILLOSCOPE**. This device results usually pop-up at simulation time. In case they didn't, you can always display them by clicking on **Debug**, then ticking the device listed on the bottom of the list. All this happens only at simulation time.
23. Implement circuit provided in Figure 10.
24. Load the generated **HEX** onto the micro-controller. Set its external frequency to **4MHz**.
25. Click on **"play"** and observe the output signal displayed on the oscilloscope.

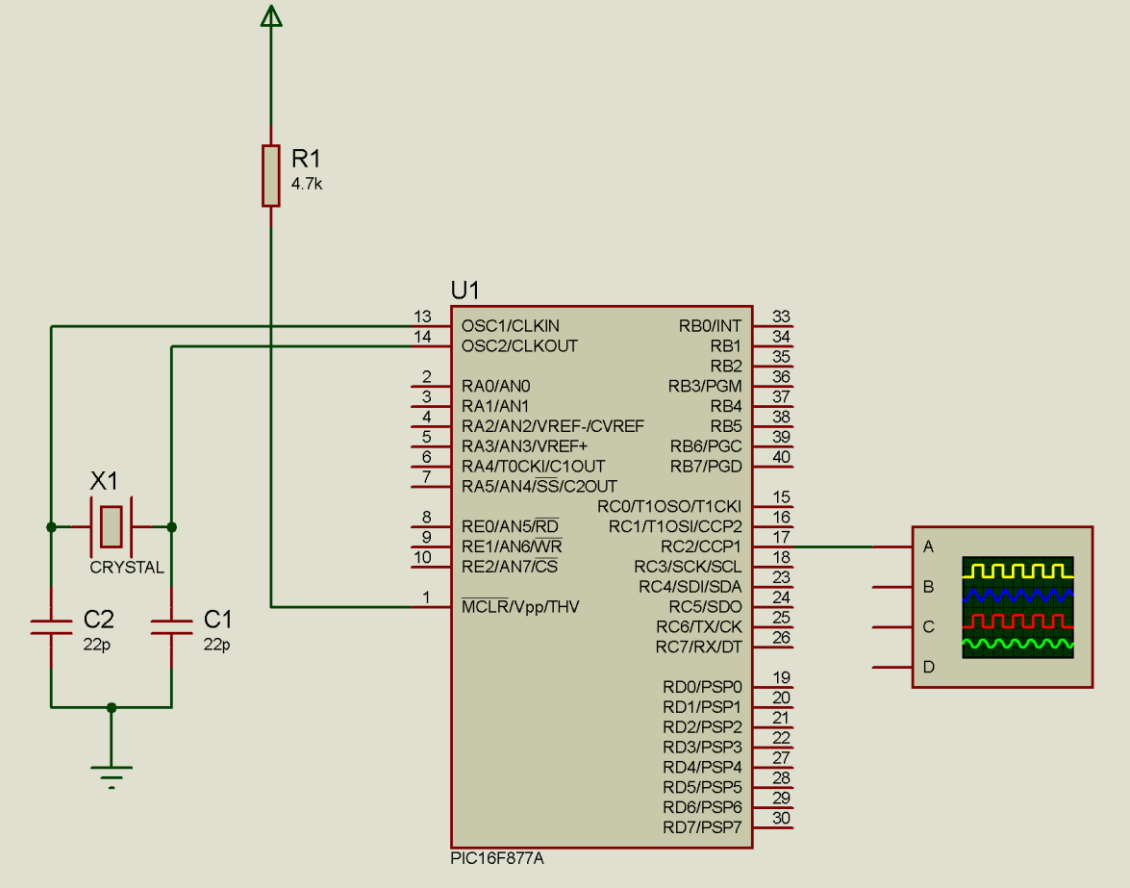

Figure 10: PWM mode connections

26. Note that the signal is not exactly 15KHz. Also it quite closer to 14KHz instead. There are two reasons for this large error percentage, can you guess them?

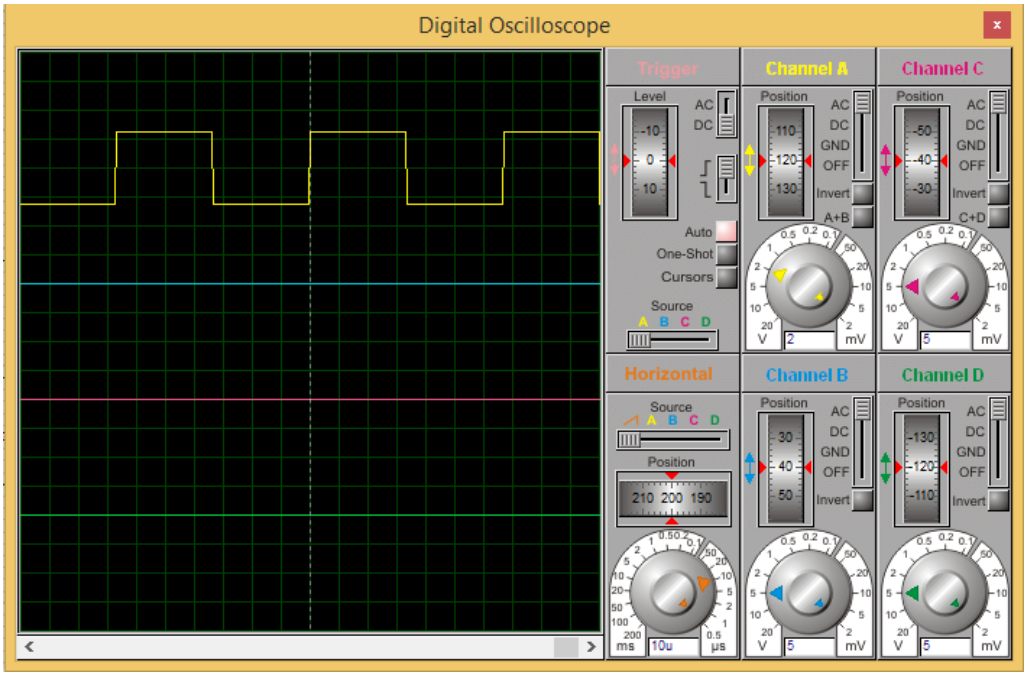

Figure 11: PWM mode results

27. Download the program on the **PIC16F877A** using **PicKit3**.
28. Apply the connections on hardware.
29. Observe the outputs you get and compare them to the simulated ones.

5. Experiment Codes

Delay.h

```
/*
 *    Delay functions for HI-TECH C on the PIC
 *
 *    Functions available:
 *          DelayUs(x)  Delay specified number of microseconds
 *          DelayMs(x)  Delay specified number of milliseconds
 *
 *    Note that there are range limits: x must not exceed 255 - for xtal
 *    frequencies > 12MHz the range for DelayUs is even smaller.
 *    To use DelayUs it is only necessary to include this file; to use
 *    DelayMs you must include delay.c in your project.
 *
 */

/*    Set the crystal frequency in the CPP predefined symbols list in
      HPDPIC, or on the PICC commmand line, e.g.
      picc -DXTAL_FREQ=4MHZ

      or
      picc -DXTAL_FREQ=100KHZ

      Note that this is the crystal frequency, the CPU clock is
      divided by 4.

 *    MAKE SURE this code is compiled with full optimization!!!

 */

#ifndef     XTAL_FREQ
#define     XTAL_FREQ   4MHZ        /* Crystal frequency in MHz */
#endif

#define     MHZ   *1000L                  /* number of kHz in a MHz */
#define     KHZ   *1                /* number of kHz in a kHz */

#if   XTAL_FREQ >= 12MHZ

#define     DelayUs(x)  { unsigned char _dcnt; \
                        _dcnt = (x)*((XTAL_FREQ)/(12MHZ)); \
                        while(--_dcnt != 0) \
                              continue; }
#else

#define     DelayUs(x)  { unsigned char _dcnt; \
                        _dcnt = (x)/((12MHZ)/(XTAL_FREQ))|1; \
                        while(--_dcnt != 0) \
                              continue; }
#endif

extern void DelayMs(unsigned char);
```

Delay.c

```
/*
 *     Delay functions
 *     See delay.h for details
 *
 *     Make sure this code is compiled with full optimization!!!
 */

#include "delay.h"

void DelayMs(unsigned char cnt)
{
#if   XTAL_FREQ <= 2MHZ
      do {
            DelayUs(996);
      } while(--cnt);
#endif

#if    XTAL_FREQ > 2MHZ
      unsigned char     i;
      do {
            i = 4;
            do {
                  DelayUs(250);
            } while(--i);
      } while(--cnt);
#endif
}
```

Lcd.h

```
extern void lcd_write(unsigned char);
extern void lcd_clear(void);
extern void lcd_puts(const char * s);
extern void lcd_goto(unsigned char pos);
extern void lcd_init(void);
extern void lcd_putch(char);
#define lcd_cursor(x) lcd_write(((x)&0x7F)|0x80)
```

Lcd.c

```
/*
*     the list of included files contains:
*     pic.h since we're going to use it with our pic microcontroller, duah
*     ldc.h which contains all the prototypes of the functions used for the
lcd
*     delay.h which contains the prototype of the melli-second delay and
the implementation
```

```
*     of the micro-second delay (which is used in the melli-second
implementation, so it has to be included)
*     delay.c contains the implementation of the delay_ms function, so it
has to be here
*/

#include <pic.h>
#include "lcd.h"
#include "delay.h"
#define LCD_STROBE ((RE1 = 1),(RE1=0))    //* The E bit on the lcd, where
it tells the lcd that we're writing data to it when it's set to 1*/
/*
*     the following write functions take a character inpute or 8 bits where
it takes the higher 4 bits and passes their values to the D port of the pic
*     then it shifts the character by 4 bits to the left in order to take
the values of the lower 4 bits and put them on the used D port bits. The
strobe
*     is used indicate that the LCD is receving data so that the values are
guaranteed to be passed in order and without interference
*
*     As for the RS bit which is connected to RE0. It is used to tell the
LCD to accept the character as a command when it's set to 0, or as a
character to
*     be displayed on the screen when it's set to 1
*/
void lcd_write(unsigned char c)
{
PORTD = (PORTD & 0x0F) | (c);
LCD_STROBE;
PORTD = (PORTD & 0x0F) | (c << 4);
LCD_STROBE;
DelayUs(40);
}
void lcd_clear(void)
{
RE0 = 0;
lcd_write(0x1);
DelayMs(2);
}
void lcd_puts(const char * s)
{
RE0 = 1;
while(*s)
lcd_write(*s++);
}
void lcd_putch(char c)
{
RE0 = 1;
PORTD = (PORTD & 0x0F) | (c);
LCD_STROBE;
PORTD = (PORTD & 0x0F) | (c << 4);
LCD_STROBE;
DelayUs(40);
}
void lcd_goto(unsigned char pos)
{
RE0 = 0;
lcd_write(0x80+pos);
}
```

```
void lcd_init(void)
{
RE0 = 0;
DelayMs(15); // power on delay
PORTD = (0x3 << 4);
LCD_STROBE;
DelayMs(5);
LCD_STROBE;
DelayUs(100);
LCD_STROBE;
DelayMs(5);
PORTD = (0x2 << 4);
LCD_STROBE;
DelayUs(40);
lcd_write(0x28); // 4 bit mode, 1/16 duty, 5x8 font
lcd_write(0x08); // display off
lcd_write(0x0F); // display on, blink curson on
lcd_write(0x06); // entry mode
}
```

String.h

```
#include <pic.h>
extern void reverse(char *str, int len);
extern unsigned int intToStr(unsigned int x, char str[], int d);
extern void ftoa(float n, char *res, int afterpoint);
```

String.c

```
// C program for implementation of ftoa()
#include <pic.h>
#include<stdio.h>
#include<math.h>
#include "string.h"

// reverses a string 'str' of length 'len'
void reverse(char *str, int len)
{
    int i=0, j=len-1, temp;
    while (i<j)
    {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
        i++; j--;
    }
}

 // Converts a given integer x to string str[].  d is the number
 // of digits required in output. If d is more than the number
 // of digits in x, then 0s are added at the beginning.
unsigned int intToStr(unsigned int x, char str[], int d)
{
    unsigned int i = 0;
    while (x)
    {
        str[i++] = (x%10) + '0';
        x = x/10;
```

```
    }

    // If number of digits required is more, then
    // add 0s at the beginning
    while (i < d)
        str[i++] = '0';

    reverse(str, i);
    str[i] = '\0';
    return i;
}

// Converts a floating point number to string.
void ftoa(float n, char *res, int afterpoint)
{
    // Extract integer part
    int ipart = (int)n;

    // Extract floating part
    float fpart = n - (float)ipart;

    // convert integer part to string
    int i = intToStr(ipart, res, 0);

    // check for display option after point
    if (afterpoint != 0)
    {
        res[i] = '.';  // add dot

        // Get the value of fraction part upto given no.
        // of points after dot. The third parameter is needed
        // to handle cases like 233.007
        fpart = fpart * pow(10, afterpoint);

        intToStr((int)fpart, res + i + 1, afterpoint);
    }
}
```

Capture.c

```
#include <pic.h>
#include <stdlib.h>
#include <stdio.h>
#include "lcd.h"
#include "delay.h"
#include "string.h"


__CONFIG(DEBUG_OFF & WDTE_OFF & LVP_OFF & FOSC_HS & BOREN_ON);

unsigned int  rising_edge_time=0; // hold the current value of rising edge
time
char str[10];

void main(void) {
      ADCON1 = 7; // Set A/D-pins as digital I/O
      nRBPU = 0;   // enable internal pullups on PORTB
      TRISE = 0;  // set port E in output mode
```

```
      TRISD = 0;  // set port D in output mode
      RE2 = 0;    // Initialize PortE pin 2


      //Timer1 configuration
      TMR1CS = 0; // use internal clock  (Fosc/4) , where Fosc= 4MHz in our
case
      TMR1IF = 0; // set Timer interrupt flag bit to 0. It is set to 1 when
an overflow is occurred in Timer1
      TMR1H  = 0x00;  // the MSB 8 bit for Timer1 , which has the value of
the timer
      TMR1L = 0x00; // the LSB 8 bit for Timer1.
      T1CKPS0 = 0;  T1CKPS1 = 0;  //set the prescale value to 1:1 (See
datasheet)
      T1SYNC = 0;// Timer is synchronized to external clock input
      TMR1IE = 1;// enable TIMER1 interrupt
      TMR1ON = 1; //Enable Timer1

      //Configure CCP module to Capture mode
      CCP1M3 = 0; CCP1M2 = 1; CCP1M1 = 0; CCP1M0 = 1;  // Capture the input
signal on every rising edge
      CCP1IE = 1; // enable CCP1 interrupt
      CCP1IF = 0;

      GIE = 1;  // Global Interrupt Enable
      PEIE = 1;

      DelayMs(100);
      lcd_init();
      lcd_clear();

      //forever loop to display the rising edge time
      while(1) {
            DelayMs(100);
            lcd_clear();
            intToStr((unsigned int)rising_edge_time, str, 5);
            lcd_puts(str); /* display value on LCD */
      }

}

//the interrupt function is called , when an overflow is occurred in Timer1
or the rising or falling edge is detected depends on the configuration of
CCP module
static void interrupt ISR(void) {
      //CCP1IF
  // check if the interrupt is 1
  if ( CCP1IF ) {
      //store the time of rising or falling edge in  rising_edge_time by
getting the time value from CCPR1L
      rising_edge_time=0;
      rising_edge_time= (int) CCPR1;
      //rest the CCP1 interrupt flag to 0
    CCP1IF = 0;
  }

  //check if the Timer1 is overflowed
  if ( TMR1IF ) {
      //rest the Timer1 counter to 0
```

```
    TMR1H = 0x00;  TMR1L = 0x00;
      //reset Timer1 overflow flag to 0
    TMR1IF = 0;
  }
}
```

Pwm.c

```
#include <pic.h>

__CONFIG(DEBUG_OFF & WDTE_OFF & LVP_OFF & FOSC_HS & BOREN_ON);

int i=0;
void main(void) {
      TRISC = 0 ; // set PORTC as output
      PORTC = 0 ; // clear PORTC

      /* configure CCP module as 15 KHz PWM output */
      PR2 = 66 ; //set PR2 Regesiter to  66 decimal
      T2CON =   0b00000100; //set Timer2 control Register , where Prescaler
is 1 and Postscale is 1:1
      CCP1CON = 0b00101100; //set CCP1 to Pulse Width Modulation Mode

      for(;;){
            /* PWM resolution is 10 bits, so only CCPRxL have to be touched
to change duty cycle */
            /* set CCPR1L to 25 to give 50% duty cycle with frequency = 15
KHz , note that DC1B1 and DC1B0 are set to 10 in CCP1CON */
            CCPR1L = 0b00100001;
      }
}
```