11.1 PIC Microcontroller Unit (PIC MCU):

PIC is generally assumed to mean Peripheral Interface Controller (PIC), it comes with a variety of families; PIC10 and PIC12 (Base-line), PIC16 (Mid-range), PIC17 (High-end), PIC18 (enhancement), Finally PIC24 and dsPIC. Here we will deeply look at PIC16 family and highlights on other families' features if needed. PIC16F877A is our interest MCU in this family.

- 8-bit microcontrollers
  - PIC10
  - PIC12
  - PIC14
  - PIC16
  - PIC17
  - PIC18
- 16-bit microcontrollers
  - PIC24F
  - PIC24H
- 32-bit microcontrollers
  - PIC32
- 16-bit digital signal controllers
  - dsPIC30
  - dsPIC33F

11.2 PIC16F877A (PIC16) identification:

PIC16F877A is very cheap, also very easy to be assembled. Additional components that you need to make this IC work are just a 5V power supply adapter, a 20MHz crystal oscillator and 2 units of 22pF capacitors.
PIC16F877A is a 40 pin chip, operating at a frequency up to 20MHz, it has five Bidirectional I/O ports A(6-bit), B(8-bit), C(8-bit), D(8-bit), E(3-bit) mapping to 33 pins, the following points highlight the most important features:

1- 8Kx14bit Program memory space.
2- Five I\O ports.
3- 8 multiplexed analog ports, with internal 10bit resolution ADC.
4- 15 kinds of interrupts.
5- 256 Bytes of user EEPROM.
6- Two Capture\Compare\PWM modules (CCP).
7- Three timers with different capabilities.
8- RS-232, I2C, and SPI interfaces (USART, MSSP).
9- 368B of RAM.
10- Wide operating frequency DC-20MHz.
11- Wide operating voltage 2.0v – 5.5v.

In addition, they have the following alternate functions:

Port	Alternative Uses of I/O Pins	No of I/O Pins
Port A	A/D Converter Inputs	6
Port B	External Interrupt Inputs	8
Port C	Serial Port, Timer I/O	8
Port D	Parallel Slave Port	8
Port E	A/D Converter Inputs	3
	Total I/O Pins	33
	Total Pins	40

Note that a single pin can have many functions, for example pin2 can be a digital I/O (RA0) or analog input (AN0); the function of the pin will be controlled using software, figure 11.1 illustrates the PIC16F877A layout in more details.

PIC16F877A as other PIC families is implemented using RISC approach, with only 35 instructions; you can build great projects, security, control, talking with GSM system, linking to the internet, communicating with PCs and more.

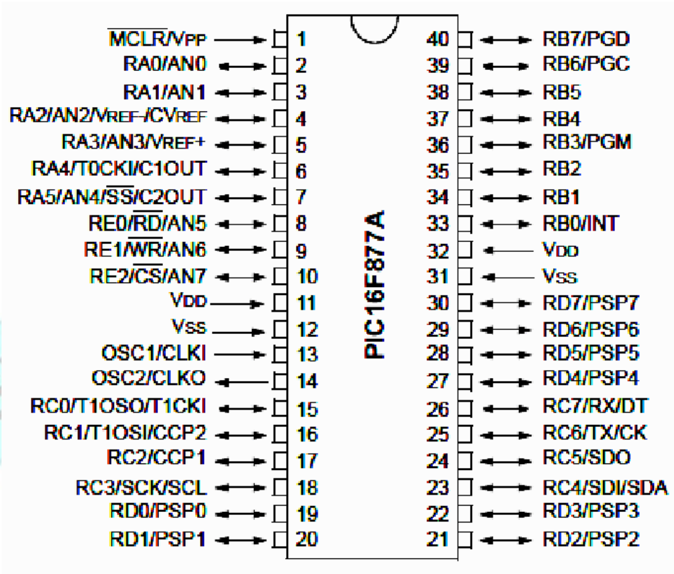


Fig. 11.1: PIC16F877A Pins Layout.

Many engineers and developers choose it in their projects and designs for three main reasons:

1. PIC16F877A comes with a variety of embedded modules.
2. PIC16F877A is considered a low cost MCU.
3. It has wide supporting articles in the internet.

The F letter in its name stand for Flash technology, Flash EEPROM, a version of EEPROM memory, has become popular in microcontroller applications and is used to store the user program. Flash EEPROM is nonvolatile and usually very fast. The data can be erased and then reprogrammed using a suitable programming device thousands and thousands of times. Letter A at the end of PIC16F877A means that this MCU is an Advanced and an improved version of a previous MCU i.e. PIC16F877.

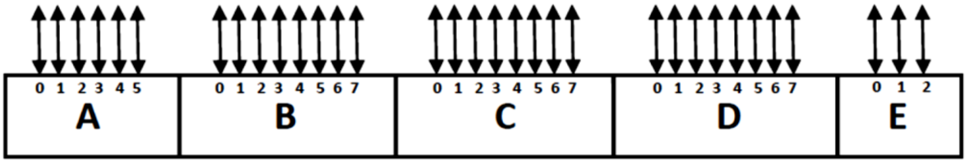
With 8K program memory, PIC16F877A can store and run programs ranges from simple (a few lines), mid, up to complex programs with many hundreds of lines, also 368B of general-purpose registers (GPR) i.e., user RAM; this size of temporary storage area can maintain and take complex operations either arithmetic or logic, float or integer, strings or characters.

Knowing the internal architecture of any MCU is a must if you want to use assembly for writing programs (in ES terminology called Firmware), contrarily of using a high level language like C; a very little architecture knowledge is needed to write a perfect firmware.

Finally, all hardware, software aspects and the embedded modules of PIC16F877A will be taken and explained using examples and some projects.

11.3 Digital I/O (Part One):

The first step for mastering any MCU is having a knowledge of how to use pins for digital input and output, as we say previously PIC16F877A has 33 I/O pins partitioned into 5 ports and each port has a specific number of pins i.e. A(6), B(8), C(8), D(8) and E(3) as shown in figure below, note that we can access the port as a whole or simply treating each pin individually.



Note: the term **bit mode** will be used when dealing with a single pin either for input or output, for ports we will use **byte mode** term; although ports **A** and **E** have less than 8 bits I/O.
PIC C compiler comes with a variety functions for dealing with digital I/O for ports and pins, table below shows the most important functions, all of these functions and more will be handled and explained using examples.

Function	Description
Output_bit(pin, bit)	Output 1(high voltage) or 0(low voltage) to the specified pin.
Input(pin)	Returns 1 or 0 corresponding to current pin status.
Output_*X*(data)	X is port name i.e. A,B,C,D,E . data is 8 bit (1 Byte).
Input_*X*( )	Returns 1 byte of data according to each pin state of port X.

In this section many examples and peripherals will be examined:

1. **Using Output function:** This example shows how to use simple digital output functions in bit mode, time delay functions, and show many of Proteus aspects, each line in this example is explained in detail.
2. **Using I/O functions:** Shows more digital I/O functions, more peripherals.
3. **Interfacing with 7-segment display:** More H/W, more S/W, but friendly more; in overall: good interfacing and programming practice.

Each example will explain something either for PIC, compiler, or Proteus. The code of the first example explained line by line, so I will step over any similar lines in the next examples either in this section or other sections.

11.3.1 Using Output functions:

Example name: Flashing LED.
Main goals: Introduction to PIC C and Proteus, using bit mode functions.
DESCRIPTION: A LED (**L**ight **E**mitting **D**iode) is connected to PIC16F877A at pin **RB7**, initially it will be ON, and after a delay of 0.5 second its state will be changed to OFF, then after half of second too; it will be toggled to ON and so on……., in other words the state of the LED will be changed every 0.5 second.

```
#include <16F877A.h>
#FUSES XT
#USE DELAY(CLOCK = 4000000)
//========================
void main()
{
    set_tris_b(0x7F);
    while(1)
    {
        output_high(pin_b7);
        delay_ms(500);
        output_low(pin_b7);
        delay_ms(500);
    }
}
```

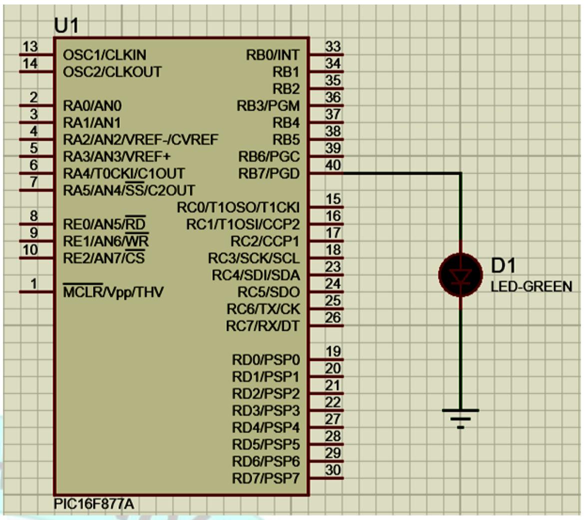


Figure 11.2: Flashing LED code and layout.

11.3.2 Modifications:

1. If we want to use pin **RC3** instead of **RB7** then, write down lines before and after modifications.

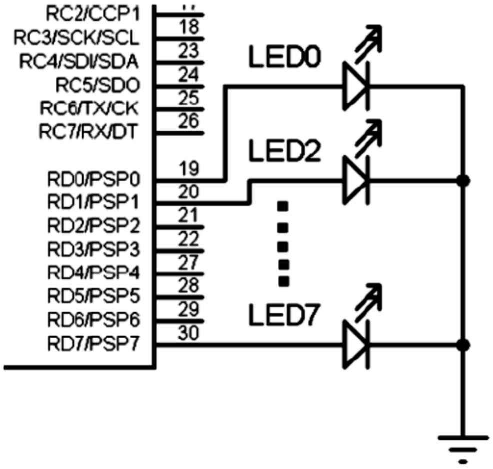
Before modification (line or keyword)		After Modification
set_tris_b(0x7F)	→	set_tris_c(0xF7)
pin_b7	→	pin_c3

2. Use another equivalent functions that included in **PIC C** to perform the same operation with one second delay, write down lines before and after modification, **suppose that the LED is connected to pin RD2**.

Before modification		After Modification
set_tris_b(0x7F)	→	set_tris_d(0xFB)
output_high(pin_b7)	→	output_bit(pin_d2,1)
delay_ms(500);	→	delay_ms(1000);
output_low(pin_b7)	→	output_bit(pin_d2,0)

3. **Example One:** Suppose now that there are 8 LEDs connected to port D, LED0 on RD0, and LED1 on RD1 and so on…, as shown below. Write down a code that performs a nibble (4 bit) toggling, if LED0 to LED3 are **ON** then LED4 to LED7 are **OFF** and vice versa, **with 300mS delay**, note that all pins of port D must be output, and you must deal with 8 bits(one Byte) at a time.

```
#include <16F877A.h>
#FUSES XT
#USE DELAY(CLOCK = 4000000)
Void main()
{
   set_tris_d(0x00); //PORT D is output
   while(1)
   {
      output_d(0x0F); //LEDs 0,1,2 and 3 are ON
      delay_ms(300);
      output_d(0xF0); //LEDs 4,5,6 and 7 are ON,
                      // others are OFF.
      delay_ms(300); //300 mS delay time.
   }
} //end main.
```



Note: output_X(BYTE) is a function that deals with the whole port. X: A,B,C,D, or E

11.3.3 Using I/O functions:

Note: I will explain the new lines of code only, any lines explained previously will not be handled.

In the previous example we show how to use digital output functions in **bit mode** only, also in the modification part for the same example we highlight some of **byte mode** topics. The next example explains how to use simple digital I/O functions either in bit mode, or in byte mode.

Example Two: Port reflection.

Description: 8 LEDs are connected to port **D** and 8 switches to **port B**. Port D(LEDs) must reflect port B(Switches). For example, if switches on RB1 and RB6 are **high** then LEDs on RD1 and RD6 will be **ON** too.

From previous description port D must be output and port B must be input. Schematic is shown in figure 11.10.

Instead of using 8 LEDs and 8 switches individually, I decided to introduce new peripherals. **LED BAR** is working as an array of sequential LEDs it can be seen in many devices as a graphical guide to demonstrates a level of some factor such as volume (Sound level) in stereos or recorders, velocity in cars, temperature and so on….., also **DIPS** (**D**ual **I**n **P**ackage **S**witch) is an array of switches and same as a switch it must be pulled-up or pulled-down, it has many advantages such that simplicity of wiring and small size.

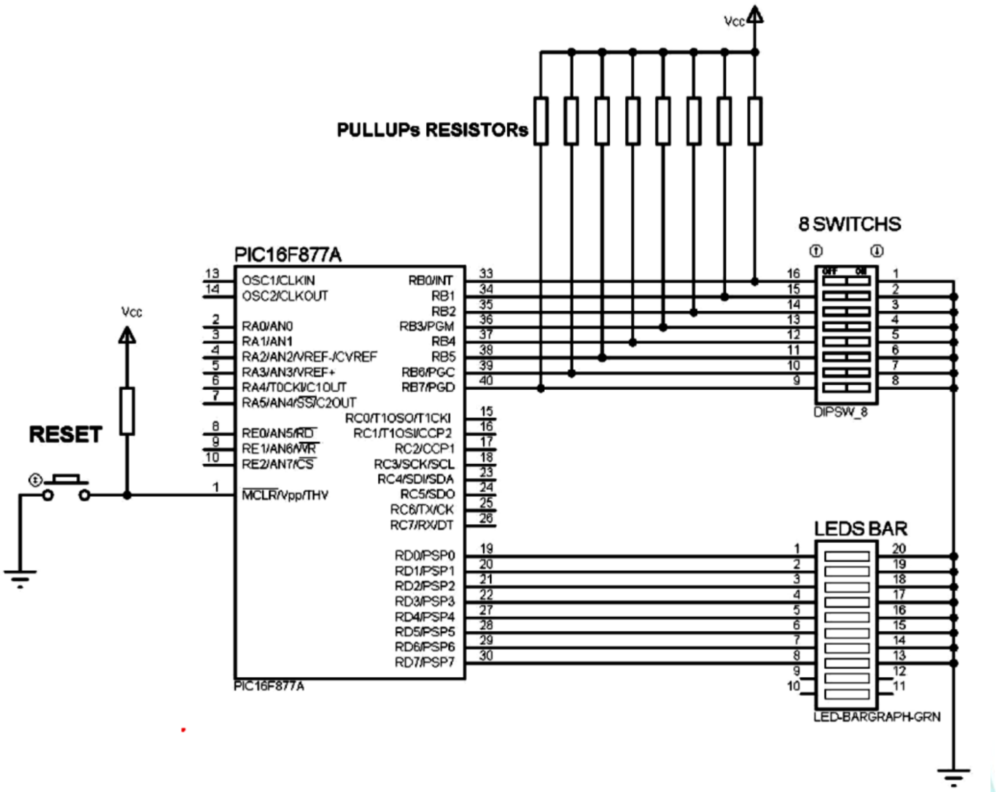


Figure 11.3: I/O schematic layout.

Pulled-up means that the output of the switch is normally high and it goes to low whenever it pressed; reset button must be pulled-up. In pulled-up switches the resistors must be connected to **V**CC from one side and to the target pin and button from the other side; as shown above, in pulled-down, just, replace **V**CC with **GND** and vice-versa.

The code of **port reflection** is as follows:

```
//Code begins here:
#include <16F877A.h>
#FUSES XT
#USE delay (clock = 4000000)
void main()
{
   //Define variables
   char x;
   set_tris_B(0xFF); //PORTB is input.
   set_tris_D(0x00); //PORTD is output.
   output_D(0x00); //Clear PORTD
   while(1)
   {
      x = input_B(); //Read PORTB
      output_D(x);
   }
}
```

Note: delay function could be appended to the above code with appropriate time delay.

11.3.4 Modifications:

1. Rewrite down the **while**(1)'s body using one line with full optimization i.e. less variables, calculations, instructions….
```
output_D( input_B() );          //No needs for variables.
```
2. As you note the above code deals with ports atomically (as a whole), either output or input in **byte mode**, write down the equivalent code of **while**(1)'s body using **bit mode** only.
```
//We can read pin status using input(pin) function, it returns 1
//for high voltage and 0 for low voltage.
output_bit(pin_D0, input(pin_B0));
output_bit(pin_D1, input(pin_B1));
output_bit(pin_D2, input(pin_B2));
    .    //repeated sequentially …..
    .    //until reaches:
output_bit(pin_D7, input(pin_B7));
```

11.3.5 Interfacing 7-segment display:

Before programmable **LCDs** (**L**iquid **C**rystal **D**isplay), the dominant display device for any embedded system was **7-segment display**, and till now you can see it in many systems such that prayer clocks in mosques, customers counter in restaurants, Microwaves timers, fridges temperature viewer, any MPUs or MCUs kit and many other devices. 7-segment can be manufactured in many ways but the most popular is LED approach. Engineers and developers prefer LED 7-segment for many reasons as:

- Low cost.
- Low power consumption.
- Illuminating device.

7-segment display is simply 7 LEDs arranged somehow to demonstrate any BCD or even Hexadecimal number, some of them comes with dot operator called **d**ecimal **p**oint (**DP**), and some are manufactured for a specific system as clock organization. Figure 11.11 shows the general layout for any 7-segment display, from this figure each letter (from **a** to **g**) demonstrates a LED so there are 7 main LEDs or segments; indeed if we ignore the decimal point.

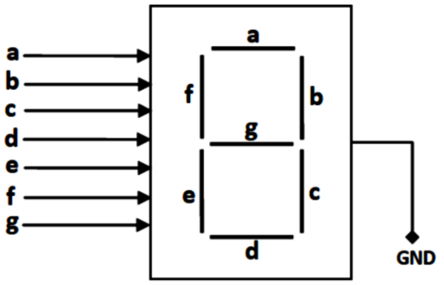


Figure 11.4: A common cathode 7-Segment display layout.

7-segment display comes in two main parts; common cathode or common anode, common cathode as shown in above figure means that all LEDs are common in **GND** so to illuminate any segment (LED) we must feed it with **5 volt**, in contrast of common anode, all segments common in $\mathbf{V_{CC}}$; so for illumination any segment 0 volt must be fed to this segment.

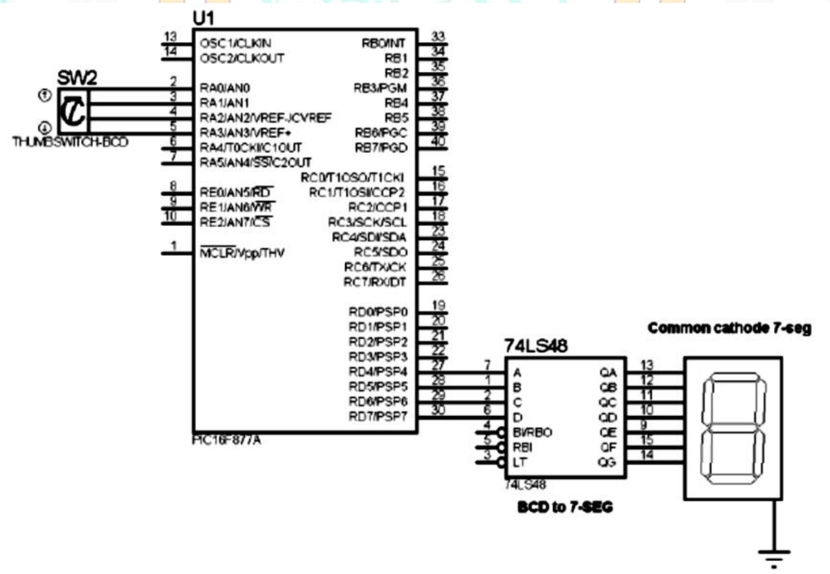
Now suppose that we need to display number 0 on a common cathode 7-seg then we must feed segments **a**, **b**, **c**, **d**, **e** and **f** with V_{CC} and segment **g** with GND. A second example, if we want to display number 2 then **a**, **b**, **g**, **e** and **d** must be in a high voltage, **c** and f must be in low voltage.

In general, the main task for 7-seg is to display any hexadecimal number, and as we know hexadecimal digits is 4 bit long so we must find a method to convert 4 bit digit to 7-seg code. Actually, there are two solutions for this problem, the first by using a dedicated IC or by building your own circuitry I prefer to call this a **hardware solution (BCD to 7-SEG) decoder**. The second is by using a piece of code that stored in the same PIC to do this job i.e. Convert from 4 bit digit to 7-seg code, I called this method a **software solution (look-up table)**.

Also along with 7-segment display (output device); I will introduce an input device named thumbwheel. is a BCD, octal or hexadecimal input device, it is somewhat user friendly, so user **Thumbwheel** can scroll down or up a wheel until reach the target number and the number's code will be generated automatically and latched into output pins. Thumbwheels are mainly found in **PLCs** (**P**rogrammable **L**ogic **A**rray).

Example Three: BCD to 7-segment.

Description: Read a BCD thumbwheel value that connected on <RA0:RA3> and display it on 7-segment connected on <RD4:RD7>.



Schematic is shown in above figure, code is shown below.

```
//Code begins here:
#include <16F877A.h>
#FUSES XT, NOWDT
#USE delay (clock = 4000000)
void main()
{
   //Initialize SFRs (Special Function Registers)
   set_tris_A(0x0F);    // Port A is input for A0-A3
   set_tris_D(0x0F);    // Port D is output for D4-D7
   output_d(0x00);      //Clear PORTD
   while(1)
   {
      output_bit(pin_d4, input(pin_A0));
```

```
        output_bit(pin_d5, input(pin_A1));
        output_bit(pin_d6, input(pin_A2));
        output_bit(pin_d7, input(pin_A3));
    }
}
```

<u>Note:</u> delay function could be appended to the above code with appropriate time delay.

As you expect from example description <RA0:RA3> must be input and <RD4:RD7> must be output, so because of partitioning ports to input and output i.e. <RD0:RD3>, RA4 and RA5 are input pins, we shouldn't write to port D as a whole, and if we reads port A we must ignore excessive bits or read it in using bit mode functions. I used bit mode function especially for port D because it is divided into input and output so if I used byte mode function say **output_D(BCDnum)** maybe I will fall in a trouble with the devices that connected to other pins in port D; if they are exist.

NOWDT is a new fuse introduced in the previous code, **WDT** is stand for **W**atch **D**og **T**imer, it is an internal timer if enabled it begins running with PIC's program until reach some defined value then a software RESET signal will be generated forcing PIC to reset. In other words the watchdog timer is designed to automatically reset the MCU on program malfunctions, by stopping or getting stuck in loop. For example suppose that the value of WDT time-out is 18m second (Typical period) and in the worst case the program needs 10m second to complete one turn of execution then if the execution time takes more than this value means that the program is getting stuck or freeze, and as we mention after 18ms.

WDT time-outs and the MCU forced to reset. In fuses line **NOWDT** means it is disabled, but writing **WDT** only will enable it (by default it's enabled). Finally, if WDT is enabled then it should be regularly reset at the beginning of **while(1)** loop; in **PIC C restart_wdt()** must be called.

The above example stand on hardware solution so **74LS48** IC is used to convert BCD code to 7-segment, you can simply build your own combinational circuitry using simple digital logic design. For more details of **74LS48** IC refer to its datasheet.

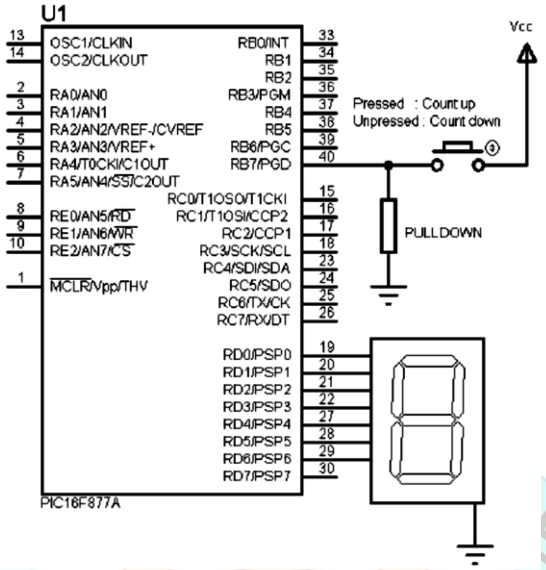
Software solution will be handled in the next example.

Example Four: Hexadecimal Up-Down counter.

Main goals: Interface 7-segment with PIC directly using software solution, more PIC C topics and programming practices.

Description: Read the state of RB7 pin, if it's high then begin counting up on a 7-segment that connected directly to portD, else count down. Counts must be in hexadecimal.

Schematic is shown below.



As you expect from example description port **D** is output and pin **RB7** is input. I think that the program is straightforward.

Referring to figure in page 98, if we imagine that each segment corresponds to one bit, so segment **a** is the least significant bit and **g** is the most, then to display number 0; <**a,b,c,d,e,f**> must be one and <**g**> th must be 0 this means **0111111** in binary and if we suppose that the 8 bit is don't care (set to 0) then this binary numbers corresponds to **0x3F** in hexadecimal; in the code it is considered the first element in an array. In the same manner we can find any digit or symbol and include it into the same array, then we can consider this array as a lookup table and map our target digit to it, for example the code of **number 0** is stored in **location 0**, **number 1** in **location 1** and so on.……

Note that the variable **i** of the **for()** loops is signed integer, the reason behind that is the condition of the first **for()** loop (**i** >= 0), this condition remains true until **i** becomes a negative number, if we use an unsigned integer then this condition will remain true forever i.e. **i** is always positive between **0** and **255**.

Code is shown below.

```
//Code begins here:
#include <16F877A.h>
#FUSES XT, NOWDT
#USE DELAY (CLOCK = 4000000)
char code7seg[16] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,
0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71};
void main()
{
    signed int i;
    set_tris_d(0);
    set_tris_b(0x80); //RB7 is input.
    output_d(0); //Clear portD
    while(1)
    {
        switch(input(pin_B7))
        {
            case 0:
                for(i = 0x0F; i >= 0; i--){
                    output_d( code7seg[i] );
```

```
                delay_ms(500);
            }
            break;
        case 1:
            for(i = 0; i<= 0x0F; i++){
                output_d( code7seg[i] );
                delay_ms(500);
            }
            break;
    }//end switch
  }
}//end main
//Code ends here.
```

H.W2:

Q1)Reprogram example four to count up or down decimal numbers (0 to 9)?
hint: use look-up table.

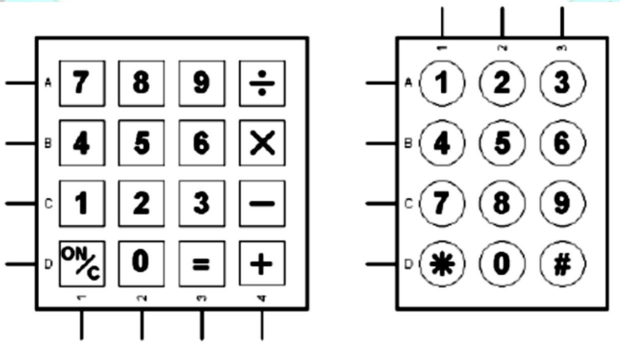
Q2)Redesign example three without using BCD thumbwheel to count up decimal numbers (0 to 9) with 0.5sec delay between each count ?
hint: use 74LS48 IC.

11.4 Digital I/O (Part 2):

In previous part we take simple digital I/O programs with very simple peripherals like LEDs, switches, 7-segments display. Now we will take the same digital I/O functions, but applying them to another complicated peripherals (Devices). Two devices will be explained in this part: **keypad** and **LCD** also we will show how to utilize from internal **EEPROM** memory.

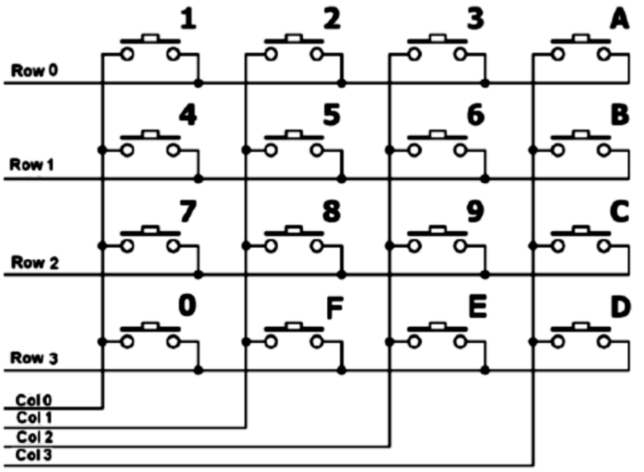
11.4.1 Keypad:

Keypad can be considered as a small keyboard, it comes with many embedded systems as a standard input device such as Phones either telephones or cell phones, Calculators, Microwaves, Security systems, Remote control modules and many more…., figure below shows a 4x3 and a 4x4 keypads.



4x4 Keypad 4x3 Keypad

Push button is the main component for any keypad, 16 push buttons for 4x4, 12 for 4x3 keypads and so on. These push buttons are connected using matrix approach, so we can consider any keypad as a two dimensional array. For example, a 4x3 keypad has 4 rows and 3 columns, in figure above rows are named A, B, C, D and columns 1, 2, 3. Matrix connection improves the connectivity and reduces the number of used pins, i.e. if we connect 16 buttons to MCU directly then we need 16 input pins and this implies more wiring, in the other hand if we connect them as a matrix then only 8 pins are needed with less wiring. Figure below shows the internal connection of a 4x4 keypad using matrix approach.



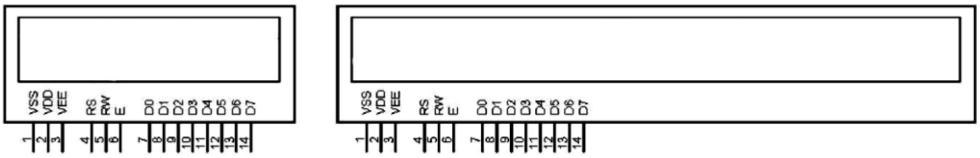
A scanning method (will be explained using comments in code) is used to get any pressed key, also either rows or columns (according to scanning method) must pull-ups.

11.4.2 LCD:

LCD (**L**iquid **C**rystal **D**isplay) is considered the dominant of display devices in embedded system now. It comes with a variety flavors such as 7-segment, Textual, Graphical and Dot matrix. LCDs exist in many commercial systems as a standard output device like Cell phones, Laptops, Digital multi-meters, Digital cameras…. and in general; most of MCUs applications.
Here we concerned in Textual LCD with integrated **HD44780** controller only, note that most of commercial LCDs are based on this controller. The purpose of **HD44780** is making an interface between LCD and any MCU for both hardware and software.
Table below shows different models of LCDs that use a built in HD44780.

Model name	Rows / Characters
LM016L	2 rows × 16 characters per row
LM017L	2 rows × 20 characters per row
LM018L	2 rows × 40 characters per row
LM044L	4 rows × 20 characters per row

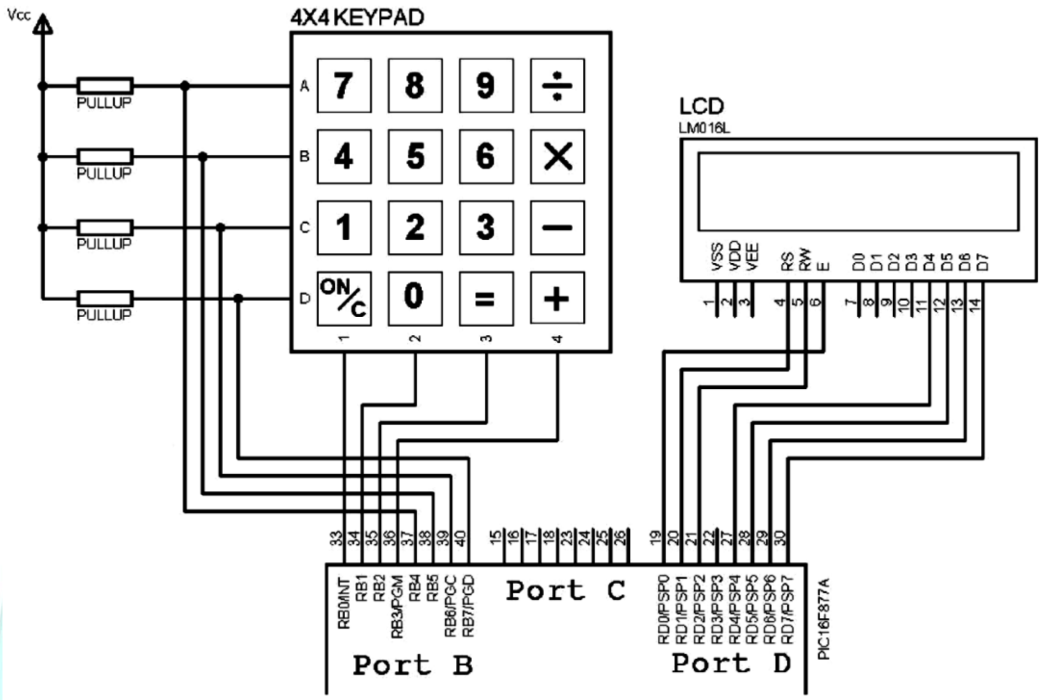
Figure below shows **LM016L** and **LM018L** LCDs layouts.



LM016L LM018L

Fortunately, PIC C has a built in libraries for LCDs based on HD44780 controller. The following example shows how to connect and program PIC16F877A with 4x4 keypad and LM016L.

Example name: Echo program.
Main goal: How to deal with keypads and LCDs.
Description: waits for a key, when pressed print it out to LCD.
Circuit connections are shown below, note that connections between LCD and PIC are implemented according to LCD's library (**...\PICC\Drivers\LCD.c**). Code is shown in the next page.




```
#include <16F877A.h>
#FUSES XT, NOWDT
#USE DELAY (CLOCK = 4000000)
#include <LCD.c> //LM016L library
/* LCD connections
    D0 -> E, D1 -> RS, D2 -> RW
    D4 -> D4, D5 -> D5, D6 -> D6, D7 -> D7
*/
//Keypad connection:
#define col0 PIN_B0
#define col1 PIN_B1
#define col2 PIN_B2
#define col3 PIN_B3
#define row0 PIN_B4
#define row1 PIN_B5
#define row2 PIN_B6
#define row3 PIN_B7
char getKey()
{
    //Columns are output, Rows are input.
    //rows are pulled-up, and PIC always reads them high unless a key is pressed.
    do
    {
        output_low(col0);output_high(col1);output_high(col2);output_high(col3);
        if(!input(row0)){
            while(!input(row0)); //wait until the pressed key is released.
```

```
            return '7'; //Key[0][0] in our keypad
        }
        if(!input(row1)){ while(!input(row1)); return '4';} //key[1][0]
        if(!input(row2)){ while(!input(row2)); return '1';} //key[2][0]
        if(!input(row3)){ while(!input(row3)); return 'c';} //key[3][0]
        output_high(col0);output_low(col1);output_high(col2);output_high(col3);
        if(!input(row0)){ while(!input(row0)); return '8';}
        if(!input(row1)){ while(!input(row1)); return '5';}
        if(!input(row2)){ while(!input(row2)); return '2';}
        if(!input(row3)){ while(!input(row3)); return '0';}
        output_high(col0);output_high(col1);output_low(col2);output_high(col3);
        if(!input(row0)){ while(!input(row0)); return '9';}
        if(!input(row1)){ while(!input(row1)); return '6';}
        if(!input(row2)){ while(!input(row2)); return '3';}
        if(!input(row3)){ while(!input(row3)); return '=';}
        output_high(col0);output_high(col1);output_high(col2);output_low(col3);
        if(!input(row0)){ while(!input(row0)); return '/';}
        if(!input(row1)){ while(!input(row1)); return '*';}
        if(!input(row2)){ while(!input(row2)); return '-';}
        if(!input(row3)){ while(!input(row3)); return '+';}
    }while(1);
}
```

The above piece of code is concerned with including LCD's library and implementing a function **getkey()** (more professional 4x4 keypad functions found in **www.ccsinfo.com/forum**) to read a key from keypad when pressed. As you see this function return the ASCII code for the key, you can modify the returned value to any type (e.g. int, BYTE...) so every returned value has its meaning. Also you can change the port or pins that used by keypad, easily by changing each pin located in **define** lines.

Main function (**main()**) is shown below.

```
void main()
{
    char key;
    set_tris_b(0xF0); //configre keypad Columns <RB0:RB3>, Rows <RB4:RB7>
    output_b(0xF0);
    lcd_init(); //Initilize LCD.
    lcd_gotoxy(1,1); //Set cursor.
    lcd_putc("... Welcome ...");
    delay_ms(1500);
    lcd_putc('\f');
    while(1)
    {
        key = getkey();
        delay_ms(50);
        if(key != 'c')
        printf(lcd_putc, "%c", key);
        else lcd_putc('\f');
    }
}
```

The code is straightforward and easy to understand.
LCD's functions are summarized in table U.

Function syntax	Description And examples
Lcd_init()	Used to initialize LCD, you have to call it before use any other function.
Lcd_gotoxy(x,y)	Set the cursor to a specific location for writing. (1,1) is the first location.
Lcd_putc(data)	Display data on LCD, data must be string (array of characters) or one character (in ASCII).
Lcd_getc(x,y)	Returns character from a specific location.

Printf(lcd_putc,String,var1,var2,…)	Used to display any variable or constant. For example:
	printf(lcd putc,"z = %u",var1);//output: z = |var1|
	Printf(lcd putc,"Temperature = %d C",t);
	Printf(lcd_putc,"%d+%d = %d",x,y,r);//out: |x| + |y| = |r|

Special meanings:	**\n** : new line. **\f** : clear LCD. **\b** : move back one location. **%d** : signed int. **%u** : unsigned int. **%f** : float. **%c**: char.

PIC16F877A has internal pull-ups resistors on <RB4:RB7>. We can utilize this feature when using a keypad in our design. To activate it just write the following line: **port_b_pullup(true)** under **set_tris_b(0xF0)**. Also you must modify your circuit by removing all pull-ups resistors on <RB4:RB7>.

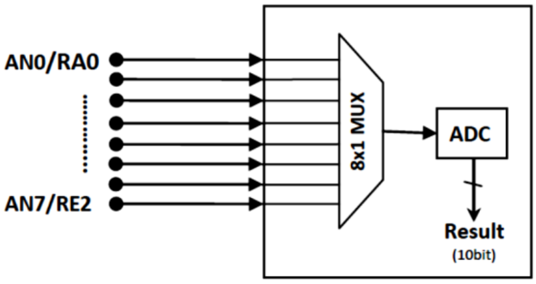
11.4.3 Internal EEPROM:

As we say in section 2, PIC16F877A has an internal 256B EEPROM. This memory is used to store any permanent data (e.g. system settings, passwords…) that must remain when the system is reset or even when shut down, also this data can be modified at run time.
PIC C gives us two main functions for writing on or reading from internal EEPROM:

1. **write_eeprom(address, data)**//writes data(one byte) to a specified address(one byte).
2. **read_eeprom(address)**//returns one data byte from a specified address(one byte).

11.5 ADC Module:

ADC is stands for **A**nalog to **D**igital **C**onverter. It is used to convert any analog signal to digital data so that it can be stored and manipulated digitally. Digital voltmeter is a good example it simply takes analog reading (Voltage) and converts it to digital using ADC then by making simple calculations on this digitized value we will have a digital reading corresponds to the original analog, now we can store it, display it on a 7-segment or LCD, send it to PC as we will see later and the most important we can modify and process this digital data. And same as digital voltmeter procedure we can handle any analog input such as temperature, pressure and so on…
PIC16F877A has 8×10bit multiplexed ADC channels (AN0-AN7) mapped to port **E** and port **A** except RA4. Next figure shows an abstraction view for ADC module.



As you note from previous figure the result of conversion is 10-bit width and this means that the result of conversion is a value between 0 and 2^{10}-1 or [0,1023]. For example if we use **5volt** as reference voltage then analog input should be in TTL level (ranges from 0 and 5volt) and in this case **0volt** analog corresponds to **0** digital in result and **5volt** analog corresponds to **1023** digital in result. In real application analog input is unknown and at the same time it is our target. The procedure for calculating this value is straightforward. Firstly, make a conversion to get a digital value that corresponds to analog input. Secondly, make a reverse calculation for the unknown analog voltage as follows (**note that** the default value for $\mathbf{V_{ref}}$ is $\mathbf{V_{DD}}$ voltage at pin 11 or 32, in general it is **5v**):

$$V_{ref} \rightarrow 1023$$
$$Analog_{unknown} \rightarrow Digital_{Known}$$

From above expression: $Analog_{unknown} = \frac{V_{ref}}{1023} \times Digital_{Known}$

NOTE: PIC C provides us with two choices for ADC manipulation either 8-bit (ADC=8) or 10-bit (ADC=10). Above equation uses 10-bit, if we use 8-bit then we must divide by 255 (2^8-1) rather than 1023 (2^{10}-1).

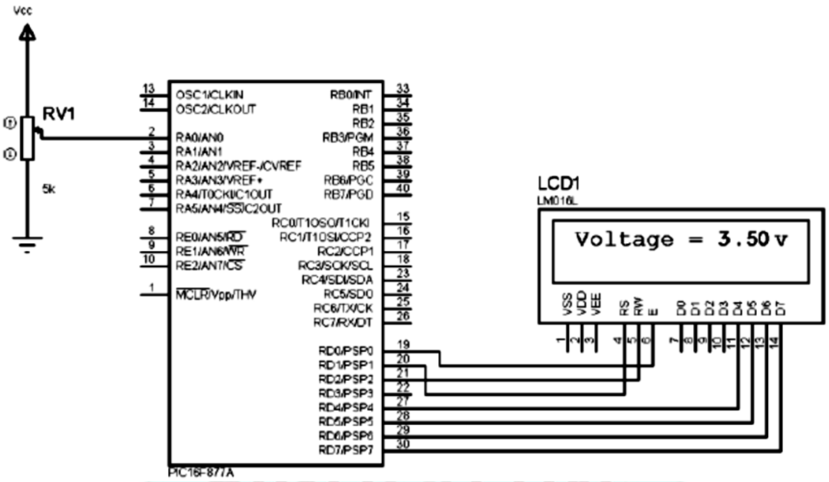
To use ADC module the following steps must be applied:

Steps:	Corresponding PIC C code (Examples):
1. Configure pins (analog or digital).	setup_adc_ports(ALL_ANALOG);
2. Configure conversion clock.	setup_adc(ADC_CLOCK_INTERNAL);
3. Select channel.	set_adc_channel(0);
4. Read conversion result.	DigX = read_adc();//now multiply DigX with Vref/1023 or Vref/255.

Once you configure the ADC (first 3 steps) you can read the converted value.
The first 3 steps can be written one time in the main() and before while(1).
You can pass other parameters for the PIC C code shown above, we will take some of them using examples.

Two examples will be taken: first, building a digital voltmeter. Second, measuring temperature and displaying it on LCD.

Example 1: TTL digital voltmeter.
Main goal: How to deal with ADC module (10bit resolution).
Description: reads a voltage ranges from 0 to 5 volt and display it on LCD.



As shown above a variable resistor is used to choose a voltage from 0 to 5v. Code is shown below.

```
#include <16F877A.h>
#DEVICE ADC=10 // return 10 bit(FULL RESOLUTION) from ADC, Will be explained later.
#FUSES XT // Crystal osc <= 4mhz.
#FUSES NOWDT // No Watch Dog Timer.
#FUSES NOPROTECT // Code not protected from reading.
#USE DELAY(CLOCK = 4000000)
#include <lcd.c>
void main()
{
    int16 digitalValue; //16 bit integer, to store the ADC result (10bit).
    float voltage;
    setup_adc_ports( ALL_ANALOG ); //All 8 pins are analog input. Vref is 5v.
    setup_adc( ADC_CLOCK_INTERNAL );//Use internal clock (TAD between 2u-6u Second)
    set_adc_channel( 0 ); //AN0 is the used here pin.
    lcd_init();
    lcd_gotoxy(1,1);
    printf(lcd_putc, "Digital voltmeter");
    delay_ms(1500);;
    while(TRUE)
    {
        digitalValue = read_adc();
        voltage = (float)digitalValue/204; //ADC equation: digitalValue*5/1023
        printf(lcd_putc,"\fVoltage = %1.2f v", voltage);//(%1.2f): display a float with 1
        //integer digit and 2 fractional.
        delay_ms(500); //can be ignored
    }
}
```

Because ADC ports are configured to **ALL_ANALOG** we can use any pin from <AN0:AN7> as analog input, for other parameters show 16F877A.h header file. In above example we use **AN0** and according to that we must read from channel 0 this is done using **set_adc_channel(0)** line. Suppose that **AN5** is used then instead of passing **0** we must pass **5** i.e.) **set_adc_channel(5)**. T_{AD} is the time required from ADC to digitize one bit (at minimum, it must be 1.6uS). By choosing **ADC_CLOCK_INTERNAL** T_{AD} automatically set between 2uS and 6uS.

Example 2: Temperature control system.

Main goals: More about ADC module, dealing with LM35 temperature sensor.

Description: Reads a temperature from $0^{O}C$ and above. If it is less than $22^{O}C$ send a high signal from RE0 to operate a heater. If the temperature is more than $27^{O}C$ send a high signal from RE1 to operate an air conditioner. Use ADC=8 and V_{ref} = 1v.

We will use LM35DZ temperature sensor, it is a 3 terminal sensor (V_{CC}, GND and O/P) and it can measure a wide temperature range (from $0^{O}C$ up to $100^{O}C$). The most important feature (for more features refer to datasheet) that its output is linear with $10mV/^{O}C$. This means that if temperature is $1^{O}C$ then LM35's output is 10mV, so and simply:

$$\frac{1^{O}C}{10mV} = \frac{Temperature}{Sensor\ output\ voltage}$$

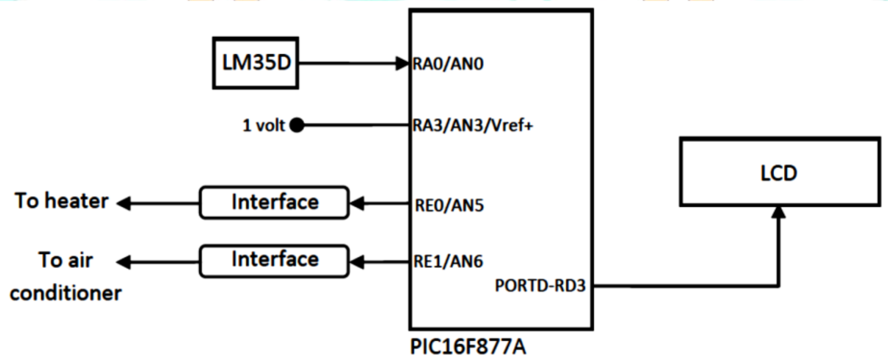
And this implies that:

$$Temperature = \frac{Sensor\ output\ voltage}{10mV} = (Sensor\ output\ voltage) \times 100$$

For example suppose that the output voltage is equal to 250mV then according to above equations the current temperature is the result of $(250\times10^{-3}\times100)$ and this equal to $25^{O}C$.

Honestly, there are many types to LM35 like LM35A, LM35C and our sensor LM35DZ, each one differ from other in temperature range (e.g. LM25C can measure from $-40^{O}C$ to $110^{O}C$) and accuracy.

As you note from description RE0 and RE1 must set to digital output, LM35 is connected to AN0 and the reference voltage (pin A3) is set to 1v. Schematic is shown below.



Tip 1: Control system like that is called a **regulator system**. It is automatically maintains a parameter at (or near) a specified value; in our example we maintain temperature.

The interface between PIC (low voltage devices) and heater or air conditioner (high voltage devices) can be done using any device that makes isolation between them like relays (certainly with other elements).

Code is shown below.

```
#include <16F877A.h>
#DEVICE ADC=8 //return 8-bit width. Don't forget to divide digitalValue over 255.
#FUSES XT, NOWDT, NOPROTECT
#USE DELAY(CLOCK = 4000000)
#include <LCD.c>
#DEFINE heater PIN_E0
#DEFINE air_c PIN_E1
void main()
{
   int8 digitlValue; //Store the result of A/D conversion. 8bit is enough.
```

```
    float temperature;
    set_tris_d(0);
    output_d(0);
    //Initialize ADC module
    setup_adc_ports(AN0_AN1_VSS_VREF); //AN0 and AN1 are analog input pins.Vref at AN3.
    setup_adc(ADC_CLOCK_INTERNAL);
    set_adc_channel(0);
    lcd_init();
    lcd_gotoxy(1,1);
    lcd_putc("Temperature\nControl System");
    delay_ms(1500);
    while(1)
    {
        digitlValue = read_adc();
        temperature = (float)digitlValue / 255; //Apply ADC equ.: digitalValue*Vref/255
        Temperature = temperature * 100; //Apply LM35 equation.
        printf(lcd_putc, "\fT = %2.2f", temperature);
        if(temperature > 27.0)
        {
            printf(lcd_putc, "\nHigh temperature!");
            output_high(air_c); //turn ON air conditioner.
        }
        else if(temperature < 22.0)
        {
            printf(lcd_putc, "\nLow temperature!");
            output_high(heater); //turn ON heater.
        }
        else
        {
            printf(lcd_putc, "\nModerate T..re!");
            output_low(air_c); //turn OFF air conditioner.
            output_low(heater); //turn OFF heater.
        }
        delay_ms(500);
    }
}//end main
```

In **(#DEVICE)** directive we use **ADC=8** instead of **ADC=10** this means that **read_adc()** function will return 8-bit only from the converted result and a variable with int8 type is enough to store this value also instead of dividing by 1023(2^{10}-1) in ADC equation **((V_{ref}×digitalValue)/1023)** we must divide by 255 (2^8-1), the overall result is a light calculation but less accuracy.

Pin **RA3**/**AN3** can be used as analog input or reference voltage input. This can be determine according to the argument that passed to **setup_adc_ports()** function. In the above code it is used as V_{ref} . By setting V_{ref} to 1v the final result will be somewhat more accurate (in examples like this only). To show the difference you can convert V_{ref} to default VDD (in general 5v) as example 1 and change the ADC equation to **((digitalValue×5)/255)**. Next table shows some **setup_adc_ports()** parameters.

Some **setup_adc_ports()** parameters.

Parameter	Description
NO_ANALOG	All pins are digital.
ALL_ANALOG	All pins are analog. $V_{ref} = V_{DD}$.
AN0	AN0 is the only analog pin. $V_{ref} = V_{DD}$.
AN0_AN1_AN3	All of these pins are analog input. $V_{ref} = V_{DD}$.
AN0_AN1_AN2_AN4_VSS_VREF	All of these pins are analog. V_{ref} is set at AN3.

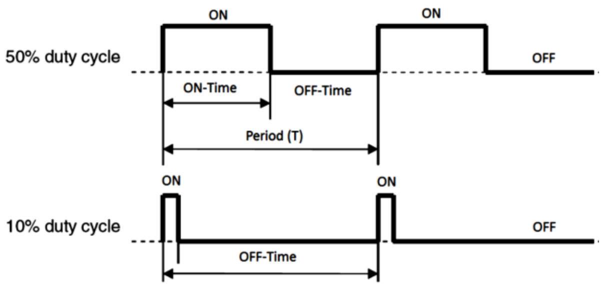
NOTE: V_{DD} or V_{CC} means the voltage that fed to PIC at pin 11 or 32; it is normally 5v but you can choose from 2v – 5.5v (according to PIC specification). So if you use V_{DD} as V_{ref} for example **ALL_ANALOG** or **AN0_AN1_AN3** then you must measure the voltage that supplied to PIC at pin 11 or 32 and change ADC equation according to it.
PIC16F877A hasn't a float point circuitry so all float calculations handled by PIC C using software and this implies to long execution time and more memory usage. In project section we introduced a method called integer coding scheme it can handle any float calculations using simple integer calculations.

11.6 CCP Module:

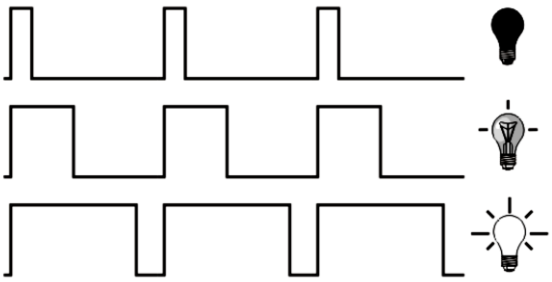
CCP is stands for Capture/Compare/PWM. PIC16F877A has two CCPs named as CCP1 and CCP2. This module can operate in one of three modes capture, compare or PWM. Here we will take PWM only.

11.7 PWM mode:

PWM (**P**ulse **W**idth **M**odulation) is a powerful technique for controlling analog devices like lamps or motors using digital signals! By controlling analog circuits digitally, system costs and power consumption can be greatly reduced.
Simply, PWM is a way of digitally encoding analog signal levels. The **duty cycle** of a square wave is modulated to encode a specific analog signal level. The PWM signal is still digital because, at any given instant of time, the full DC supply is either fully on or fully off. The voltage or current source is supplied to the analog load by means of a repeating series of on and off pulses. The **ON-Time** is the time duration which the DC supply is applied to the load, and the **OFF-Time** is the period duration which that the DC supply is switched off.
Figure below shows two different PWM signals. One signal shows a PWM output at a 10% duty cycle. That is, the signal is ON for 10% of the period and OFF the other 90%. The second signal shows PWM outputs at 50% duty cycle. These PWM outputs encode two different analog signal values, at 10% and 50% of the full strength. If, for example, the supply is 12V and the duty cycle is 10%, a 1.2V analog signal results. In the other hand on 50% duty cycle the result is 6V.

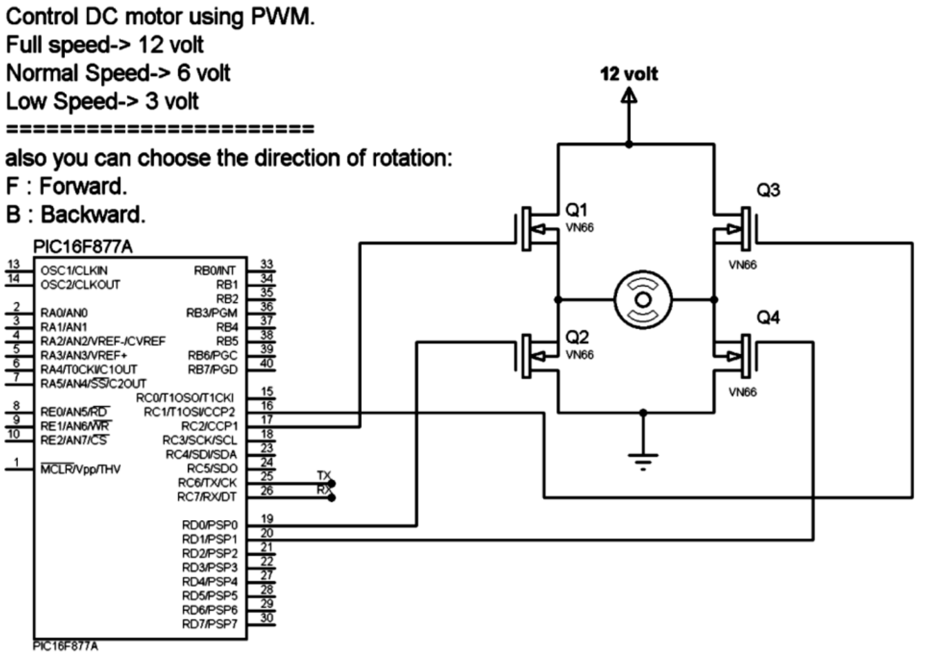


The next figure demonstrates the effect of applying signals with different duty-cycles to a lamp.



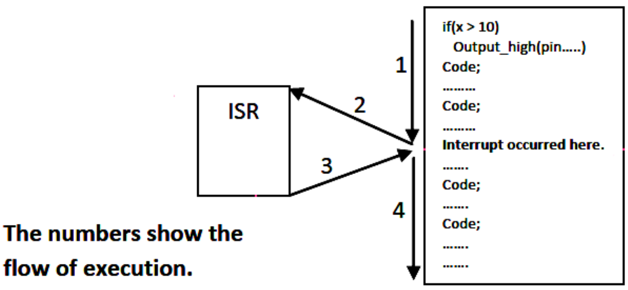
In the next example PWM technique used to control the speed of a DC motor. The motor is interfaced to PIC using H-bridge. PWM **must be used along with timer 2**. The code is full commented and explained.

Note: you can use BD135 NPN BJT (I_C up to 1.5 A) instead of the VN66 FET.



11.8 Interrupts:

Interrupt is a special event forced MPU or MCU to stop their normal execution and jump to a known location that considered the beginning of a block of code called **I**nterrupt **S**ervice **R**outine **(ISR)**. And after executing this ISR they return to their normal execution. Next figure shows that.



A simple interrupts life cycle.

PIC16F877A has up to 15 different types of interrupts. Table below highlights the most important interrupts.

Interrupt name (PIC C naming)	Description
EXT	External edge triggered interrupt on RB0/INT.
RB	Any changes on pins <RB7:RB4> from PORTB. Pin must be input.
RDA	USART received data. A very useful interrupt.
TIMERx	Timer x overflow, where x is equal to 0, 1 or 2.

Before using any interrupt we must enable it and also you must enable a global interrupt bit called **GLOBAL** as we will see later. The following functions are used to enable or disable interrupts:

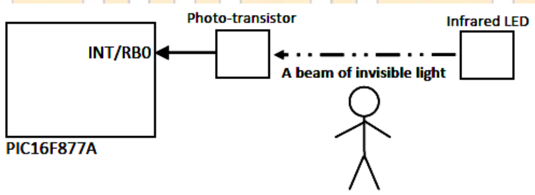
enable_interrupts(INT_name);
disable_interrupts(INT_name);

Where INT_name is the interrupt name like INT_EXT, INT_RB, INT_TIMER1 and so on…, except GLOBAL it is used as is. External interrupt (INT_EXT) has a special feature that its edge is programmed; this means that we can control the input edge on pin **INT/RB0** to be either negative or positive.

Example 1: Visitors counter.
Main goal: Dealing with interrupts.
Description: Suppose that you are to design a control system for a library. This control system is concerned with temperature regulation, cameras, lighting and so on…. And the library manager requests from you to count the number of visitors that get in the library. Using the external interrupt, design visitors counter part.

It is clear that cameras need continues controlling (and somehow temperature and lighting) this situation called event driven. In contrast visitors counter must be implemented using interrupt driven approach i.e.) each time a visitor enter the main entrance you must increment a counter by 1.

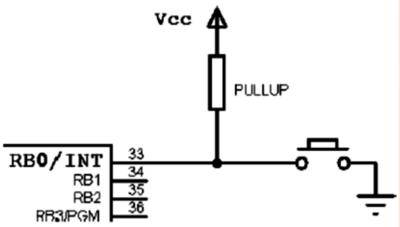
The schematic is shown in the next page; I use a photo-transistor (or photo-diode) with an infrared LED (IR transmitter). Infrared LED is always emits a beam to the photo-transistor so that the default state on pin **INT/RB0** is high, and will get to low as long as a person is in the way of it and this implies to generate **INT_EXT** interrupt. Code is shown below.




```
//Code begins here
#include <16F877A.h>
#FUSES XT,NOWDT
#USE DELAY (CLOCK = 4000000)
#include <LCD.c>
int16 VisitorsCounter;
#INT_EXT //you must use INT_name as a directive before ISR().
void Vcounter() //ISR name
{
   VisitorsCounter+=1; //increment counter.
   lcd_putc("\f# of visitors is\n");
   printf(lcd_putc,"%lu till now",VisitorsCounter); //Display it.
}
```

```
void main()
{
    enable_interrupts(INT_EXT);//enable external interrupt.
    enable_interrupts(GLOBAL); //enable global interrupt bit.
    ext_int_edge(H_TO_L); //State the edge of interruption.
                                //(H_to_L) for negative edge and (L_TO_H) for posiyive edge.
    lcd_init();
    lcd_gotoxy(1,1);
    VisitorsCounter = 0; //Clear counter.
    while(1)
    {
        //you can write here any event driven actions.
        //as temperature or lighting calculation.
        //controlling cameras.
    }
}
//Code ends here
```

Note that **ext_int_edge()** is a special function concerned with external interrupt only. To simulate the response of photo-transistor and infrared LED using proteus or any other simulation program you can use a push button connected to pin **INT/RB0** as shown below.



Important Tips:

- You must write a directive (#) INT_name before each ISR() as shown in the code.
- You can choose any name for ISR().
- Don't forget to enable the global and the target interrupts.
- You can enable as many as interrupts you need.