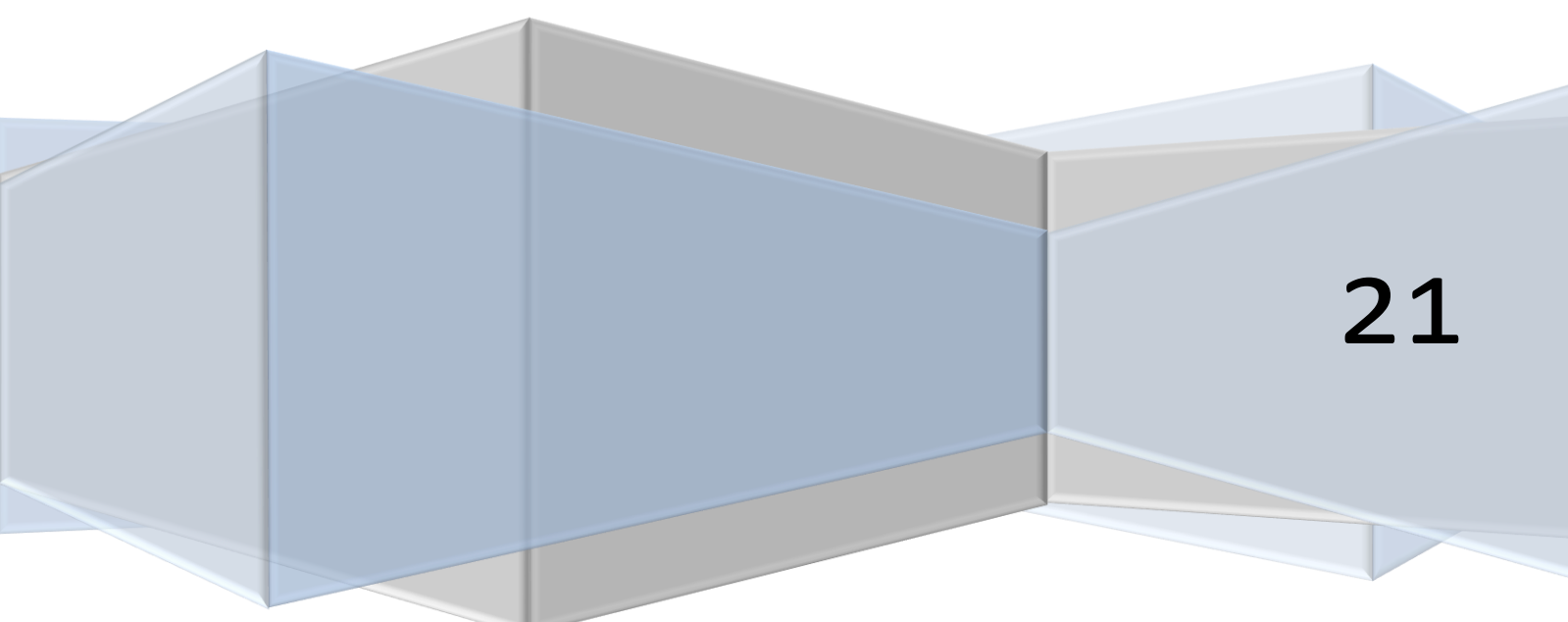


Work Book-PIC16f877A

www.pantechsolutions.net



21

Contents

PIC16F877A.....	1
PIC16F877A GPIO Tutorial	6
PIC16F877A – DC Motor Interfacing.....	15
PIC16F877A – Relay Interfacing	18
PIC16F877A – LCD 4Bit Interfacing	21
PIC16F877A – Keypad Interfacing.....	27
PIC16F877A – Timer/Counter	33
PIC16F877A – USART	42
PIC16F877A – ADC	49
PIC16F877A- I2C.....	56
PIC16F877A – Interrupt Tutorial	66
Serial Interrupt – PIC16F877A Interrupt Tutorial.....	73

PIC16F877A

Introduction

This is the Series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In this article, we will see the PIC16F877A Introduction and features.

PIC Introduction

Peripheral Interface Controllers (PIC) is one of the advanced microcontrollers developed by microchip technologies. These microcontrollers are widely used in modern electronics applications. A PIC controller integrates all types of advanced interfacing ports and memory modules. These controllers are more advanced than normal microcontrollers like 8051. The first PIC chip was announced in 1975 (PIC1650). As with a normal microcontroller, the PIC chip also combines a microprocessor unit called CPU and is integrated with various types of memory modules (RAM, ROM, EEPROM, etc.), I/O ports, timers/counters, communication ports, etc.

All PIC microcontroller family uses Harvard architecture. This architecture has the program and data accessed from separate memories so the device has a program memory bus and a data memory bus (more than 8 lines in a normal bus). This improves the bandwidth (data throughput) over traditional von Neumann architecture where program and data are fetched from the same memory (accesses over the same bus). Separating program and data memory further allows instructions to be sized differently than the 8-bit wide data word. Now we will move to PIC16F877A.

PIC16F877A Introduction

Microcontroller PIC16F877A is one of the PIC micro Family microcontrollers which is popular at this moment, start from beginner until all professionals. Because very easy to use PIC16F877A and use FLASH memory technology so that can be write-erase until thousand times. The superiority this Risc Microcontroller compared to with another microcontroller 8-bit especially at a speed of and his code compression.

He 16F877A is a capable microcontroller that can do many tasks because it has a large enough programming memory (large in terms of sensor and control projects) of 8k words and 368 Bytes of RAM. This is enough to do many different projects.

Note: There is a more modern part (the 16F887) that has nearly the same functionality as the 16F877A but also includes an internal clock – like the 16F88 and the 18F4550. In addition, the 16F887 also has low power operation using Nano

watt™ technology.

Differences Between 16F877A and 16F887

Interface	16F877A	16F887	Description
RA4/T0CKI	Open drain	Normal CMOS	The pin is physically different so the input characteristics are changed.
Cost	Expensive	Cheaper	Modern devices are usually cheaper.
ADC	Yes	More useful	More controls although different registers are used.
Nano Watt™	No	Yes	16F88x - ultra low power operation (battery operation).
Internal Clock	No	Yes	8Mhz to 31kHz 1% accuracy.
External Gate	No	Yes	External Timer1 gate input (start Timer1 counter).
Volt reference	No	Yes	Internal 0.6V voltage reference.
RS485, LIN 2.0	No	Yes	Enhanced USART supports RS485 and LIN 2.0 operation.
Parallel Slave port	Yes	No	Acts as an 8-bit processor interface i.e. another 8 bit processor can read and write to this interface controlling the 16F877A as a slave processor.

The four features that might make you use a 16F887 instead of a 16F877 (A) are External gate.

Volt Reference.

Nano Watt™.

Internal Clock.

The gate could be used to more accurately capture an input time e.g. for a reciprocal frequency counter.

The volt reference means you don't need an external reference although it will probably not be useful for highly accurate operation. It is definitely more useful in a battery-powered operation where you want to compare the input battery voltage to a known reference e.g. using the comparator and the internal 0.6V reference.

Nano Watt™ could be useful for battery-powered operations.

The internal clock is useful for lab development (not for accuracy) and for general operation – it can also be set to 31kHz so consuming less power.

All the above depend on your specific application requirements.

PIC16F877A has 40 pins by 33 paths of I/O. The 40 pins make it easier to use the peripherals as the functions are spread out over the pins. This makes it easier to

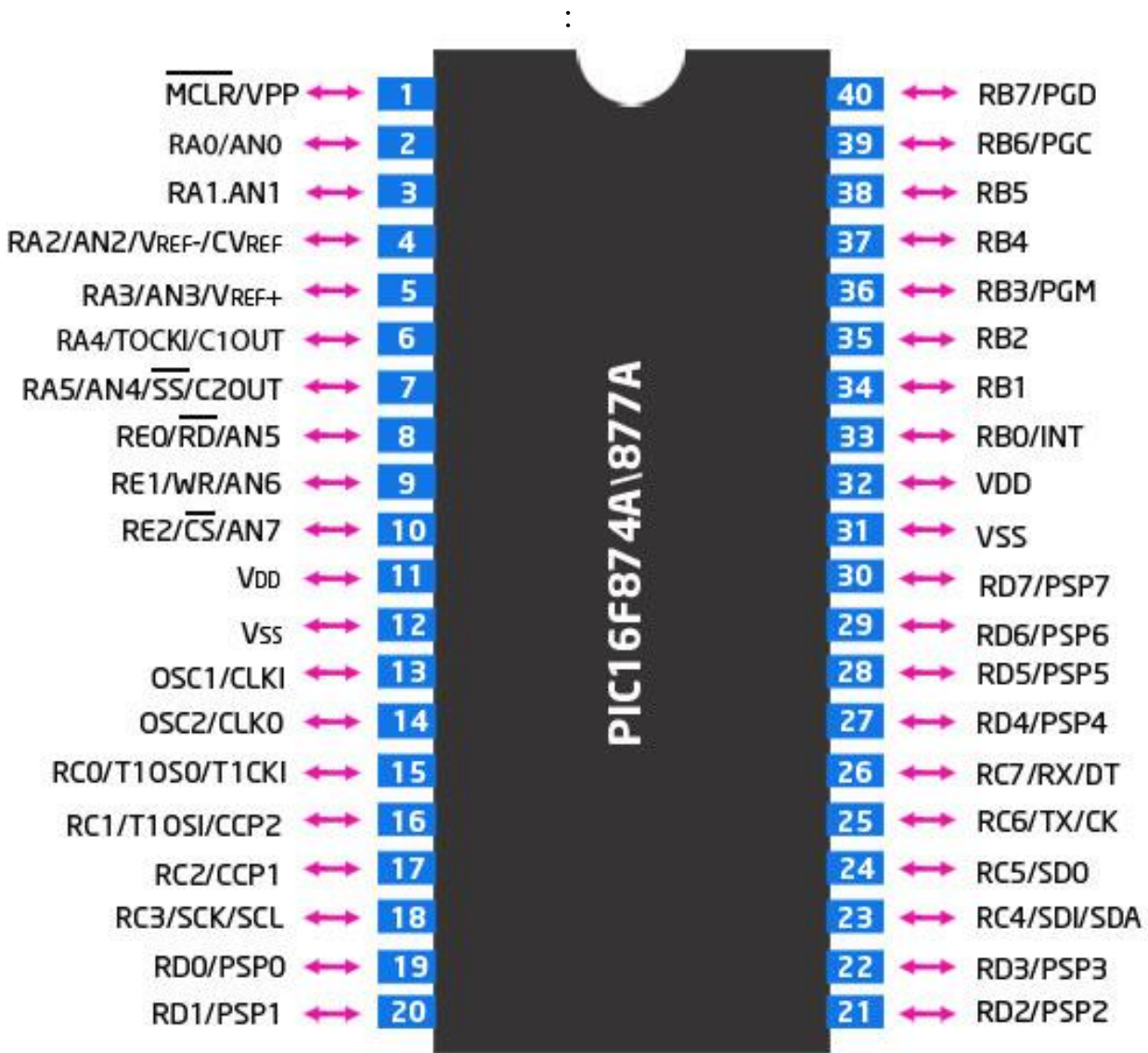
decide what external devices to attach without worrying too much if there are enough pins to do the job.

One of the main advantages is that each pin is only shared between two or three functions so it's easier to decide what the pin function (other devices have up to 5 functions for a pin).

A slight disadvantage of the device is that it has no internal oscillator so you will need an external crystal or another clock source. However the internal oscillator is only 1% accurate and adding a crystal (max 20MHz crystal – for 5MHz internal instruction cycle) and two 15pF capacitors is not a great chore – the accuracy will be 100ppm depending on the crystal used.

Pin out

The pin out of the 16F877A is:



Features of PIC16F877A (PIC16F877A Introduction)

The PIC16F877A CMOS FLASH-based 8-bit microcontroller is upward compatible with the PIC16C5x, PIC12Cxxx, and PIC16C7x devices. It features 200 ns instruction execution, 256 bytes of EEPROM data memory, self-programming, an ICD, 2 Comparators, 8 channels of 10-bit Analog-to-Digital (A/D) converter, 2 capture/compare/PWM functions, an asynchronous serial port that can be configured as either 3-wire SPI or 2-wire I2C bus, a USART, and a Parallel Slave Port.

High-Performance RISC CPU

Lead-free; RoHS-compliant

Operating speed: 20 MHz, 200 ns instruction cycle

Operating voltage: 4.0-5.5V

Industrial temperature range (-40° to +85°C)

15 Interrupt Sources

35 single-word instructions

All single-cycle instructions except for program branches (two-cycle)

Special Microcontroller Features

Flash Memory: 14.3 Kbytes (8192 words)

Data SRAM: 368 bytes

Data EEPROM: 256 bytes

Self-reprogrammable under software control

In-Circuit Serial Programming via two pins (5V)

Watchdog Timer with on-chip RC oscillator

Programmable code protection

Power-saving Sleep mode

Selectable oscillator options

In-Circuit Debug via two pins

Peripheral Features

33 I/O pins; 5 I/O ports

Timer0: 8-bit timer/counter with 8-bit Prescaler

Timer1: 16-bit timer/counter with Prescaler

Can be incremented during Sleep via external crystal/clock

Timer2: 8-bit timer/counter with 8-bit period register, Prescaler, and postscaler

Two Capture, Compare, PWM modules

16-bit Capture input; max resolution 12.5 ns

16-bit Compare; max resolution 200 ns

10-bit PWM

Synchronous Serial Port with two modes:

SPI Master

I2C Master and Slave

USART/SCI with 9-bit address detection
Parallel Slave Port (PSP)
8 bits wide with external RD, WR, and CS controls
Brown-out detection circuitry for Brown-Out Reset
Analog Features
10-bit, 8-channel A/D Converter
Brown-Out Reset
Analog Comparator module
2 analog comparators
Programmable on-chip voltage reference module
Programmable input multiplexing from device inputs and internal VREF
Comparator outputs are externally accessible

PIC16F877A GPIO Tutorial

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. Now we are going to see PIC16F877A LED Interfacing Tutorial (PIC16F877A GPIO Tutorial). If you want to interface with LED, you should know the Registers used for GPIO. At the end of this tutorial, you will be familiar with the PIC GPIO's and the associated registers for configuring and accessing the GPIO's.

Prerequisites

[PIC16F877A Introduction](#)

PIC16F877A GPIO Tutorial

Introduction

PIC16F877A has 33-GPIO's grouped into five ports namely PORTA to PORTE. They are used for the input/output interfacing with other devices/circuits. Most of these port pins are multiplexed for handling alternate functions for peripheral features on the devices. All ports in a PIC chip are bi-directional. When the peripheral action is enabled in a pin, it may not be used as its general input/output functions.

PORTA

PORTB

PORTC

PORTD

PORTE

Now we will get into GPIO registers.
GPIO Registers

Register	Description
TRISX	This Register is used for Select that respected IO port as a input or output.
PORTX	This is the IO Port

Note: Here 'x' could be A,B,C,D,E so on depending on the number of ports supported by the controller.

TRISX Register

Before accessing the PORTX register, we should declare that port whether input or output. So this register is used to select that direction. If you set 0 that IO port will act as the output port. If you set 1 that IO port will act as an input port. Just see the snippet below. Then you will understand.

TRISB = 0xff; // Configure PORTB as Input.

TRISC = 0x00; // Configure PORTC as Output.

TRISD = 0x0F; // Configure lower nibble of PORTD as Input and higher nibble as Output

TRISD = (1<<0) | (1<<3) | (1<<6); // Configure PD0,PD3,PD6 as Input and others as Output

PORTX Register

This register is used to read/write the data from/to port pins. Writing 1's to PORTx will make the corresponding PORTx pins as HIGH. Similarly writing 0's to PORTx will make the corresponding PORTx pins as LOW. PORTX registers and their alternate functions are shown in the below table.

PORT	Number of Pins	Alternative Function
PORTA	6 (PA0-PA5)	ADC
PORTB	8 (PB0-PB7)	Interrupts
PORTC	8 (PC0-PC7)	UART,I2C,PWM
PORTD	8 (PD0-PD7)	Parallel Slave Port
PORTE	3 (PE0-PB2)	ADC

I think this is enough to make code. Let's move into the coding part.

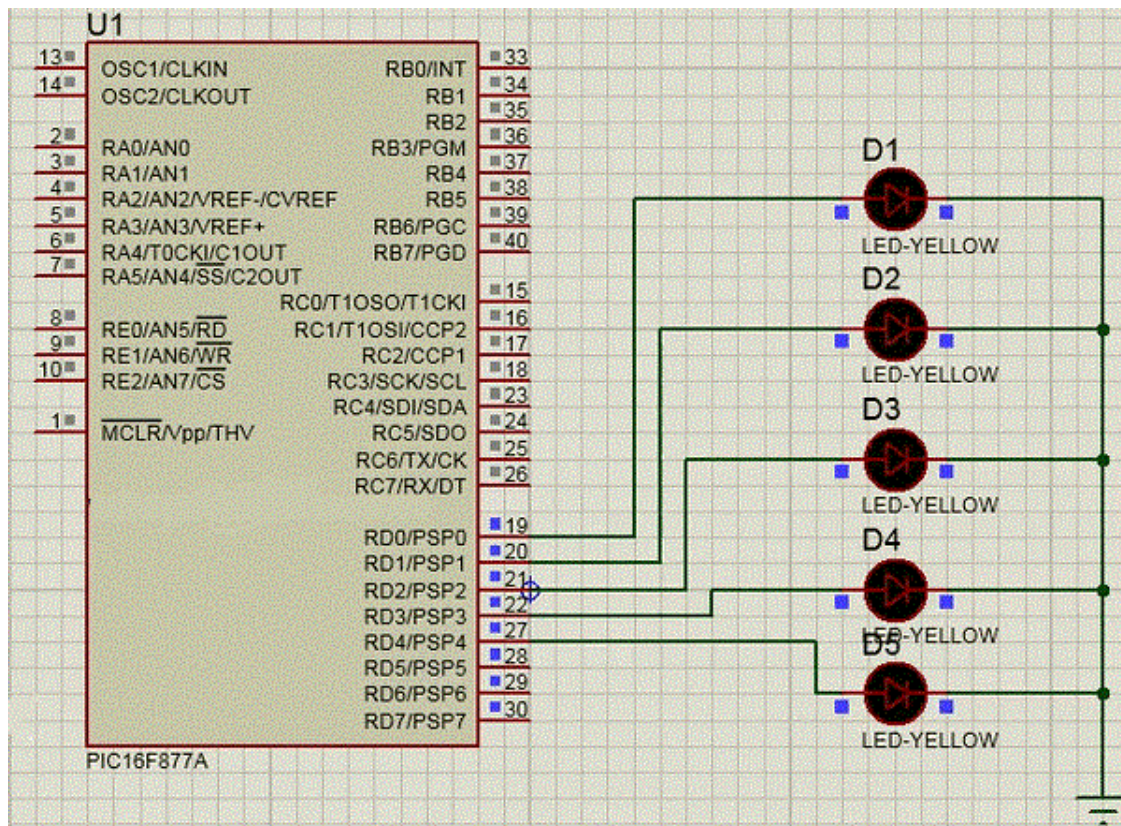
LED Interfacing

Code

In this code, I've connected LEDs to Port D.

```
#include<htc.h>
void delay()
{
    unsigned int a;
    for(a=0;a<10000;a++);
}
void main()
{
    TRISD=0; //Port D is act as Output
    while(1) {
        PORTD=0xFF; //Port D ON
        delay();
        PORTD=0x00; //Port D OFF
        delay();
    }
}
```

Output



Switch Interfacing

Code

In this code, Switch is connected into Port D. LEDs are connected into Port B.

```
#include<htc.h>
```

```
#define sw RD0 //Switch is connected at PORTD.0
```

```
void delay()
```

```
{
```

```
    unsigned int a;
```

```
    for(a=0;a<10000;a++);
```

```
}
```

```
void main()
```

```
{
```

```
    TRISB=0; //Port B act as Output
```

```
    TRISD=0xff; //Port D act as Input
```

```
    while(1) {
```

```
        if(!sw) {
```

```
            PORTB=0xff; //LED ON
```

```
        } else
```

```
            PORTB=0; //LED OFF
```

```
    }
```

}

PIC16F877A – LCD Interfacing Tutorial

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In our previous tutorial, we have seen how to use multiple external interrupts in PIC16F877A. In this tutorial, we will learn LCD Interfacing with PIC16F877A.

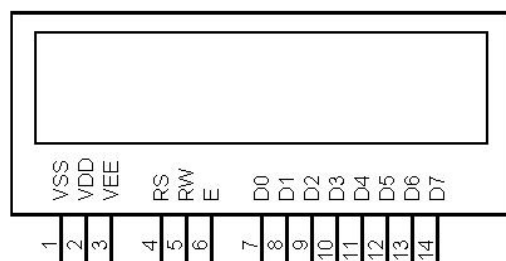
LCD Interfacing with PIC16F877A LCD DISPLAY

We always use devices made up of Liquid Crystal Displays (LCDs) like computers, digital watches and also DVD and CD players. They have become very common and have taken a giant leap in the screen industry by clearly replacing the use of Cathode Ray Tubes (CRT). CRT draws more power than LCD and are also bigger and heavier. All of us have seen a LCD, but no one knows the exact working of it. Let us take a look at the working of a LCD.

Here we are using alphanumeric LCD 16×2. A 16×2 LCD display is very basic module and is very commonly used in various devices and circuits. These modules are preferred over seven segments and other multi segment LEDs. The reasons being: LCDs are economical; easily programmable; have no limitation of displaying special & even custom characters (unlike in seven segments), animations and so on.

A 16×2 LCD means it can display 16 characters per line and there are 2 such lines. In this LCD each character is displayed in 5×7 pixel matrix. This LCD has two registers, namely, Command and Data. The command register stores the command instructions given to the LCD. A command is an instruction given to LCD to do a predefined task like initializing it, clearing its screen, setting the cursor position, controlling display etc. The data register stores the data to be displayed on the LCD. The data is the ASCII value of the character to be displayed on the LCD.

Pin Diagram



Pin Description

Pin	Function	Name
-----	----------	------

No		
1	Ground (0V)	Ground
2	Supply voltage; 5V (4.7V – 5.3V)	V _{CC}
3	Contrast adjustment; through a variable resistor	V _{EE}
4	Selects command register when low; and data register when high	Register Select
5	Low to write to the register; High to read from the register	Read/write
6	Sends data to data pins when a high to low pulse is given	Enable
7	8-bit data pins	DB0
8		DB1
9		DB2
10		DB3
11		DB4
12		DB5
13		DB6
14		DB7
15	Backlight V _{CC} (5V)	Led+
16	Backlight Ground (0V)	Led-

The LCD display module requires 3 control lines as well as either 4 or 8 I/O lines for the data bus. The user may select whether the LCD is to operate with a 4-bit data bus or an 8-bit data bus. If a 4-bit data bus is used the LCD will require a total of 7 data lines (3 control lines plus the 4 lines for the data bus). If an 8-bit data bus is used the LCD will require a total of 11 data lines (3 control lines plus the 8 lines for the data bus).

The three control lines are referred to as EN, RS, and RW.

The EN line is called “Enable.” This control line is used to tell the LCD that you are sending it data. To send data to the LCD, your program should make sure this line is low (0) and then set the other two control lines and/or put data on the data bus. When the other lines are completely ready, bring ENhigh (1) and wait for the minimum amount of time required by the LCD datasheet (this varies from LCD to LCD), and end by bringing it low (0) again.

The RS line is the “Register Select” line. When RS is low (0), the data is to be treated as a command or special instruction (such as clear screen, position cursor, etc.). When RS is high (1), the data being sent is text data which should be displayed on the screen. For example, to display the letter “T” on the screen you would set RS high.

The RW line is the “Read/Write” control line. When RW is low (0), the information on the data bus is being written to the LCD. When RW is high (1), the program is effectively querying (or reading) the LCD. Only one instruction (“Get

LCD status”) is a read command. All others are write commands—so RW will almost always be low.

Finally, the data bus consists of 4 or 8 lines (depending on the mode of operation selected by the user). In the case of an 8-bit data bus, the lines are referred to as DB0, DB1, DB2, DB3, DB4, DB5, DB6, and DB7.

LCD Commands

Code (Hex)	Command to LCD Instruction Register
1	Clear Display screen
2	Return home
4	Decrement cursor (Shift cursor to left)
6	Increment cursor (Shift cursor to Right)
5	Shift display right
7	Shift display left
8	Display off, cursor off
A	Display on, cursor off
C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80	Force cursor to beginning of 1 st line
0C0	Force cursor to beginning of 2 nd line
38	2 lines and 5x7 Matrix

Now let's move to programming.

Connections

RS – Port C .0 (RC0)

RW – Port C.1 (RC1)

EN – Port C.2 (RC2)

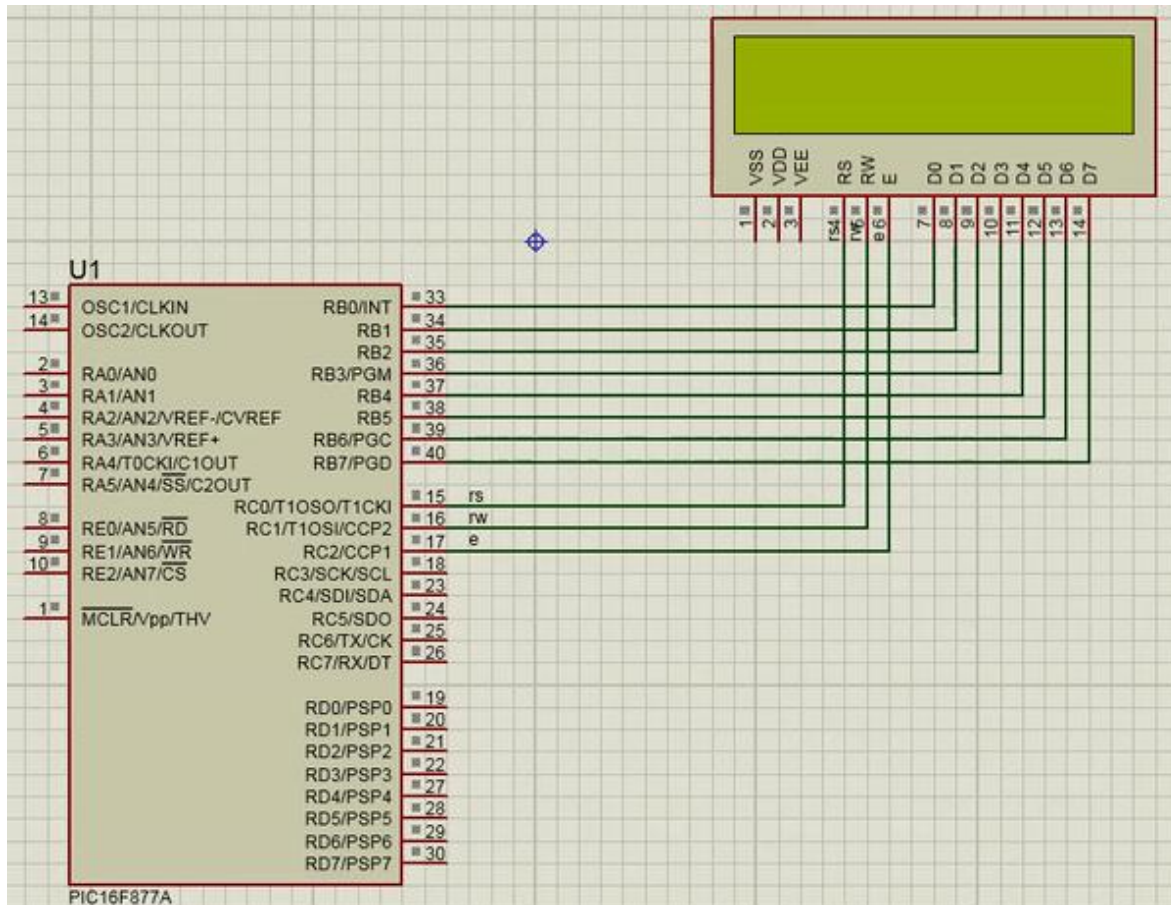
Data lines – Port B

Code

```
#include<pic.h>
#define rs RC0
#define rw RC1
#define en RC2
#define delay for(j=0;j<1000;j++)
int j;
void lcd_init();
void cmd(unsigned char a);
void dat(unsigned char b);
void show(unsigned char *s);
__CONFIG( FOSC_HS & WDTE_OFF & PWRTE_OFF & CP_OFF &
BOREN_ON & LVP_OFF & CPD_OFF & WRT_OFF & DEBUG_OFF);
void main()
{
    unsigned int i;
    TRISB=TRISC0=TRISC1=TRISC2=0;
    lcd_init();
    cmd(0x8A); //forcing the cursor at 0x8A position
    show("WELCOME TO Pantech");
    while(1) {
        for(i=0;i<15000;i++);
        cmd(0x18);
        for(i=0;i<15000;i++);
    }
}
void lcd_init()
{
    cmd(0x38);
    cmd(0x0c);
```

```
cmd(0x06);
cmd(0x80);
}
void cmd(unsigned char a)
{
PORTB=a;
rs=0;
rw=0;
en=1;
delay;
en=0;
}
void dat(unsigned char b)
{
PORTB=b;
rs=1;
rw=0;
en=1;
delay;
en=0;
}
void show(unsigned char *s)
{
while(*s) {
dat(*s++);
}
}
```

Output – LCD Interfacing with PIC16F877A



PIC16F877A – DC Motor Interfacing

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In the previous tutorial, we have used that LCD in 4-bit mode. In this tutorial, we are going to learn DC Motor Interfacing with PIC16F877A.

Motor Driver Working (L293D)

Introduction

When we talk about controlling the robot, the first thing that comes into mind is controlling DC motors. Interfacing DC motor to the microcontroller is a very important concept in Robotic applications. By interfacing the DC motor to the microcontroller, we can do many things like controlling the direction of the motor, controlling the speed of the motor. This article describes how to control the DC motor using the PIC16F877A controller.

EN Pin High (En1 = 1 or En2 = 1)

Input 1 or Input 3 Pin High (In1 = 1 or In3=1)
 Input 2 or Input 4 Pin Low (In2 = 0 or In4 = 0)

Reverse

EN Pin High (En1 = 1 or En2 = 1)
 Input 1 or Input 3 Pin Low (In1 = 0 or In3=0)
 Input 2 or Input 4 Pin Low (In2 = 1 or In4 = 1)

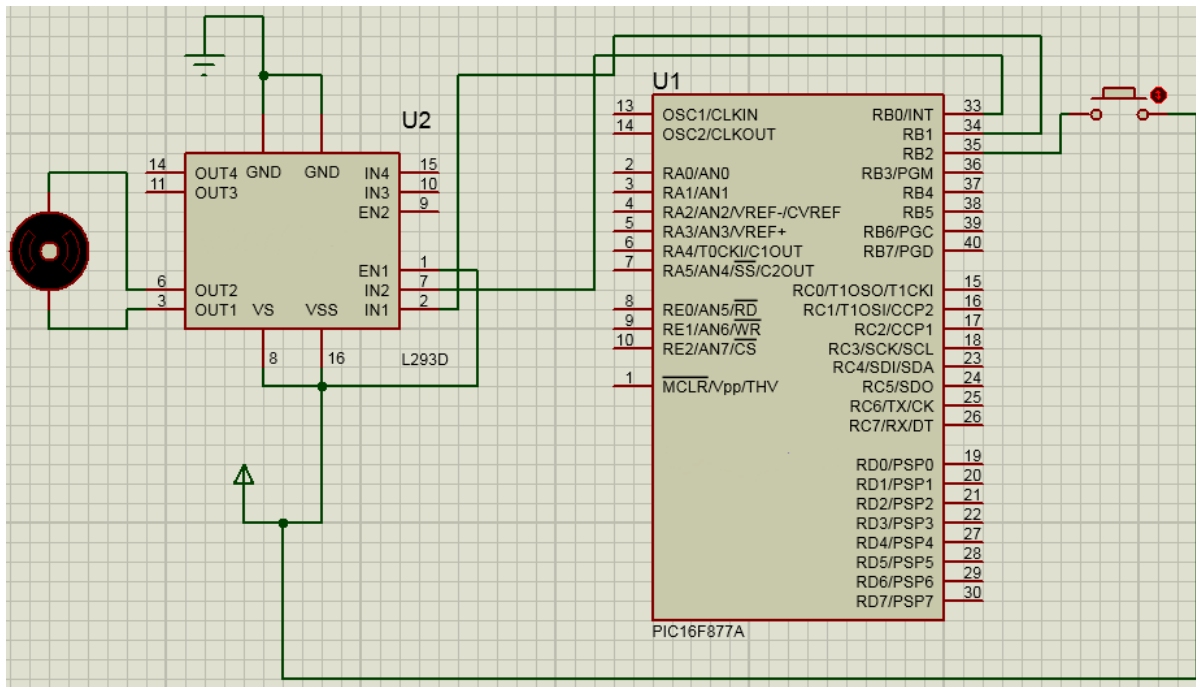
Code

In this code, the motor will rotate forward when we press the button. Here I've connected buttons one end into +5v. So in the if function I'm checking with 1

If (SW==1). But if you connect that end into the ground, you should check with 0 if (SW==0).

```
#include<pic.h>
#define in1 RB0
#define in2 RB1
#define sw RB2
Void main ()
{
  TRISB0=0;
  TRISB1=0;
  TRISB2=1;
  while(1) {
    if(sw==1) {
      in1=1;
      in2=0;
    } else {
      in1=in2=0;
    }
  }
}
```

Output



Notes:

The maximum current capacity of L293 is 600mA/channel. So do not use a motor that consumes more than that.

The supply voltage range of L293 is between 4.5 and 36V DC. So you can use a motor falling in that range.

Mostly in Robotic application, we will use DC gear Motor. So the same logic is used for that Gear motor also.

PIC16F877A – Relay Interfacing

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In the previous tutorial, we have interfaced the DC motor with PIC16F877A. In this tutorial, we will learn Relay interfacing with PIC16F877A. Relay is a very important component to interface the heavy appliances with the help of a microcontroller.

Relay

Relays are devices that allow low power circuits to **switch** a relatively high Current/Voltage ON/OFF. A relay circuit is typically a smaller switch or device

which drives (opens/closes) an electric switch that is capable of carrying much larger current amounts.

Principle

Current flowing through the coil of the relay creates a magnetic field that attracts a lever and changes the switch contacts. The coil current can be on or off so relays have two switch positions and most have double throw (changeover) switch contacts.

Construction and working

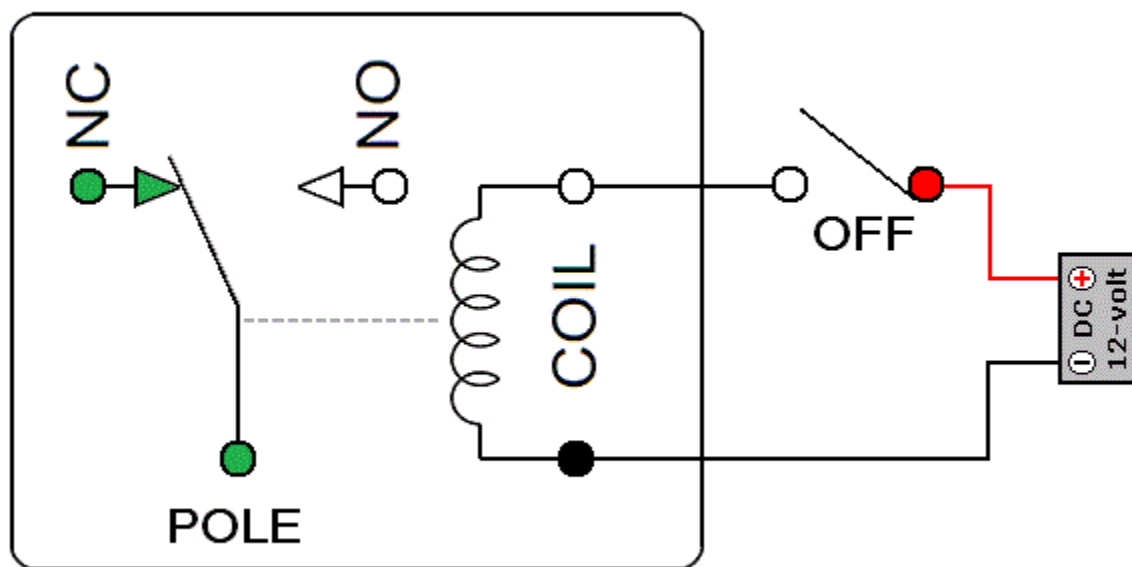
Relays are made up of an electromagnet and a set of contacts generally based on Single Pole Double Throw (SPDT) or Double Pole Double Throw (DPDT) switching method. It has 3 pins to perform a function –

COM = Common, always connect to NC; it is the moving part of the switch.

NC = Normally Closed, COM is connected to this when the relay coil is off.

NO = Normally Open, COM is connected to this when the relay coil is on.

You can easily understand the concept of Relay by looking the below image.



Prerequisites

Before Interfacing, everyone should know about the Relay Driver which is used to interface the relay to the microcontroller.

L293d – Relay Driver Working

Relay Interfacing with PIC16F877A

Circuit Diagram

In this tutorial, I'm connecting One Relay in Port B.0. And the switch is connected into Port B.1.

PIC16F877A – LCD 4Bit Interfacing

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In the previous LCD tutorial, we have used that LCD in 8-bit mode. But here we will see LCD 4-bit interfacing with PIC16F877A Microcontroller.

Introduction

8-bit mode – Using 8 data lines in LCD (totally 8 data lines are there)

4-bit mode – Using only 4 data lines in the LCD module

8-bit mode is already working and that looks awesome. Then why we are going to 4-bit mode? This is the question that comes to mind whenever I said 4-bit mode. Yeah, that 8-bit mode is nice. But Just assume. I'm doing one project which requires more number of hardware. But PIC16F877A has only 33 GPIOs. So in that time I can use this 4-bit mode and reduce the pin required for the LCD module. Am I right? Great. That's why 4-bit mode also important. Already we know the LED's operation. If we want to enable 4-bit mode we have to do small modifications in the normal method. Let's see that.

In initializing time we have to give 0x28 commands. That's all.

LCD Initializing

```
void lcd_init()
{
    cmd(0x02);
    cmd(0x28);
    cmd(0x0e);
    cmd(0x06);
    cmd(0x80);
}
```

Sending command

Here everything is the same except the way of data writing. Here we have only 4 bits. So we need to send nibble by nibble. So first we need to send first nibble then followed by the second. See that code. I'm writing into Port B's last 4 bits.

Because the last 4 bits are connected to LCD.

```
void cmd(unsigned char a)
{
  rs=0;
  PORTB&=0x0F;
  PORTB|=(a&0xf0);
  en=1;
  lcd_delay();
  en=0;
  lcd_delay();
  PORTB&=0x0f;
  PORTB|=(a<<4&0xf0);
  en=1;
  lcd_delay();
  en=0;
  lcd_delay();
}
```

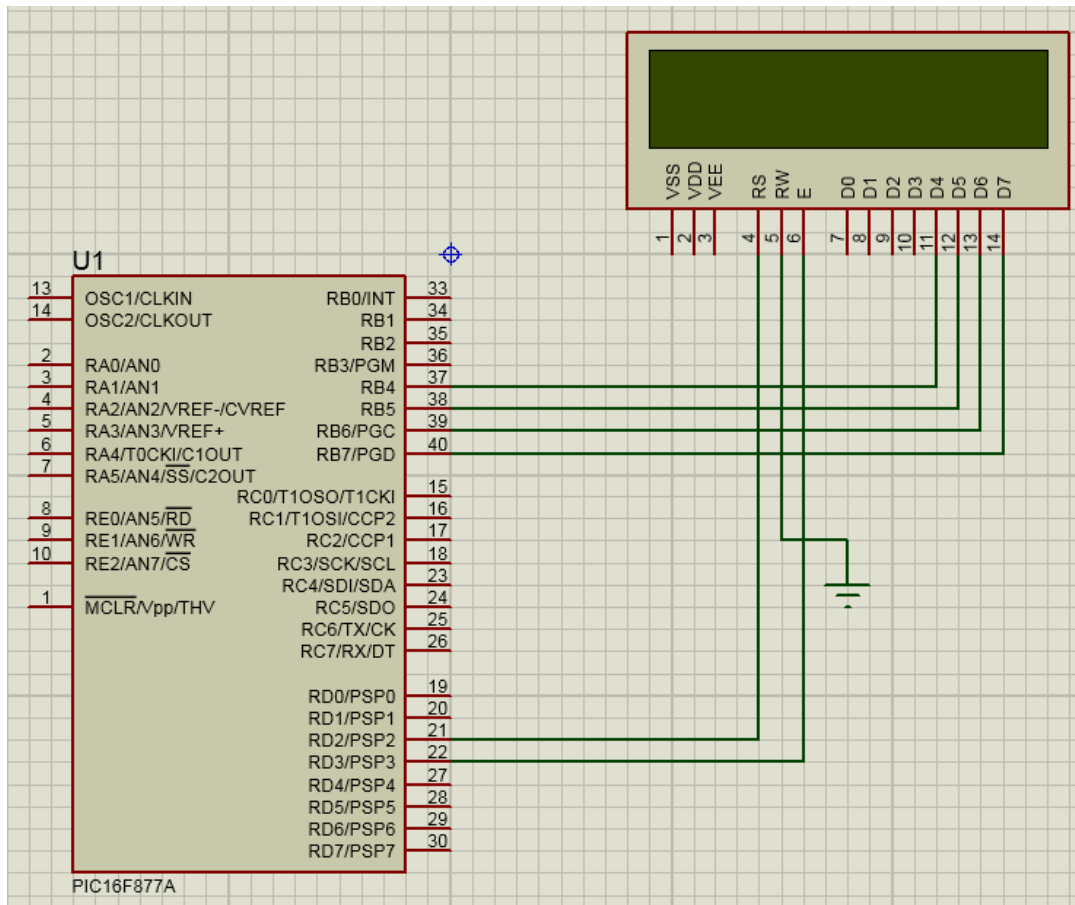
Sending Data

Same as sending the command.

```
void dat(unsigned char b)
{
  rs=1;
```

```
PORTB&=0x0F;  
PORTB|=(b&0xf0);  
en=1;  
lcd_delay();  
en=0;  
lcd_delay();  
PORTB&=0x0f;  
PORTB|=(b<<4&0xf0);  
en=1;  
lcd_delay();  
en=0;  
lcd_delay();  
}
```

Circuit Diagram



Code

```
#include <pic.h>
__CONFIG( FOSC_HS & WDTE_OFF & PWRTE_OFF & CP_OFF &
BOREN_ON & LVP_OFF & CPD_OFF & WRT_OFF & DEBUG_OFF);
#define rs RD2
#define en RD3
void lcd_init();
void cmd(unsigned char a);
void dat(unsigned char b);
void show(unsigned char *s);
void lcd_delay();
void main()
{
    unsigned int i;
    TRISB=TRISD2=TRISD3=0;
    lcd_init();
    cmd(0x90);
```

```

show("www.EmbeTronicX.com");
while(1)
{
for(i=0;i<15000;i++);
cmd(0x18);
for(i=0;i<15000;i++);
}
}
void lcd_init()
{
cmd(0x02);
cmd(0x28);
cmd(0x0e);
cmd(0x06);
cmd(0x80);
}
void cmd(unsigned char a)
{
rs=0;
PORTB&=0x0F;
PORTB|=(a&0xf0);
en=1;
lcd_delay();
en=0;
lcd_delay();
PORTB&=0x0f;
PORTB|=(a<<4&0xf0);
en=1;
lcd_delay();
en=0;
lcd_delay();
}
void dat(unsigned char b)
{
rs=1;
PORTB&=0x0F;
PORTB|=(b&0xf0);
en=1;
lcd_delay();
en=0;

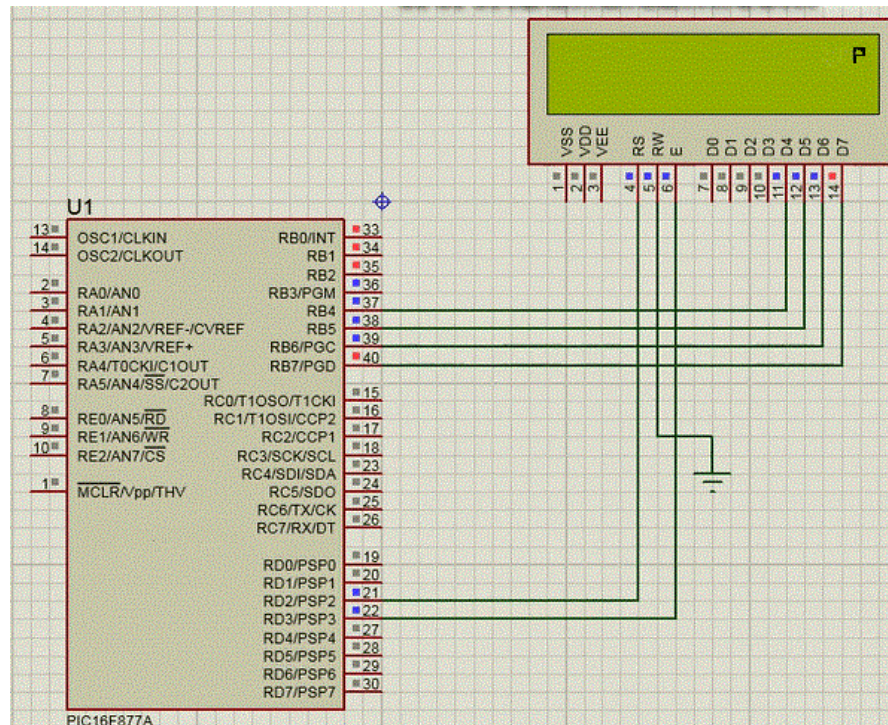
```

```

lcd_delay();
PORTB&=0x0f;
PORTB|=(b<<4&0xf0);
en=1;
lcd_delay();
en=0;
lcd_delay();
}
void show(unsigned char *s)
{
while(*s) {
dat(*s++);
}
}
void lcd_delay()
{
unsigned int lcd_delay;
for(lcd_delay=0;lcd_delay<=1000;lcd_delay++);
}

```

Output



PIC16F877A – Keypad Interfacing

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In the previous tutorial, we have interfaced the Relay with PIC16F877A. In this tutorial, we will learn Keypad Interfacing with PIC16F877A.

Prerequisites

- Matrix Keypad Operation
- LCD Interfacing (4 Bit Mode) with PIC16F877A

Components Required

- 4×4 Keypad
- PIC16F877A Microcontroller

Keypad Interfacing with PIC16F877A

Circuit Diagram

LCD:

RS – Port D.2

RW – GND

EN – Port D.3

Data Lines – Port D.4 – Port D.7

Keypad:

Row1 – RB0

Row2 – RB1

Row3 – RB2

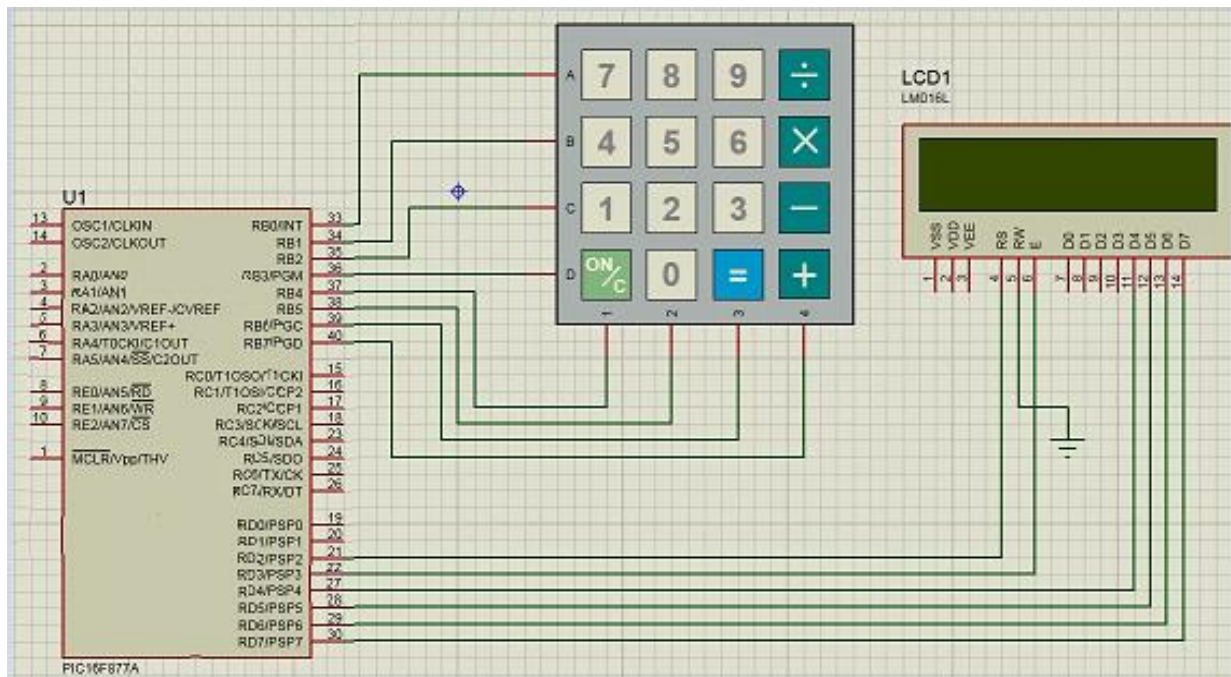
Row4 – RB3

Col1 – RB4

Col2 – RB5

Col3 – RB6

Col4 – RB7



Code

```
#include <pic.h>
```

```
__CONFIG( FOSC_HS & WDTE_OFF & PWRTE_OFF & CP_OFF &
BOREN_ON & LVP_OFF & CPD_OFF & WRT_OFF & DEBUG_OFF);
```

```
#define rs RD2
```

```
#define en RD3
```

```
#define R1 RB0
```

```
#define R2 RB1
```

```
#define R3 RB2
```

```
#define R4 RB3
```

```
#define C1 RB4
```

```
#define C2 RB5
```

```
#define C3 RB6
```

```
#define C4 RB7
```

```
void lcd_init();
```

```
void cmd(unsigned char a);
```

```

void dat(unsigned char b);
void show(unsigned char *s);
void lcd_delay();

```

```

unsigned char key();
void keyinit();

```

```

unsigned      char      keypad[4][4]={ {'7','8','9','/'},{ '4','5','6','*'},{'1','2','3','-
'},{'C','0','=','+'}}};
unsigned char rowloc,colloc;

```

```

void main()
{
    unsigned int i;
    TRISD=0;
    lcd_init();
    keyinit();
    unsigned char b;
    cmd(0x80);
    show(" Enter the Key ");
    while(1)
    {
        cmd(0xc7);
        b=key();
        dat(b);

    }
}

```

```

void lcd_init()
{
    cmd(0x02);
    cmd(0x28);
    cmd(0x0e);
    cmd(0x06);
    cmd(0x80);
}

```

```

void cmd(unsigned char a)
{

```

```

    rs=0;
    PORTD&=0x0F;
    PORTD|=(a&0xf0);
    en=1;
    lcd_delay();
    en=0;
    lcd_delay();
    PORTD&=0x0f;
    PORTD|=(a<<4&0xf0);
    en=1;
    lcd_delay();
    en=0;
    lcd_delay();
}

void dat(unsigned char b)
{
    rs=1;
    PORTD&=0x0F;
    PORTD|=(b&0xf0);
    en=1;
    lcd_delay();
    en=0;
    lcd_delay();
    PORTD&=0x0f;
    PORTD|=(b<<4&0xf0);
    en=1;
    lcd_delay();
    en=0;
    lcd_delay();
}

void show(unsigned char *s)
{
    while(*s) {
        dat(*s++);
    }
}

void lcd_delay()

```

```

{
    unsigned int lcd_delay;
    for(lcd_delay=0;lcd_delay<=1000;lcd_delay++);
}

void keyinit()
{
    TRISB=0XF0;
    OPTION_REG&=0X7F;      //ENABLE PULL UP
}

unsigned char key()
{
    PORTB=0X00;
    while(C1&&C2&&C3&&C4);
    while(!C1||!C2||!C3||!C4) {
        R1=0;
        R2=R3=R4=1;
        if(!C1||!C2||!C3||!C4) {
            rowloc=0;
            break;
        }
        R2=0;R1=1;
        if(!C1||!C2||!C3||!C4) {
            rowloc=1;
            break;
        }
        R3=0;R2=1;
        if(!C1||!C2||!C3||!C4) {
            rowloc=2;
            break;
        }
        R4=0; R3=1;
        if(!C1||!C2||!C3||!C4){
            rowloc=3;
            break;
        }
    }
    if(C1==0&&C2!=0&&C3!=0&&C4!=0)
        colloc=0;
}

```



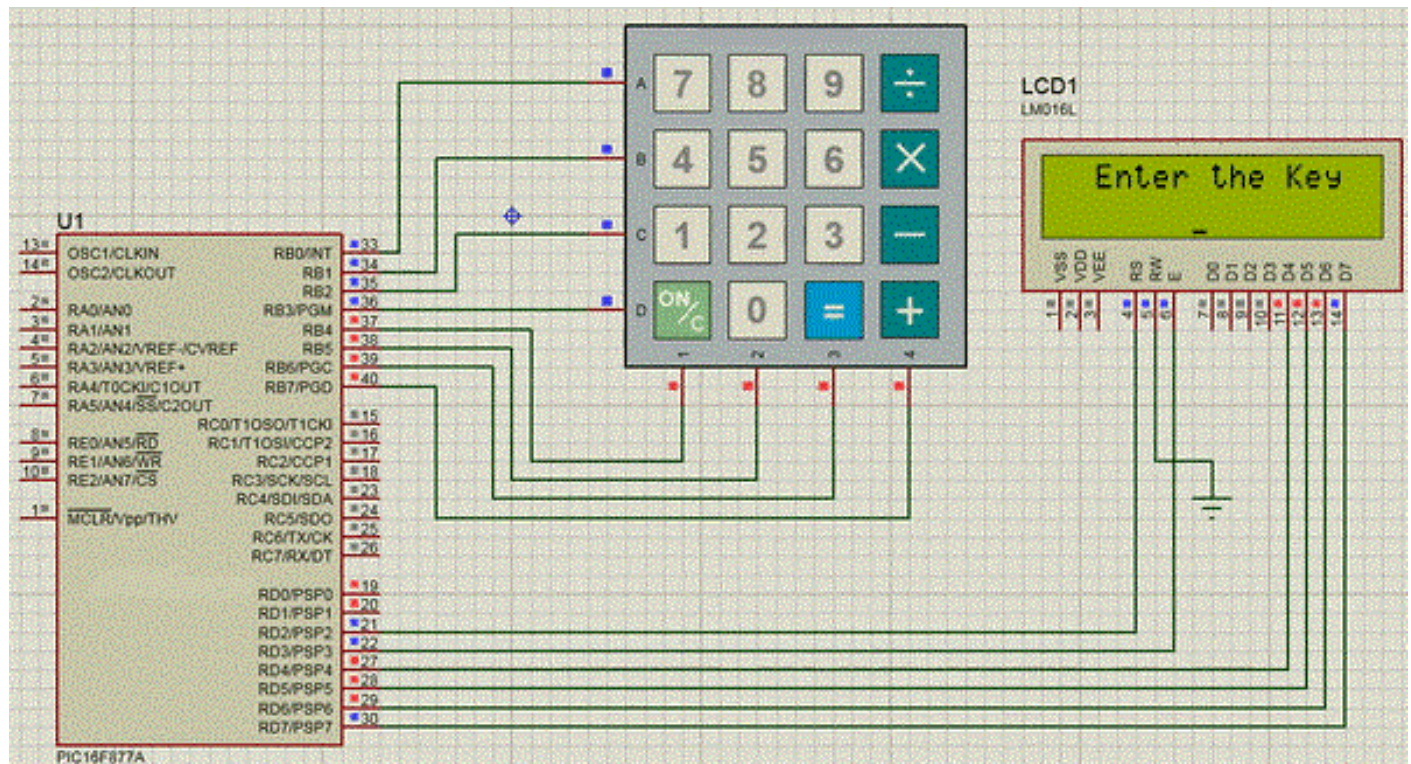
```

else if(C1!=0&&C2==0&&C3!=0&&C4!=0)
    colloc=1;
else if(C1!=0&&C2!=0&&C3==0&&C4!=0)
    colloc=2;
else if(C1!=0&&C2!=0&&C3!=0&&C4==0)
    colloc=3;
while(C1==0||C2==0||C3==0||C4==0);
return (keypad[rowloc][colloc]);
}

```

Output

Whatever we are typing in the keypad, it will display that character in LCD Module.



PIC16F877A – Timer/Counter

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In our previous tutorial, we have seen LED Interfacing with PIC16F877A (GPIO). Now we will see the PIC16F877A Timer Tutorial. If you read this tutorial, you could generate this precise delay for your applications.

PIC16F877A Timer Tutorial

As the name suggests these are used to measure the time or generate the accurate time delay. The microcontroller can also generate/measure the required time delays by running loops, but the timer relieves the CPU from that redundant and repetitive task, allowing it to allocate maximum processing time for other tasks.

The timer is nothing but a simple binary counter that can be configured to count clock pulses(Internal/External). Once it reaches the max value, it will roll back to zero setting up an OverFlow flag and generates the interrupt if enabled.

PIC16F877a has three timers.

Timer0 (8-bit timer)

Timer1 (16-bit timer)

Timer2 (8-bit timer)

All Timers can act as a timer or counter or PWM Generation. now we will see each one by one.

To start using a timer we should understand some of the fancy terms like **8-bit/16-bit timer, Prescaler, Timer interrupts, and Fosc**. Now, let us see what each one really means. As said earlier there are both the 8-bit and 16-bit Timers in our PIC16F877A. The main difference between them is that the 16-bit Timer has a much better Resolution than the 8-bit Timer.

Prescaler is a name for the part of a microcontroller that divides oscillator clock before it will reach logic that increases timer status. The range of the Prescaler id is from 1 to 256 and the value of the Prescaler can be set using the OPTION Register (we will see this register later).

As the timer increments and when it reaches its maximum value of 255 (for 8-bit timers) or 65536 (for 16-bit timers), it will trigger an interrupt and initialize itself to 0 back again. This interrupt is called as the Timer Interrupt. This interrupt informs the MCU that this particular time has lapped.

The **Fosc stands for Frequency of the Oscillator**, it is the frequency of the Crystal used. The time taken for the Timer register depends on the value of Prescaler and the value of the Fosc.

Timer 0

- The Timer0 module timer/counter has the following features:
- 8-bit timer/counter
- Readable and writable
- 8-bit software programmable Prescaler
- Internal or external clock select
- Interrupt on overflow from FFh to 00h
- Edge select for external clock

Registers used for Timer0

- OPTION_REG
- TMR0
- INTCON

OPTION_REG

We perform all the necessary settings with OPTION_REG Register. The size of the register is 8 bits.

OPTION_REG REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

R = Readable bit W = Writable bit U = Unimplemented bit, read as '0' -n = Value at POR '1' = Bit is set '0' = Bit is cleared x = Bit is unknown

RBPU: PORTB Pull-up Enable bit (This bit is not used for timers)

- 1 = PORTB pull-ups are disabled
- 0 = PORTB pull-ups are enabled by individual port latch values

INTEDG (This bit is not used for timers)

T0CS: TMR0 Clock Source Select bit

- 1 = Transition on T0CKI pin
- 0 = Internal instruction cycle clock (CLKO)

T0SE: TMR0 Source Edge Select bit

- 1 = Increment on high-to-low transition on T0CKI pin
- 0 = Increment on low-to-high transition on T0CKI pin

PSA: Prescaler Assignment bit

- 1 = Prescaler is assigned to the WDT
- 0 = Prescaler is assigned to the Timer0 module

PS2:PS0: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Note: There is only one Prescaler available which is mutually exclusively shared between the Timer0 module and the Watchdog Timer. A Prescaler assignment for the Timer0 module means that there is no Prescaler for the Watchdog Timer and vice versa. This Prescaler is not accessible but can be configured using PS2:PS0 bits of OPTION_REG.

INTCON Register

INTCON REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
bit 7							bit 0

R = Readable bit
'1' = Bit is set

W = Writable bit
'0' = Bit is cleared

U = Unimplemented bit, read as '0'
x = Bit is unknown

- n = Value at POR

GIE: Global Interrupt Enable bit

- 1-Enables all unmasked interrupts
- 0-Disables all interrupts

PIE: Peripheral Interrupt Enable bit

- 1-Enables all unmasked peripheral interrupts
- 0-Disables all peripheral interrupts

TMR0IE: TMR0 Overflow Interrupt Enable bit

- 1-Enables the TMR0 interrupt
- 0-Disables the TMR0 interrupt

INTE: Not for Timers

RBIE: Not for Timers

TMR0IF: TMR0 Overflow Interrupt Flag bit

- 1-TMR0 register has overflowed (must be cleared in software)
- 0-TMR0 register did not overflow

INTF: Not for Timers

RBIF: Not for Timers

TMR0 Register

This is the 8-bit register that holds the timer values. For example, initially, it will be 0. It will increment by one per one clock cycle. When it reaches 255, it will trigger the TMR0IF bit in INTCON Register. Then again starts from 0.

Delay Calculation for 1 second

$$f_{out} = \frac{f_{clk}}{4 * \text{Prescaler} * (256 - \text{TMR0}) * \text{Count}} \quad \text{where} \quad T_{out} = \frac{1}{f_{out}}$$

Here, My fclk = 11.0592MHz (You can put your board's fclk)

Prescaler = 256 (It is based on PS0 – PS2 bits in OPTION_REG)

TMR0 = 0. (My TMR0's value will be 0)

Desire Delay (Tout = 1 second) So Fout = 1 (Tout = 1/Fout)

Apply this value to that above formula.

Count = 11059200 / (4*256*256*1)

Count = 42.1875 (approximately 42).

Timer0 Code

In this code LED is connected to Port B. Those LEDs are blinking every 1 second.

```
#include<pic.h>
void t0delay();
void main()
{
    TRISB=0;
    OPTION_REG=0x07; //Prescale is assigned to Timer 0, Prescaler value = 256,
    Fclk = 11.0592MHz
    while(1) {
        PORTB=0xff;
        t0delay();
        PORTB=0x00;
        t0delay();
    }
}
void t0delay()      // 1 second
{
    int i;
    for(i=0;i<42;i++) {
        while(!T0IF);
        T0IF=0;
    }
}
```

Timer 1

The timer TMR1 module is a 16-bit timer/counter with the following features:

- 16-bit timer/counter with two 8-Bit registers TMR1H/TMR1L
- Readable and writable
- software programmable Prescaler up to 1:8
- Internal or external clock select
- Interrupt on overflow from FFFFh to 00h
- Edge select for external clock

Registers used for Timer1

- T1CON
- TMR1 (TMR1H, TMR1L)
- PIR1

T1CON Register

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

R = Readable bit
'1' = Bit is set

W = Writable bit
'0' = Bit is cleared

U = Unimplemented bit, read as '0'
x = Bit is unknown

- n = Value at POR

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

T1OSCEN: Timer1 Oscillator Enable Control bit

- 1-Oscillator is enabled
- 0-Oscillator is shut-off

T1SYNC: Timer1 External Clock Input Synchronization Control bit

- 1-Does not synchronize external clock input
- 0-Synchronize external clock input

TMR1CS: Timer1 Clock Source Select bit

- 1-External clock from pin RC0/T1OSO/T1CKI (on the rising edge)
- 0-Internal clock (FOSC/4)

TMR1ON: Timer1 On bit

- 1-Enables Timer1
- 0-Stops Timer1

TMR1 Register

Timer1 has a register called the TMR1 register, which is 16 bits in size.

Actually, the TMR1 consists of two 8-bits registers:

- TMR1H
- TMR1L

It increments from 0000h to the maximum value of 0xFFFFh (or 65,535 decimal). The TMR1 interrupt, if enabled, is generated on overflow which is latched in the interrupt flag bit, TMR1IF (PIR1<0>). This interrupt can be enabled/disabled by setting/clearing TMR1 interrupt enable bit, TMR1IE (PIE1<0>). You can initialize the value of this register to whatever you want (not necessarily "0").

PIR1 Register

This register contains the Timer1 overflow flag. (TMR1IF).

TMR1IF – TMR1 overflow Interrupt Flag bit.

This flag marks the end of ONE cycle count. The flag needs to be reset in the software if you want to do another cycle count. We can read the value of the register TMR1 and write it into it. We can reset its value at any given moment (write) or we can check if there is a certain numeric value that we need (read).

Delay Calculation for 1 second

$f_{out} = \frac{f_{clk}}{4 * \text{Prescaler} * (65536 - \text{TMR1}) * \text{Count}}$	where	$T_{out} = \frac{1}{f_{out}}$
---	-------	-------------------------------

- Here, My fclk = 11.0592MHz (You can put your board's fclk)
- Prescaler = 1 (It is based on T1CKPS1 – T1CKPS0 bits in T1CON Register)
- TMR1 = 0. (My TMR1's value will be 0)
- Desire Delay (Tout = 1 second) So Fout = 1 (Tout = 1/Fout)
- Apply this values to that above formula.
- Count = 11059200 / (4*1*65536*1)
- Count = 42.1875 (approximately 42).

Timer1 Code

In this code LED is connected to Port B. Those LEDs are blinking every 1 second.

```
#include<pic.h>
```

```
void t1delay();
```

```
void main()
```

```
{
```

```
TRISB=0;
```

```
T1CON=0x01; //Prescale value = 1:1, It using Internal clock, Timer 1 ON
```

```
while(1) {
```

```
PORTB=0xff;
```

```
t1delay();
```

```
PORTB=0;
```

```
t1delay();
```

```
}
```

```
}
```

```
void t1delay()
```

```
{
```

```
int i;
```



```

for(i=0;i<42;i++) {
TMR1H=TMR1L=0;
while(!TMR1IF);
TMR1IF=0;
}
}

```

Timer 2

- The TIme2 module is an 8-bit timer/counter with the following features:
- 8-bit timer/counter
- Readable and writable
- Software programmable Prescaler/PostScaler up to 1:16
- Interrupt on overflow from FFh to 00h

Registers used for Timer2

- T2CON
- TMR2
- PIR1
- PR2

T2CON Register

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

R = Readable bit
'1' = Bit is set

W = Writable bit
'0' = Bit is cleared

U = Unimplemented bit, read as '0'
x = Bit is unknown

- n = Value at POR

TOUTPS3:TOUTPS0: Timer2 Output Postscale Select bits

0000 = 1:1 postscale

0001 = 1:2 postscale

0010 = 1:3 postscale

-
-
-

1111 = 1:16 postscale

TMR2ON: Timer2 On bit

- 1-Timer2 is on
- 0-Timer2 is off

T2CKPS1:T2CKPS0: Timer2 Clock Prescale Select bits

- 00 = Prescaler is 1

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

TMR2 & PR2 Register

- TMR2 – The register in which the “initial” count value is written.
- PR2 – The register in which the final or the maximum count value is written.

PIR1 Register

This register contains the Timer2 overflow flag(TMR2IF).

Delay Calculation for 1 second

$$f_{out} = \frac{f_{clk}}{4 * \text{Prescaler} * (\text{PR2} - \text{TMR2}) * \text{Postscaler} * \text{Count}} \quad \text{where} \quad T_{out} = \frac{1}{f_{out}}$$

- Here, My fclk = 11.0592MHz (You can put your board's fclk)
- Prescaler = 1 (It is based on T2CKPS1:T2CKPS0 bits in T2CON)
- Postscaler = 16 (It is based on TOUTPS3:TOUTPS0 bits in T2CON)
- TMR2 = 0. (My TMR2's value will be 0)
- PR2 = 255 (My PR2's value will be 255)
- Desire Delay (Tout = 1 second) So Fout = 1 (Tout = 1/Fout)
- Apply this values to that above formula.
- Count = 11059200 / (4*1*(256-0)*16*1)
- Count = 675.

Timer2 Code

In this code LED is connected to Port B. Those LEDs are blinking every 1 second.

```
#include<pic.h>
```

```
#include<htc.h>
```

```
void t2delay();
```

```
void main()
```

```
{
```

```
    TRISB=0;
```

```
    T2CON=0b01111000;    //postscale=16,prescale=1,timer off
```

```
    while(1)
```

```
    {
```

```
        PORTB=255;
```

```
        t2delay();
```

```

    PORTB=0;
    t2delay();
}
}

void t2delay()
{
    unsigned int i;
    T2CON|=(1<<2);    //timer2 on
    for(i=0;i<675;i++)
    {
        while(!TMR2IF);
        TMR2IF=0;
    }
}

```

PIC16F877A – USART

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In our previous tutorial, we have seen PIC16F877A Timer Tutorial. In this tutorial, we are going to learn PIC16F877A Serial Communication (USART). USART (Universal Synchronous Asynchronous Receiver Transmitter) is one of the basic interfaces which provide a cost-effective simple and reliable communication between one controller to another controller or between a controller and PC.

Prerequisites

If you are new to UART please go through our previous article about UART.

- Basic UART (Serial Communication)

PIC16F877A Serial Communication Tutorial

PIC16F877A comes with inbuilt USART which can be used for Synchronous/Asynchronous communication. USART is a two-wire communication system in which the data flow serially. USART is also a full-duplex communication, which means you can send and receive data at the same

time which can be used to communicate with peripheral devices, such as CRT terminals and personal computers.

The USART can be configured in the following modes:

- Asynchronous (full-duplex)
- Synchronous – Master (half-duplex)
- Synchronous – Slave (half-duplex)

But here We will be discussing only the UART (Asynchronous).

Registers used for Serial Communication

- TXSTA (Transmit Status And Control Register)
- RCSTA (Receive Status And Control Register)
- SPBRG (USART Baud Rate Generator)
- TXREG (USART Transmit Register)
- RCREG (USART Receiver Register)

TXSTA (Transmit Status And Control Register)

This register is used to configure the Serial communication for TX.

TXSTA: TRANSMIT STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D
bit 7							bit 0

CSRC: Clock Source Select bit (Asynchronous mode: Don't care).

TX9: 9-bit Transmit Enable bit

- 1 = Selects 9-bit transmission
- 0 = Selects 8-bit transmission

TXEN: Transmit Enable bit

- 1 = Transmit enabled
- 0 = Transmit disabled

SYNC: USART Mode Select bit

- 1 = Synchronous mode
- 0 = Asynchronous mode

BRGH: High Baud Rate Select bit

- 1 = High speed
- 0 = Low speed

TRMT: Transmit Shift Register Status bit

- 1 = TSR empty
- 0 = TSR full

TX9D: 9th bit of Transmit Data, can be a Parity bit

RCSTA (Receive Status And Control Register)

This register is used to configure the Serial communication for RX.

RCSTA: RECEIVE STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

SPEN: Serial Port Enable bit

- 1 = Serial port enabled (configures RC7/RX/DT and RC6/TX/CK pins as serial port pins)
- 0 = Serial port disabled

RX9: 9-bit Receive Enable bit

- 1 = Selects 9-bit reception
- 0 = Selects 8-bit reception

SREN: Single Receive Enable bit (Asynchronous mode: Don't care)

CREN: Continuous Receive Enable bit

Asynchronous mode:

- 1 = Enables continuous receive
- 0 = Disables continuous receive

ADDEN: Address Detect Enable bit

Asynchronous mode 9-bit (RX9 = 1):

- 1 = Enables address detection, enables interrupt and load of the receive buffer when RSR is set
- 0 = Disables address detection, all bytes are received and the ninth bit can be used as a parity bit

FERR: Framing Error bit

- 1 = Framing error (can be updated by reading RCREG register and receive next valid byte)
- 0 = No framing error

OERR: Overrun Error bit

- 1 = Overrun error (can be cleared by clearing bit CREN)
- 0 = No overrun error

RX9D: 9th bit of Received Data (can be parity bit but must be calculated by user firmware)

SPBRG (USART Baud Rate Generator)

The main criteria for UART communication are its baud rate. Both the devices Rx/Tx should be set to the same baud rate for successful communication. This can be achieved by the SPBRG register. SPBRG is an 8-bit register that controls the baud rate generation. The SPBRG register controls the period of a free-running 8-bit timer. In Asynchronous mode, bit BRGH (TXSTA<2>) also controls the baud rate. In Synchronous mode, bit BRGH is ignored.

Given the desired baud rate and FOSC, the nearest integer value for the SPBRG register can be calculated using the below formula.

BAUD RATE FORMULA

SYNC	BRGH = 0 (Low Speed)	BRGH = 1 (High Speed)
0	(Asynchronous) Baud Rate = $F_{osc}/(64 (X + 1))$	Baud Rate = $F_{osc}/(16 (X + 1))$
1	(Synchronous) Baud Rate = $F_{osc}/(4 (X + 1))$	N/A

Legend: X = value in SPBRG (0 to 255)

Calculation:

- My Fosc = 11.0592MHz (You can put your board Fosc)
- Baud Rate = 9600
- $9600 = 11059200 / (64X + 64)$
- $64X + 64 = 1152$
- $X = 17$.

If we want to generate 9600 Baudrate (Fosc = 11.0592MHz) you have to set 17 to SPBRG Register.

TXREG (USART Transmit Register)

This is like a transmit buffer. But it is only an 8-bit register. So we have to place the character whatever we want to transmit via USART.

RCREG (USART Receiver Register)

This is like a receiver buffer. But it is only an 8-bit register. So we can take the character whatever we microcontroller received via USART.

Programming

Initializing USART

- Configure the TXSTA Register for Transmit
- Configure the RCSTA Register for Receiver

- Feed the value for baud rate that you have calculated using the above formula to SPBRG Register

```
void ser_int()
{
TXSTA=0x20; //BRGH=0, TXEN = 1, Asynchronous Mode, 8-bit mode
RCSTA=0b10010000; //Serial Port enabled,8-bit reception
SPBRG=17; //9600 baudrate for 11.0592Mhz
TXIF=RCIF=0;
}
```

Transmit

- Load the new char to be transmitted into THR.
- Wait till the char is transmitted. TXIF will be set when the TXREG is empty.
- Clear the TXIF for the next cycle.

```
void tx(unsigned char a)
{
TXREG=a;
while(!TXIF);
TXIF = 0;
}
```

Receive

- Wait till the Data is received. RCIF will be set once the data is received in the RCREG register.
- Clear the receiver flag(RCIF) for the next cycle.
- Copy/Read the received data from the RCREG register.

```
unsigned char rx()
{
while(!RCIF);
RCIF=0;
return RCREG;
}
```

Full code

This program first transmits some strings (EmbeTronicX: Enter the letters on the keyboard). Then it will act as an echo. Whatever we pressed on the keyboard, it will print that in the serial terminal.

```
#include<htc.h>
__CONFIG( FOSC_HS & WDTE_OFF & PWRTE_OFF & CP_OFF &
BOREN_ON & LVP_OFF & CPD_OFF & WRT_OFF & DEBUG_OFF);
void ser_int();
void tx(unsigned char);
unsigned char rx();
void txstr(unsigned char *);
void main()
{
    TRISC6=0; //Output (TX)
    TRISC7=1; //Input (RX)
    ser_int();
    txstr("(Pantech): Enter the letters in keyboard\n\r\r");
    while(1) {
        tx(rx());
    }
}
void ser_int()
{
    TXSTA=0x20; //BRGH=0, TXEN = 1, Asynchronous Mode, 8-bit mode
    RCSTA=0b10010000; //Serial Port enabled,8-bit reception
    SPBRG=17; //9600 baudrate for 11.0592Mhz
    TXIF=RCIF=0;
}
void tx(unsigned char a)
{
    TXREG=a;
    while(!TXIF);
    TXIF = 0;
}
```



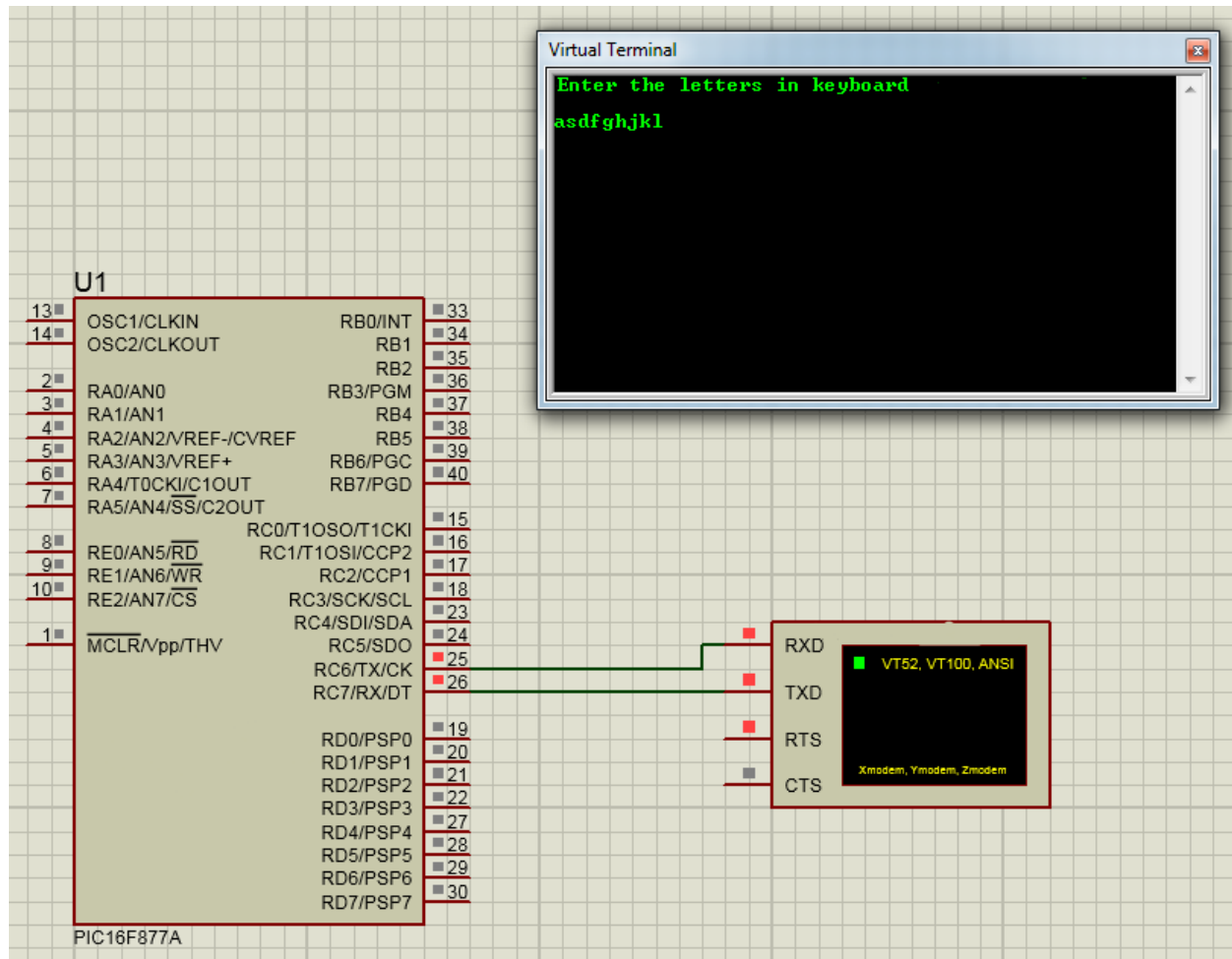
```

unsigned char rx()
{
    while(!RCIF);
    RCIF=0;
    return RCREG;
}

void txstr(unsigned char *s)
{
    while(*s) {
        tx(*s++);
    }
}

```

Output



PIC16F877A – ADC

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In our previous tutorial, we have seen PIC16F877A USART Tutorial. Now we are going to see PIC16F877A ADC Tutorial.

PIC16F877A ADC Tutorial

Prerequisites

- LCD Interfacing with PIC16F877A

Introduction

Microcontrollers are very useful especially when it comes to communicating with other devices, such as sensors, motors, switches, memory, and even another microcontroller. As we all know many interface methods have been developed over years to solve the complex problems of balancing needs of features, cost, size, power consumption, reliability, etc. but the ADC Analog-to-Digital converter remains famous among all. Using this ADC we can connect any type of Analog sensor.

PIC16F877A ADC Module

The Analog-to-Digital (A/D) Converter module has eight for the 40/44-pin devices.

The conversion of an analog input signal results in a corresponding 10-bit digital number. The A/D module has high and low-voltage reference input that is software selectable to some combination of VDD, VSS, RA2, or RA3.

The A/D converter has the unique feature of being able to operate while the device is in Sleep mode. To operate in Sleep, the A/D clock must be derived from the A/D's internal RC oscillator.

PIC16F877A ADC Pins

ADC Channel	Pin
Channel 0	RA0/AN0 (Port A)
Channel 1	RA1/AN1 (Port A)
Channel 2	RA2/AN2/VRef- (Port A)
Channel 3	RA3/AN3/VRef+ (Port A)

ADC Channel	Pin
Channel 4	RA5/AN4 (Port A)
Channel 5	RE0/AN5 (Port E)
Channel 6	RE1/AN6 (Port E)
Channel 7	RE2/AN7 (Port E)

Registers used for ADC

- A/D Control Register 0 (ADCON0)
- A/D Control Register 1 (ADCON1)
- A/D Result High Register (ADRESH)
- A/D Result Low Register (ADRESL)
- A/D Control Register 0 (ADCON0)

The ADCON0 register, shown in the below image, controls the operation of the A/D module i.e. Used to Turn ON the ADC, Select the Sampling Freq, and also Start the conversion.

ADCON0 REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	—	ADON
bit 7							bit 0

ADCS1-ADCS0: A/D Conversion Clock Select bits. These bits are based on ADCON1 Register's ADCS2 bit.

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	$F_{osc}/2$
0	01	$F_{osc}/8$
0	10	$F_{osc}/32$
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	$F_{osc}/4$
1	01	$F_{osc}/16$
1	10	$F_{osc}/64$
1	11	FRC (clock derived from the internal A/D RC oscillator)

CHS2-CHS0: Analog Channel Select bits

- 000 = Channel 0 (AN0)
- 001 = Channel 1 (AN1)
- 010 = Channel 2 (AN2)
- 011 = Channel 3 (AN3)
- 100 = Channel 4 (AN4)
- 101 = Channel 5 (AN5)

- 110 = Channel 6 (AN6)
- 111 = Channel 7 (AN7)

GO/DONE: A/D Conversion Status bit

When ADON = 1:

1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)

0 = A/D conversion not in progress

ADON: A/D On bit

1 = A/D converter module is powered up

0 = A/D converter module is shut-off and consumes no operating current

A/D Control Register 1 (ADCON1)

The ADCON1 register, shown below, configures the functions of the port pins i.e. Used to configure the GPIO pins for ADC. The port pins can be configured as analog inputs (RA3 can also be the voltage reference) or as digital I/O.

ADCON1 REGISTER

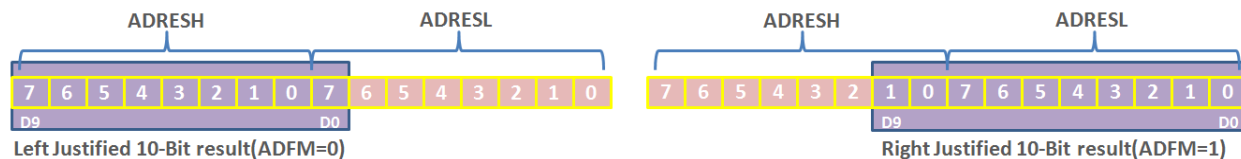
R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

ADFM: A/D Result Format Select bit

1 = Right justified. Six (6) Most Significant bits of ADRESH are read as '0'.

0 = Left justified. Six (6) Least Significant bits of ADRESL are read as '0'.

ADCS2: A/D Conversion Clock Select bit



PCFG3-PCFG0: A/D Port Configuration Control bits

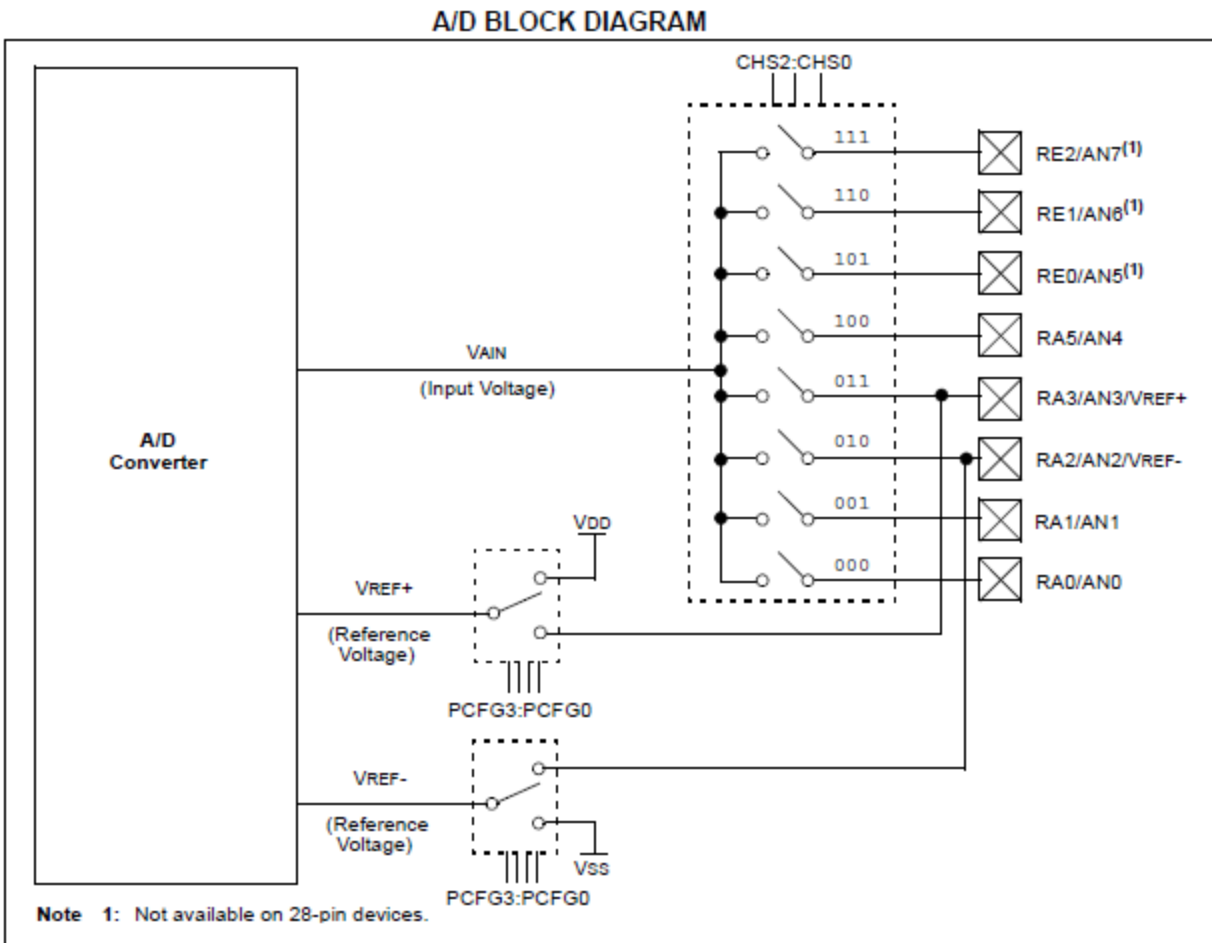
PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O

C/R = # of analog input channels/# of A/D voltage references

A/D Result High Register (ADRESH) & A/D Result Low Register (ADRESL)

The ADRESH: ADRESL registers contain the 10-bit result of the A/D conversion. When the A/D conversion is complete, the result is loaded into this A/D Result register pair, the GO/DONE bit (ADCON0<2>) is cleared and the A/D interrupt flag bit ADIF is set. The block diagram of the A/D module is shown below.



Steps to follow

To do an A/D Conversion, follow these steps:

1. Configure the A/D module:
 - Configure analog pins/voltage reference and digital I/O (ADCON1)
 - Select A/D input channel (ADCON0)
 - Select A/D conversion clock (ADCON0)
 - Turn on A/D module (ADCON0)
2. Configure A/D interrupt (if desired):
 - Clear ADIF bit
 - Set ADIE bit
 - Set PEIE bit
 - Set GIE bit
3. Wait the required acquisition time.
4. Start conversion:
 - Set GO/DONE bit (ADCON0)
5. Wait for A/D conversion to complete by either:

- Polling for the GO/DONE bit to be cleared (interrupts disabled); OR
- Waiting for the A/D interrupt

6. Read A/D Result register pair (ADRESH: ADRESL), clear bit ADIF if required.

7. For the next conversion, go to step 1 or step 2 as required.

Circuit Diagram

LCD:

RS – RC0

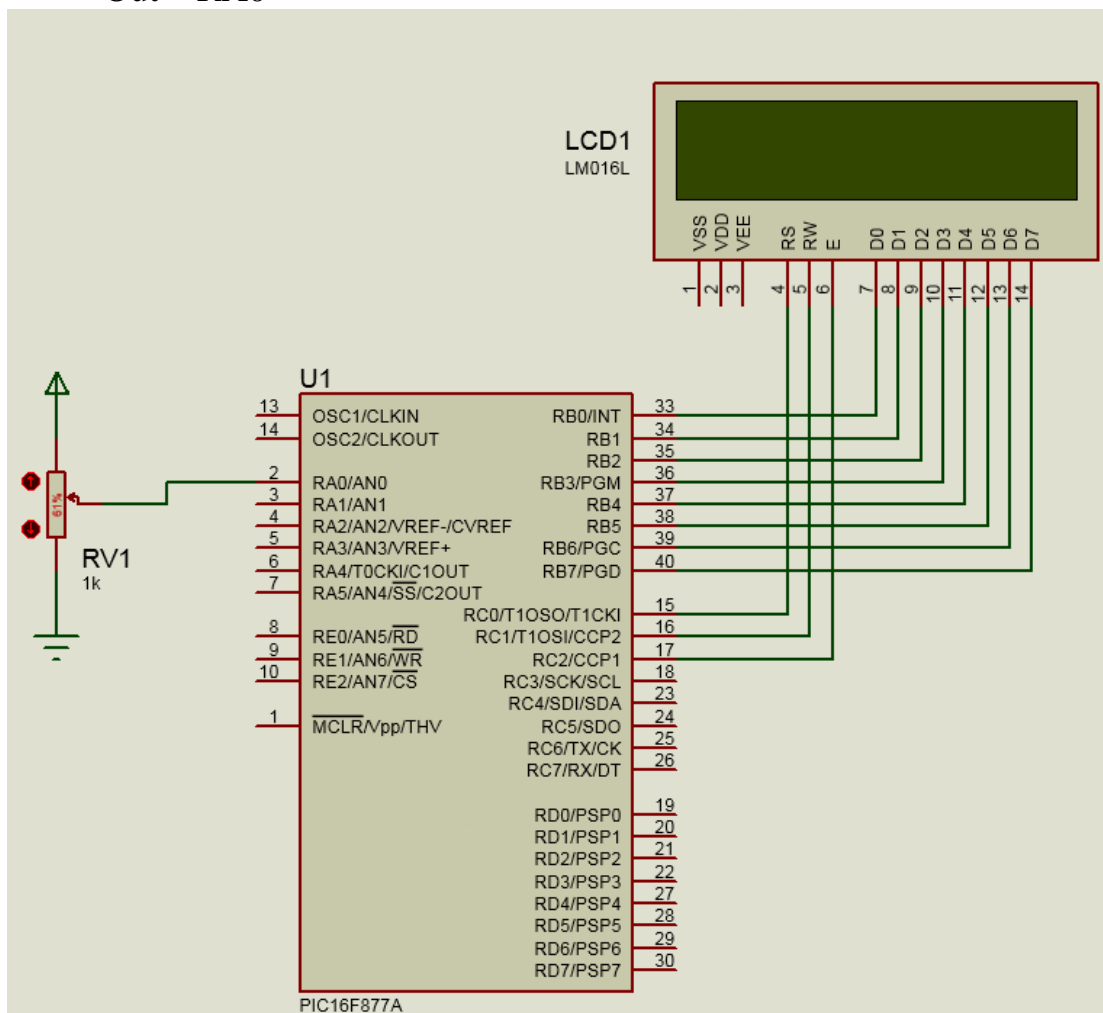
RW – RC1

EN – RC2

Data Lines – Port B

POT:

Out – RA0



Programming

Whenever value changes in Potentiometer it will display in LCD Module. In this pin, you can connect any sensors. You can download the full project here.

```
#include<pic.h>
```

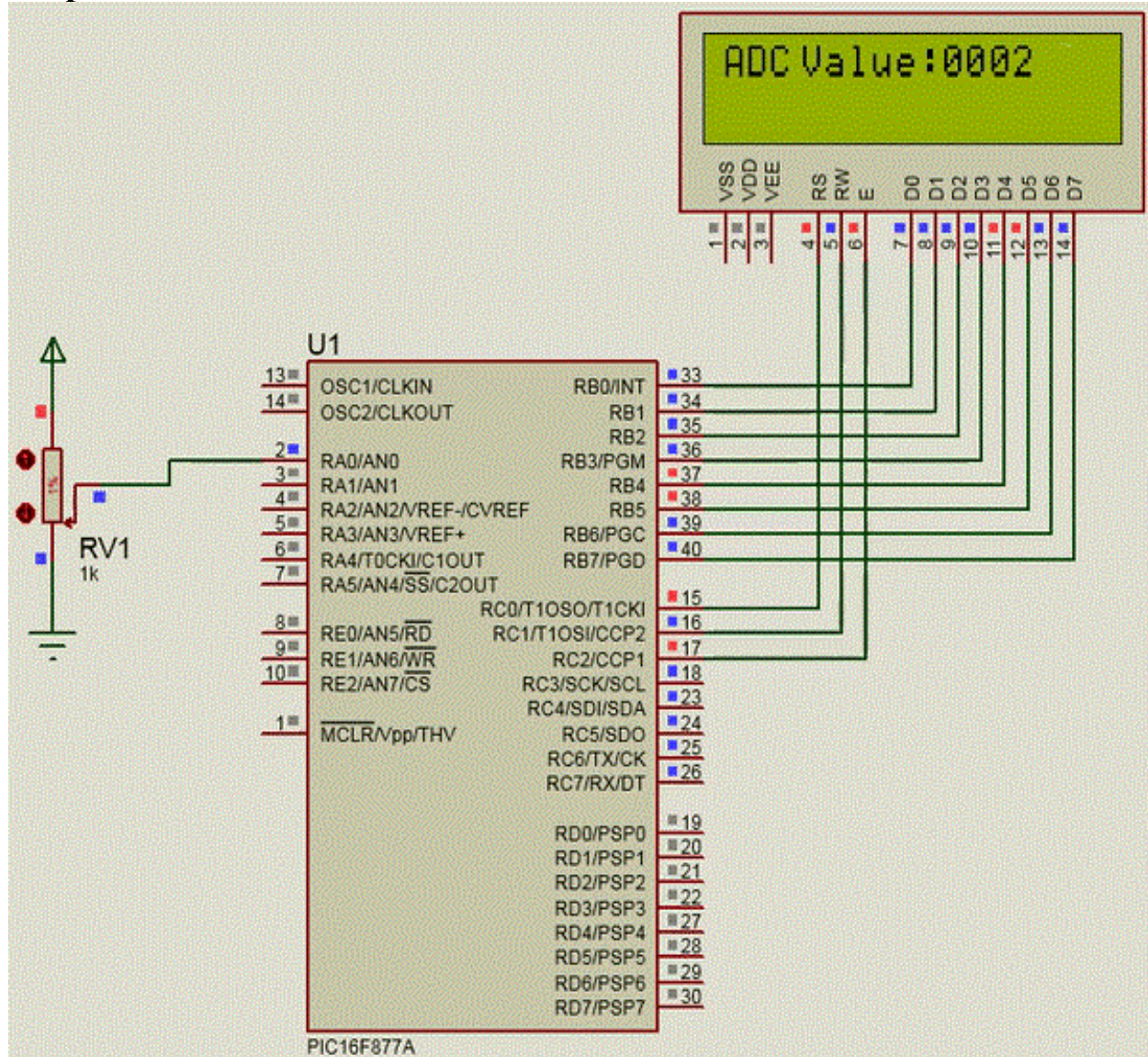
```

#define delay for(i=0;i<=1000;i++)
#define rs RC0
#define rw RC1
#define e RC2 __CONFIG()
void adc();
void lcd_int();
void cmd(unsigned char a);
void dat(unsigned char b);
void show(unsigned char *s);
int i;
void main()
{
    TRISB=TRISC=0;      //Port B and Port C is Output (LCD)
    TRISA0=1;           //RA0 is input (ADC)
    lcd_int();
    show("ADC Value :");
    while(1) {
        cmd(0x8C);
        adc();
    }
}
void lcd_int(){ cmd(0x38); cmd(0x0c); cmd(0x06); cmd(0x80);}
void cmd(unsigned char a){ PORTB=a; rs=0; rw=0; e=1; delay; e=0;}
void dat(unsigned char b){ PORTB=b; rs=1; rw=0; e=1; delay; e=0;}
void show(unsigned char *s){ while(*s) { dat(*s++); }}
void adc()
{
    unsigned int adcval;

    ADCON1=0xc0;        //right justified
    ADCON0=0x85;        //adc on, fosc/64
    while(GO_nDONE);    //wait until conversion is finished
    adcval=((ADRESH<<8)|(ADRESL)); //store the result
    adcval=(adcval/3)-1;
    dat((adcval/1000)+48);
    dat(((adcval/100)%10)+48);
    dat(((adcval/10)%10)+48);
    dat((adcval%10)+48);
}

```


Output – PIC16F877A ADC Tutorial



PIC16F877A- I2C

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In our previous tutorial, we have seen PIC16F877A ADC Tutorial. Now we are going to see PIC16F877A I2C Tutorial.

Prerequisites

Before getting into the I2C Tutorial of PIC16F877A, Please read the below topics....

- I²C Basics
- I²C Advanced
- LCD Interfacing with PIC16F877A

Introduction

I²C is a serial computer bus, which is invented by NXP semiconductors (Philips semiconductors). The I²C bus is used to attach low-speed peripheral integrated circuits to microcontrollers and processors. I²C bus uses two bidirectional open-drain lines such as SDA (serial data line) and SCL (serial clock line) and these are pulled up with resistors. I²C bus permits a master device to start communication with a slave device. Data is interchanged between these two devices. Typical voltages used are +3.3V or +5V although systems with extra voltages are allowed. Nowadays new microcontrollers have inbuilt I²C Registers. PIC16F877A also has separate registers for I2C. Unlike UART, you can connect and communicate to multiple devices using the same I2C bus.

Most of the PIC microcontrollers have built-in **Master Synchronous Serial Port (MSSP)**.

Master Synchronous Serial Port (MSSP)

The Master Synchronous Serial Port (MSSP) module is a serial interface, useful for communicating with other peripheral or microcontroller devices. These peripheral devices may be serial EEPROMs, shift registers, display drivers, A/D converters, etc. The MSSP module can operate in one of two modes:

- Serial Peripheral Interface (SPI)
- Inter-Integrated Circuit (I2C)

The I2C interface supports the following modes in hardware:

- Master mode
- Multi-Master mode
- Slave mode

In this tutorial, we will learn how to operate the MSSP module of the PIC Microcontroller as an I²C master. And EEPROM will act as a slave.

The MSSP module in I2C mode fully implements all master and slave functions (including general call support) and provides interrupts on Start and Stop bits in

hardware to determine a free bus (multi-master function). The MSSP module implements the standard mode specifications, as well as 7-bit and 10-bit addressing. Two pins are used for data transfer:

- Serial clock (SCL) – RC3/SCK/SCL
- Serial data (SDA) – RC4/SDI/SDA

The user must configure these pins as inputs or outputs through the TRISC<4:3> bits.

Registers Used for I2C

The MSSP module has three associated registers. These three registers are used for I2C.

The MSSP module has six registers for I2C operation. These are:

- MSSP Status Register (SSPSTAT)
- MSSP Control Register 1 (SSPCON1)
- MSSP Control Register 2 (SSPCON2)
- Serial Receive/Transmit Buffer Register (SSPBUF)
- MSSP Shift Register (SSPSR) – Not directly accessible
- MSSP Address Register (SSPADD)

SSPSTAT – MSSP Status Register

This register is the status register of the MSSP Module. The lower six bits of the SSPSTAT are read-only. The upper two bits of the SSPSTAT are read/write.

SSPSTAT: MSSP STATUS REGISTER (I²C MODE)

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/A	P	S	R/W	UA	BF
bit 7							bit 0

BF: This is the Buffer Full status bit. In the transmit mode this bit will set when we write data to SSPBUF register and it is cleared when the data is shifted out. In the receive mode, this bit will set when the data or address is received in the SSPBUF register and it is cleared when we read the SSPBUF register.

UA: This is the Update Address bit and is used only in 10-bit address mode. It indicates that the user needs to update the address in the SSPADD register.

R/W: This is the Read/Write bit information. In the slave mode, it indicates the status of the R/W bit during the last address match. In the master mode, 1 indicates that transmit is in progress and vice versa.

S: This bit indicates that a Start bit is detected last and it will be cleared automatically during Reset.

P: As above, this bit indicates that a Stop bit is detected last and it will be cleared automatically during Reset.

D/A: This is the data or addresses indicator bit and it is used only in slave mode. If it is set, the last byte received was data otherwise it will be addressed.

CKE: Setting this bit enables SMBus specific inputs. SMBus is another bus similar to I2C, which is compatible with each other.

SMP: Setting this bit disables slew rate control and vice versa.

SSPCON1 – MSSP Control Register 1

The SSPCON1 register is readable and writable, which is used to control the I2C.

SSPCON1: MSSP CONTROL REGISTER 1 (I²C MODE)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
bit 7							bit 0

SSPM0 ~ SSPM3: These are synchronous serial port mode select bits.

1111 = I2C Slave mode, 10-bit address with Start and Stop bit interrupts enabled

1110 = I2C Slave mode, 7-bit address with Start and Stop bit interrupts enabled

1011 = I2C Firmware Controlled Master mode (Slave Idle)

1000 = I2C Master mode, clock = FOSC/(4 * (SSPADD + 1))

0111 = I2C Slave mode, 10-bit address

0110 = I2C Slave mode, 7-bit address

CKP: This is the SCL clock release control bit. It is used only in slave mode.

Setting this bit releases the clock. If zero, it holds the clock (clock stretch).

SSPEN: This is the synchronous serial port enable bit. Setting this bit enables the serial port.

SSPOV: This is receiving an overflow indicator bit. If this bit is set during receive mode, it indicates that a byte is received while SSPBUF is holding the previous value. And it has no application in transmit mode. We must clear this bit in software.

WCOL: It is the write collision detect bit. If this bit is set during master to transmit mode, it indicates that a write to SSPBUF register was attempted when I2C conditions were not valid for a transmission to be started. And if it is set during a slave to transmit mode, it indicates that the SSPBUF register is written when it is transmitting the previous word. We must clear this bit in software.

SSPCON2 – MSSP Control Register 2

The SSPCON2 register is readable and writable, which is used to control the I2C.

SSPCON2: MSSP CONTROL REGISTER 2 (I²C MODE)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN
bit 7							bit 0

SEN: Start Condition or Stretch Enable bit. In master mode, setting this bit to initiate start condition on SCL & SDA pins and it will be automatically cleared by

the hardware. And in slave mode setting this bit enables clock stretching for both slave receive and slave transmit. If it is cleared in slave mode, clock stretching is enabled only for slave transmit.

RSEN: Repeated start condition enable bit. This bit has application only in master mode. Setting this bit will initiate repeated start condition on both SCL & SDA pins and it will be cleared automatically in hardware.

PEN: Stop condition enable bit. This bit has application only in master mode. Setting this bit will initiate stop condition on both SCL & SDA pins and it will be automatically cleared in hardware.

RCEN: Receive enable bit. This bit also has application only in master mode. Setting this bit enables receive mode for I2C.

ACKEN: Acknowledge sequence enable bit. Setting this bit initiates acknowledge sequence on SCL & SDA lines and it will send ACKDT (see below) bit. This bit will be automatically cleared in hardware. It has application only in master receive mode.

ACKDT: Acknowledge data bit. 1 means not acknowledge and 0 means acknowledge. This value will be transmitted when we set the ACKEN bit (above). This bit has application only in master receive mode.

ACKSTAT: Acknowledge status bit. 1 indicates that acknowledgment was not received from the slave and vice versa. This bit has applications in master transmit mode only.

GCEN: General call enable bit. Setting this bit enables interrupt when a general call address is received in the register SSPSR.

SSPBUF & SSPSR & SSPADD

SSPSR is the shift register used for shifting data in or out. SSPBUF is the buffer register to which data bytes are written to or read from.

SSPADD register holds the slave device address when the SSP is configured in I2C Slave mode. When the SSP is configured in Master mode, the lower seven bits of SSPADD act as the baud rate generator reload value.

In receive operations, SSPSR and SSPBUF together create a double-buffered receiver. When SSPSR receives a complete byte, it is transferred to SSPBUF and the SSPIF interrupt is set.

During transmission, the SSPBUF is not double buffered. A write to SSPBUF will write to both SSPBUF and SSPSR.

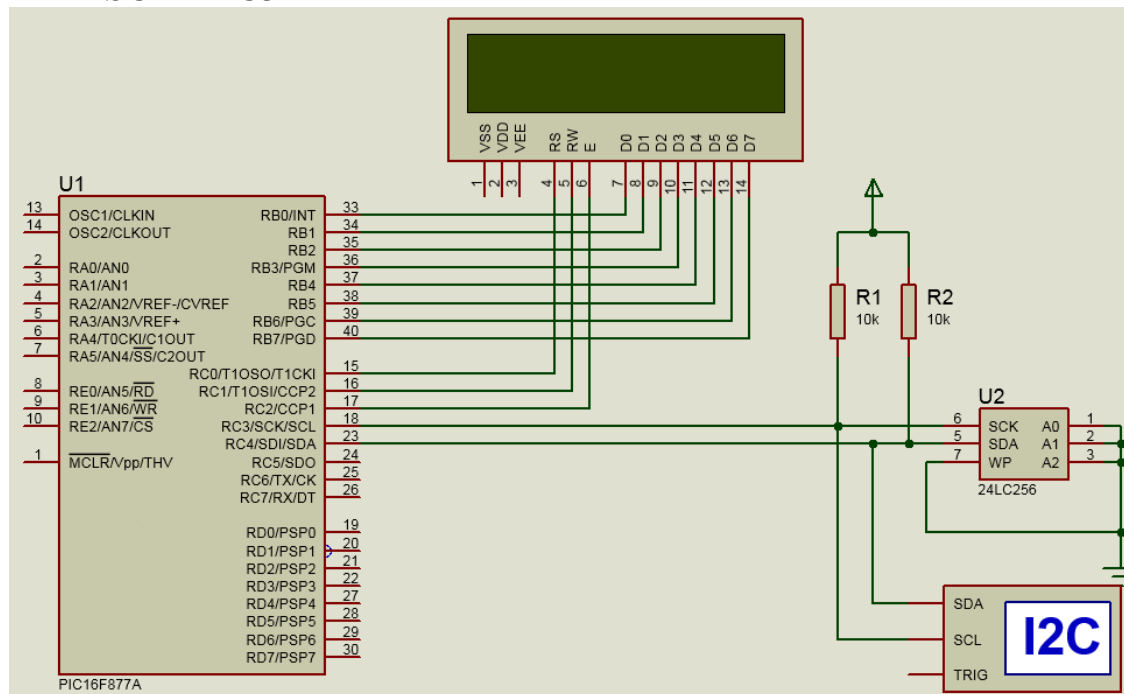
Circuit Diagram

LCD:

- RS – RC0
- RW – RC1
- EN – RC2
- Data Lines – Port B

EEPROM:

- SDA – RC4
- SCL – RC3



Programming – PIC16F877A I2C Tutorial

In this code, I'm writing "Pantech" into EEPROM. Then I'm reading the data byte by byte using I2C. After that Displaying the data received from EEPROM.

Main.c

```
#include<pic.h>
#include"lcd.h"
#include"i2c.h"__CONFIG();
void delay1();
void main()
{
    lcd_init(); i2c_init();
    show(" I2C TuTorial ");cmd(0xc2); show("Writing...");
    i2c_send_byte(0x0000,"Pantech"); delay1();
    cmd(0xc2);show("Reading...");
    delay1(); cmd(0xc2);
```

```

    dat(i2c_read_byte(0x0000));dat(i2c_read_byte(0x0001));
    dat(i2c_read_byte(0x0002));dat(i2c_read_byte(0x0003));
    dat(i2c_read_byte(0x0004));dat(i2c_read_byte(0x0005));
    dat(i2c_read_byte(0x0006));dat(i2c_read_byte(0x0007));
    dat(i2c_read_byte(0x0008));dat(i2c_read_byte(0x0009));
    dat(i2c_read_byte(0x000a));while(1);
}
void delay1()
{
    unsigned int j,k;for(j=0;j<60000;j++) { for(k=0;k<2;k++); }
}

```

I2C.H

```

#define write_cmd 0xA0
#define read_cmd 0xA1
void i2c_init();void i2c_start();
void i2c_stop();void i2c_restart();
void i2c_ack();void i2c_nak();
void waitmssp();void i2c_send(unsigned char dat);
void i2c_send_byte(unsigned char addr,unsigned char *count);
unsigned char i2c_read();unsigned char i2c_read_byte(unsigned char addr);
void i2c_init()
{
    TRISC3=TRISC4=1;
    SSPCON=0x28;           //SSP Module as Master
    SSPADD=((11059200/4)/100)-1;  //Setting Clock Speed, My PCLK =
11.0592MHz
}
void i2c_start()
{
    SEN=1;
    waitmssp();
}
void i2c_stop()
{
    PEN=1;
    waitmssp();
}

void i2c_restart()

```

```

{
    RSEN=1;
    waitmssp();
}
void i2c_ack()
{
    ACKDT=0;
    ACKEN=1;
    waitmssp();
}
void i2c_nak()
{
    ACKDT=1;
    ACKEN=1;
    waitmssp();
}
void waitmssp()
{
    while(!SSPIF);
    SSPIF=0;
}
void i2c_send(unsigned char dat)
{
    L1: SSPBUF=dat;
    waitmssp();
    while(ACKSTAT){i2c_restart;goto L1;}
}
void i2c_send_byte(unsigned char addr,unsigned char *count)
{
    i2c_start();
    i2c_send(write_cmd);
    i2c_send(addr>>8);
    i2c_send(addr);
    while(*count) {
        i2c_send(*count++);
    }
    i2c_stop();
}

unsigned char i2c_read()

```



```

{
    RCEN=1;
    waitmssp();
    return SSPBUF;
}
unsigned char i2c_read_byte(unsigned char addr)
{
    unsigned char rec;
L: i2c_restart();
    SSPBUF=write_cmd;
    waitmssp();
    while(ACKSTAT){ goto L;}
    i2c_send(addr>>8);
    i2c_send(addr);
    i2c_restart();
    i2c_send(read_cmd);
    rec=i2c_read();
    i2c_nak();
    i2c_stop();
    return rec;
}

```

LCD.H

```

#define rs RC0#define rw RC1
#define en RC2#define delay for(i=0;i<1000;i++)
int i;
void lcd_init();
void cmd(unsigned char a);
void dat(unsigned char b);
void show(unsigned char *s);
void lcd_init()
{
    TRISB=TRISC0=TRISC1=TRISC2=0;
    cmd(0x38);
    cmd(0x0c);
    cmd(0x06);
    cmd(0x80);
}

void cmd(unsigned char a)

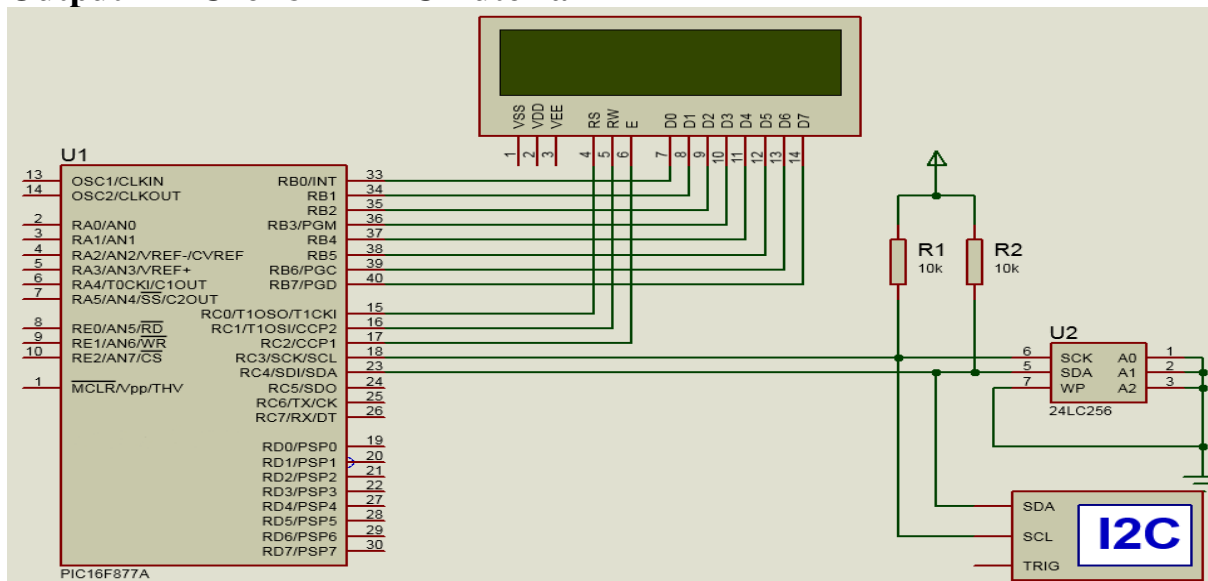
```

```

{
    PORTB=a;
    rs=0;
    rw=0;
    en=1;
    delay;
    en=0;
}
void dat(unsigned char b)
{
    PORTB=b;
    rs=1;
    rw=0;
    en=1;
    delay;
    en=0;
}
void show(unsigned char *s)
{
    while(*s) {
        dat(*s++);
    }
}

```

Output – PIC16F877A I2C Tutorial



PIC16F877A – Interrupt Tutorial

This article is a continuation of the series of tutorials on the PIC16F877A Microcontroller. The aim of this series is to provide easy and practical examples that anyone can understand. In our previous tutorial, we have seen PIC16F877A I2C Tutorial. Now we are going to see PIC16F877A Interrupt Tutorial.

Prerequisites

Before learning PIC16F877A Interrupt Tutorial, we should know the basic interrupts and their functioning. Please refer to these below links.

- [Interrupt Introduction and Its Functions](#)
- [LCD Interfacing with PIC16F877A](#)
- [PIC16F877A USART Tutorial](#)
- [PIC16F877A Timer Tutorial](#)

As the name suggests Interrupts are special events that require immediate attention, it stops a microcontroller/microprocessor from the running task and serves a special task known as Interrupt Service Routine (ISR) or Interrupt Handler.

- External
- Timer 0
- RB Port Change
- Parallel Slave Port Read/Write
- A/D Converter
- USART Receive
- USART Transmit
- Synchronous Serial Port
- CCP1 (Capture, Compare, PWM)
- CCP2 (Capture, Compare, PWM)
- TMR2 to PR2 Match
- Comparator
- EEPROM Write Operation
- Bus Collision

In this tutorial, we will see USART Interrupts, Timer Interrupts, External Interrupts.

Registers Used for Interrupts

- INTCON
- OPTION_REG
- PIE1
- PIR1
- PIE2
- PIR2

INTCON Register

The INTCON register is a readable and writable register, which contains various enable and flag bits for the TMR0 register overflow, RB port change and external RB0/INT pin interrupt.

INTCON REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
bit 7							bit 0

R = Readable bit
'1' = Bit is set

W = Writable bit
'0' = Bit is cleared

U = Unimplemented bit, read as '0'
x = Bit is unknown

- n = Value at POR

GIE: Global Interrupt Enable bit

1-Enables all unmasked interrupts
0-Disables all interrupts

PIE: Peripheral Interrupt Enable bit

1-Enables all unmasked peripheral interrupts
0-Disables all peripheral interrupts

TMR0IE: TMR0 Overflow Interrupt Enable bit

1-Enables the TMR0 interrupt
0-Disables the TMR0 interrupt

INTE: RB0/INT External Interrupt Enable bit

1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

RBIE: RB Port Change Interrupt Enable bit

1 = Enables the RB port change interrupt
 0 = Disables the RB port change interrupt

TMR0IF: TMR0 Overflow Interrupt Flag bit

1-TMR0 register has overflowed (must be cleared in software)
 0-TMR0 register did not overflow

INTF: RB0/INT External Interrupt Flag bit

1 = The RB0/INT external interrupt occurred (must be cleared in software)
 0 = The RB0/INT external interrupt did not occur

RBIF: RB Port Change Interrupt Flag bit

1 = At least one of the RB7:RB4 pins changed state; a mismatch condition will continue to set the bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared (must be cleared in software).
 0 = None of the RB7:RB4 pins have changed state

OPTION_REG

The OPTION_REG Register is a readable and writable register, which contains various control bits to configure the TMR0 Prescaler/WDT Postscaler (single assignable register known also as the Prescaler), the external INT interrupt, TMR0, and the weak pull-ups on PORTB.

OPTION_REG REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

RBPU: PORTB Pull-up Enable bit (This bit is not used for timers)

INTEDG Interrupt Edge Select bit

1 = Interrupt on the rising edge of RB0/INT pin
 0 = Interrupt on the falling edge of RB0/INT pin

T0CS: TMR0 Clock Source Select bit

T0SE: TMR0 Source Edge Select bit

1 = Increment on high-to-low transition on T0CKI pin
 0 = Increment on low-to-high transition on T0CKI pin

PSA: Prescaler Assignment bit

1 = Prescaler is assigned to the WDT
 0 = Prescaler is assigned to the Timer0 module

PS2:PS0: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Note: There is only one Prescaler available which is mutually exclusively shared between the Timer0 module and the Watchdog Timer. A Prescaler assignment for the Timer0 module means that there is no Prescaler for the Watchdog Timer and vice versa. This Prescaler is not accessible but can be configured using **PS2:PS0** bits of **OPTION_REG**.

PIE1 Register

The PIE1 register contains the individual enable bits for the peripheral interrupts.

PIE1 REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE ⁽¹⁾	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

PSPIE: Parallel Slave Port Read/Write Interrupt Enable bit(1)

1 = Enables the PSP read/write interrupt

0 = Disables the PSP read/write interrupt

Note (1): PSPIE is reserved on PIC16F873A/876A devices; always maintain this bit clear.

ADIE: A/D Converter Interrupt Enable bit

1 = Enables the A/D converter interrupt

0 = Disables the A/D converter interrupt

RCIE: USART Receive Interrupt Enable bit

1 = Enables the USART to receive interrupt

0 = Disables the USART receive interrupt

TXIE: USART Transmit Interrupt Enable bit

1 = Enables the USART to transmit interrupt

0 = Disables the USART transmit interrupt

SSPIE: Synchronous Serial Port Interrupt Enable bit

1 = Enables the SSP interrupt

0 = Disables the SSP interrupt

CCP1IE: CCP1 Interrupt Enable bit

1 = Enables the CCP1 interrupt

0 = Disables the CCP1 interrupt

TMR2IE: TMR2 to PR2 Match Interrupt Enable bit

1 = Enables the TMR2 to PR2 match interrupt

0 = Disables the TMR2 to PR2 match interrupt

TMR1IE: TMR1 Overflow Interrupt Enable bit

1 = Enables the TMR1 overflow interrupt

0 = Disables the TMR1 overflow interrupt

PIR1 Register

The PIR1 register contains the individual flag bits for the peripheral interrupts.

Note: Interrupt flag bits are set when an interrupt condition occurs regardless of the state of its corresponding enable bit or the global enable bit, GIE (INTCON<7>). User software should ensure the appropriate interrupt bits are clear prior to enabling an interrupt.

PIR1 REGISTER

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF ⁽¹⁾	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

PSPIF: Parallel Slave Port Read/Write Interrupt Flag bit(1)

1 = A read or a write operation has taken place (must be cleared in software)

0 = No read or write has occurred

Note (1): PSPIF is reserved on PIC16F873A/876A devices; always maintain this bit clear.

ADIF: A/D Converter Interrupt Flag bit

1 = An A/D conversion completed

0 = The A/D conversion is not complete

RCIF: USART Receive Interrupt Flag bit

1 = The USART receive buffer is full

0 = The USART receive buffer is empty

TXIF: USART Transmit Interrupt Flag bit

1 = The USART transmit buffer is empty

0 = The USART transmit buffer is full

SSPIF: Synchronous Serial Port (SSP) Interrupt Flag bit

CCP1IF: CCP1 Interrupt Flag bit

TMR2IF: TMR2 to PR2 Match Interrupt Flag bit

1 = TMR2 to PR2 match occurred (must be cleared in software)

0 = No TMR2 to PR2 match occurred

TMR1IF: TMR1 Overflow Interrupt Flag bit

1 = TMR1 register overflowed (must be cleared in software)

0 = TMR1 register did not overflow

PIE2 Register

The PIE2 register contains the individual enable bits for the CCP2 peripheral interrupt, the SSP bus collision interrupts, EEPROM writes operation interrupt, and the comparator interrupt.

PIE2 REGISTER

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIE	—	EEIE	BCLIE	—	—	CCP2IE
bit 7							bit 0

CMIE: Comparator Interrupt Enable bit

1 = Enables the comparator interrupt

0 = Disable the comparator interrupt

EEIE: EEPROM Write Operation Interrupt Enable bit

1 = Enable EEPROM write interrupt

0 = Disable EEPROM write interrupt

BCLIE: Bus Collision Interrupt Enable bit

1 = Enable bus collision interrupt
 0 = Disable bus collision interrupt

CCP2IE: CCP2 Interrupt Enable bit

1 = Enables the CCP2 interrupt
 0 = Disables the CCP2 interrupt

PIR2 Register

The PIR2 register contains the flag bits for the CCP2 interrupt, the SSP bus collision interrupt, EEPROM write operation interrupt and the comparator interrupt.

PIR2 REGISTER (ADDRESS 0Dh)

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIF	—	EEIF	BCLIF	—	—	CCP2IF
bit 7							bit 0

CMIF: Comparator Interrupt Flag bit

1 = The comparator input has changed (must be cleared in software)
 0 = The comparator input has not changed

EEIF: EEPROM Write Operation Interrupt Flag bit

1 = The write operation completed (must be cleared in software)
 0 = The write operation is not complete or has not been started

BCLIF: Bus Collision Interrupt Flag bit

1 = A bus collision has occurred in the SSP when configured for I2C Master mode
 0 = No bus collision has occurred

CCP2IF: CCP2 Interrupt Flag bit

Serial Interrupt – PIC16F877A Interrupt Tutorial

Circuit Diagram

LCD:

RS – RC0

RW – RC1

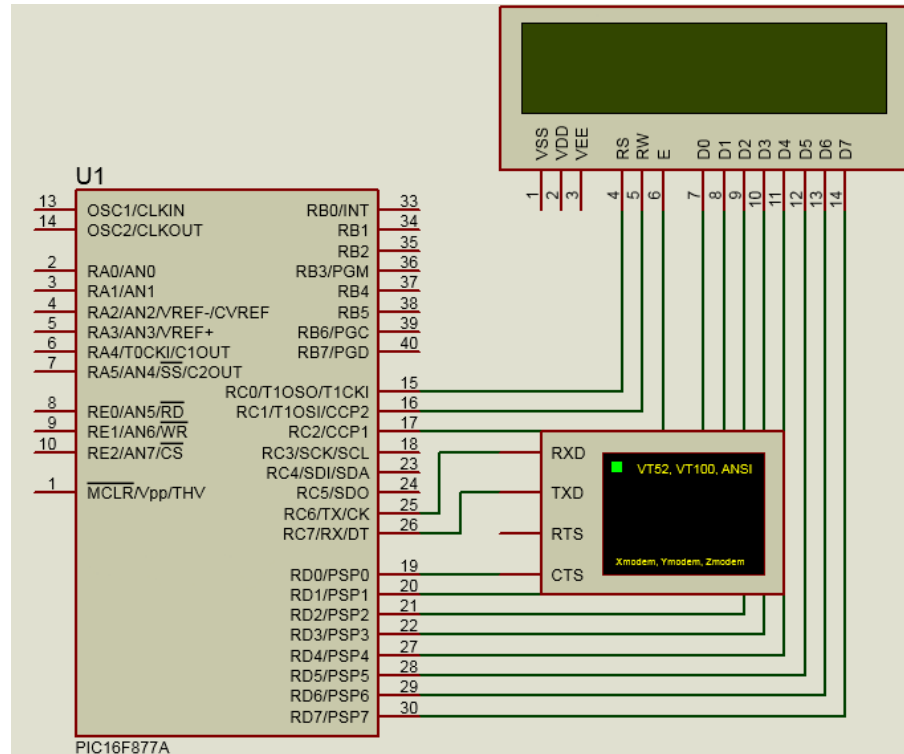
EN – RC2

Data Lines – Port D

UART:

TX – RC6

RX – RC7



Programming

In this program, I have added only the main code. You can find the header files and full project here. My PCLK is 11.0592MHz.

```
#include<pic.h>
#include"serial.h"
#include"lcd.h"

__CONFIG( FOSC_HS & WDTE_OFF & PWRTE_OFF & CP_OFF & BOREN_ON & LVP_OFF &
CPD_OFF & WRT_OFF & DEBUG_OFF);

#define delay for(z=0;z<=50000;z++)

unsigned int z;

void interrupt ser();

void main()
{
    TRISD=0;
    INTCON|=0b11000000;
    PIE1=0b00100000;
    lcd_init();
    serial_init();
    while(1) {
        cmd(0x01);
    }
}

void interrupt ser()
{
    unsigned char a = RCREG;
    tx(a);
    cmd(0x80);
    show("Serial interrupt");
    cmd(0xc0);
    show("  Key : ");
    cmd(0xc8);
    dat(a);
    delay;delay;
}
```

Output

Here I pressed 'O' in UART Terminal. Then it is displayed in LCD Module.

