Experiment #8 (HW#3)
Analog to Digital Conversion and Serial Communication

1. Prerequisites

ENCS 538, C programming language, PICC compiler, Microchip PIC16F877A datasheet, basic knowledge about Java and NetBeans IDE.

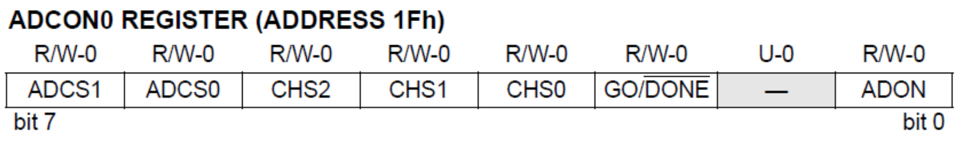
2. Objectives

- Getting familiar with ADC (Analog to Digital Converters).
- Getting familiar with serial communication with PIC16F877A microcontroller.
- Configuring the USART module to send and receive data with PC via standard COM ports.

3. Background

3.1 Analog Data Reading

Within the PIC16F877A, there is an 8-bit analog port that is distributed on two ports that can be either dedicated to work as digital or analog ports. These two ports are the A and the E ports. In order to be able to control these ports to work in either digital or analog modes. There are only four registers that you need to understand to configure the ADC. They are ADCON0, ADCON1, ADRESH and ADRESL. The two most important ones are ADCON0 and ADCON1.ADRESH and ADRESL are just the registers where the ADC stores the result of the conversion.

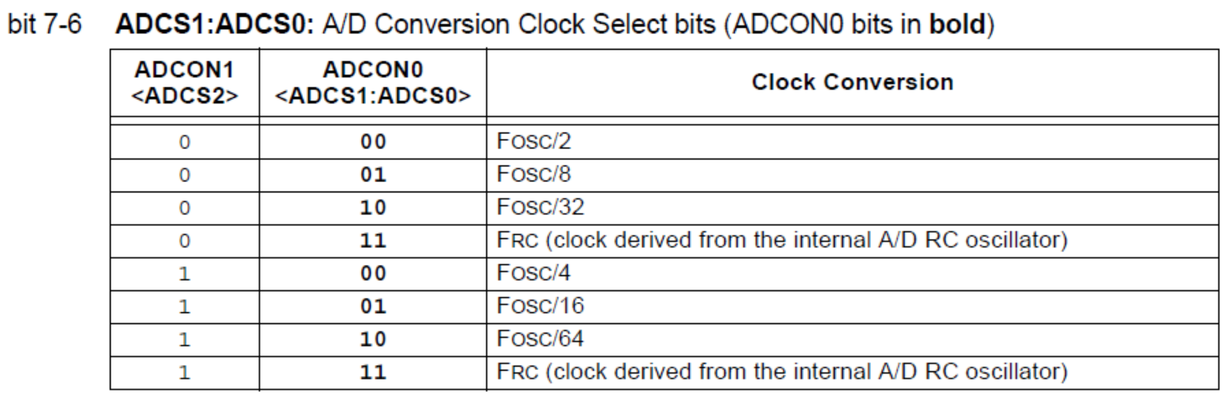


ADCON0 REGISTER (ADDRESS 1Fh)

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/$\overline{\text{DONE}}$	—	ADON

bit 7 bit 0

Figure 1: the ADCON0 register



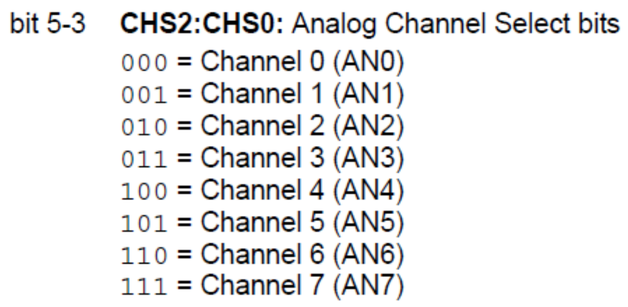
bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

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

Figure 2: the ADCON0 clock selection bits

The user has to select the correct clock conversion. The period must be at least more than 1.6us to obtain an accurate conversion. For example, using a 4MHz crystal oscillator on PIC16F877A. If we select Fosc/2, that is 2MHz, this will result in a 500ns periodic time which is far less than the minimum required (1.6us), Leading to cause an inaccurate conversion.

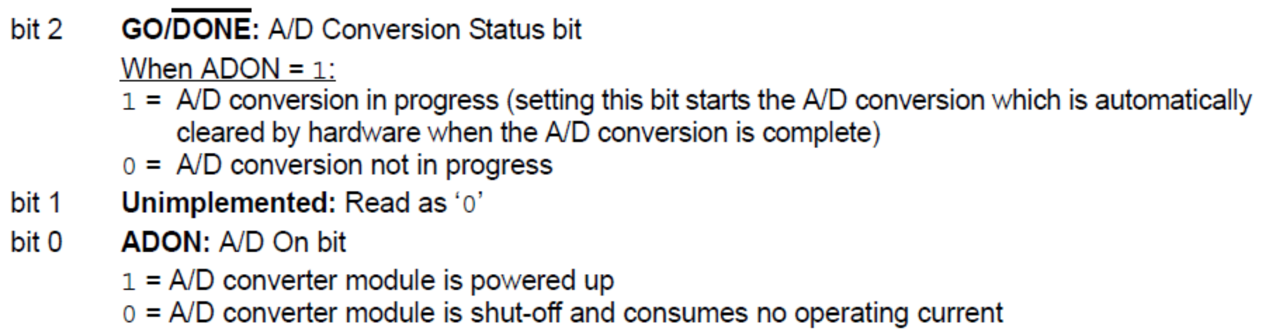
So what do we need to choose in order to solve this? If we choose Fosc/8, that is 0.5MHz which results a periodic time of 2us. That is more than minimum requires (1.6us). So far, our ADCON0 is **01xx xxxx**.



bit 5-3 **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)

Figure 3: the ADCON0 analog channel selection bits

Now that we finished selecting the clock selection bits, we need to select the analog channel were we need to import our reading. The ADC takes one channel only at a time, providing us with a sequential way in reading analog values on AN-port. Assuming that we want to start reading a value on AN2, then we will need to set our 3bits channel-selection inputs to 010. At this stage, our ADCON0 will take the value **0101 0xxx**.



bit 2 **GO/$\overline{\text{DONE}}$:** 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
bit 1 **Unimplemented:** Read as '0'
bit 0 **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

Figure 4: the ADCON0 analog conversion control bits

According to the stated information in Figure 4, the GO/DONE must be set to 1 to start the conversion process for the applied value on channel2. The hardware will automatically set this value back to zero at the end of the conversion making us able to start reading another value.

The unimplemented bit is set as zero, and of course, since we're reading an analog value, we need to set the ADON bit to one. Finally, our ADCON0 register will take the value **0101 0101** at the start of the analog value reading and it will be automatically set to **0101 0001** at the end of it.

ADCON1 REGISTER (ADDRESS 9Fh)

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

bit 7 **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'.

Figure 5: the ADCON1 register

The ADFM bit determines how the result of the ADC is justified. The ADC on the PIC16F877A has 10-bits of resolution, so of course a single register (that has 8 bits) is not enough to contain the 10-bits result. Therefore, two registers are required to store the results. ADRESH and ADRESL (H is the high byte while L is the low byte).

Two registers will allow us to store up to 16 bits, but since there are only 10 bits, we have the flexibility to align it right justified or left justified. Hopefully you will get the picture from the diagram below.

A/D RESULT JUSTIFICATION

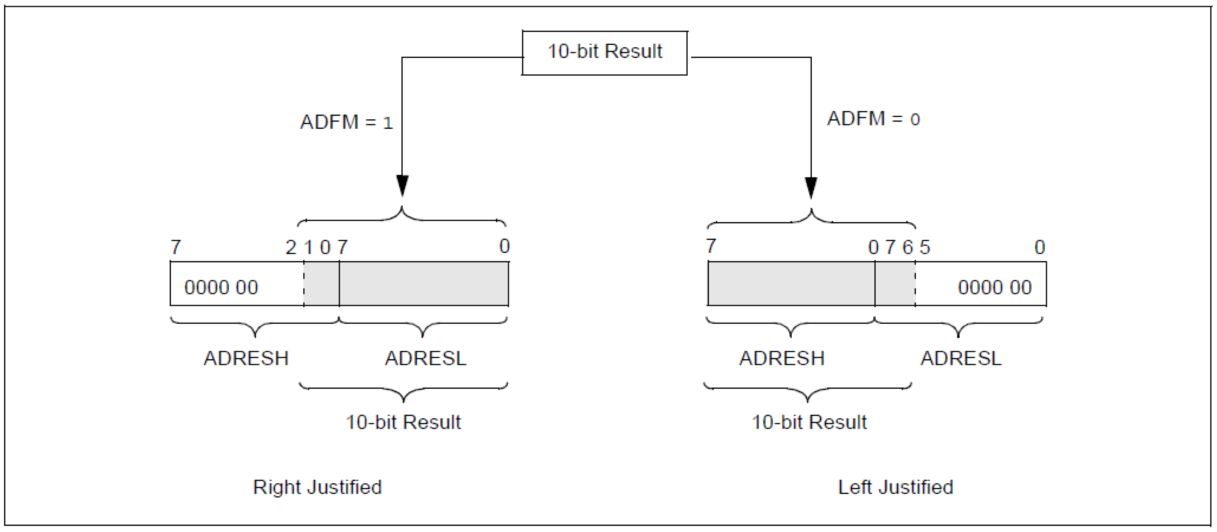
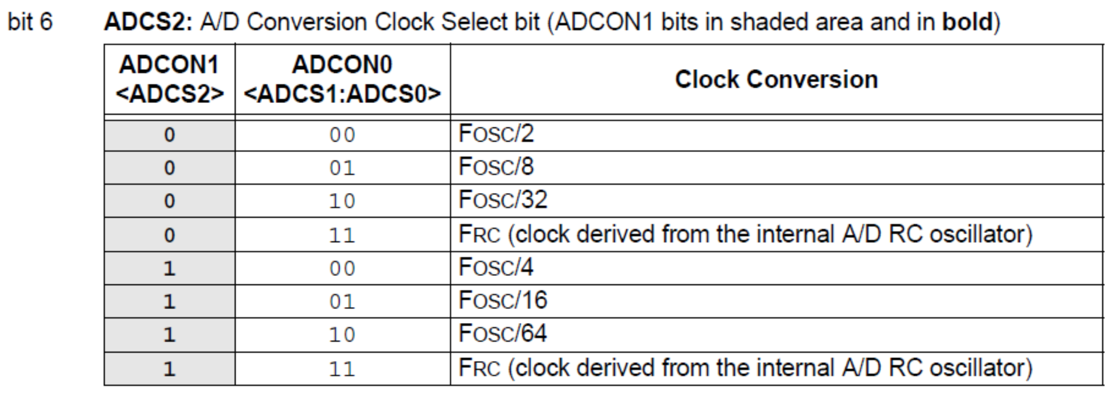


Figure 6: the ADRES register result justification

Storing the result in left justified mode is weird and unusual but it gives the user flexibility. Let's say that our application does not need the 10-bit accuracy, where 8 bits is more than enough. We can just take the result in ADRESH and ignore the remaining two least significant bits in ADRESL. However, you cannot ignore the two highest significant bit because that will cause the result to be inaccurate when choosing right justified mode. Setting a left justification will result the value of ADCON1 to be **0xxx xxxx**.

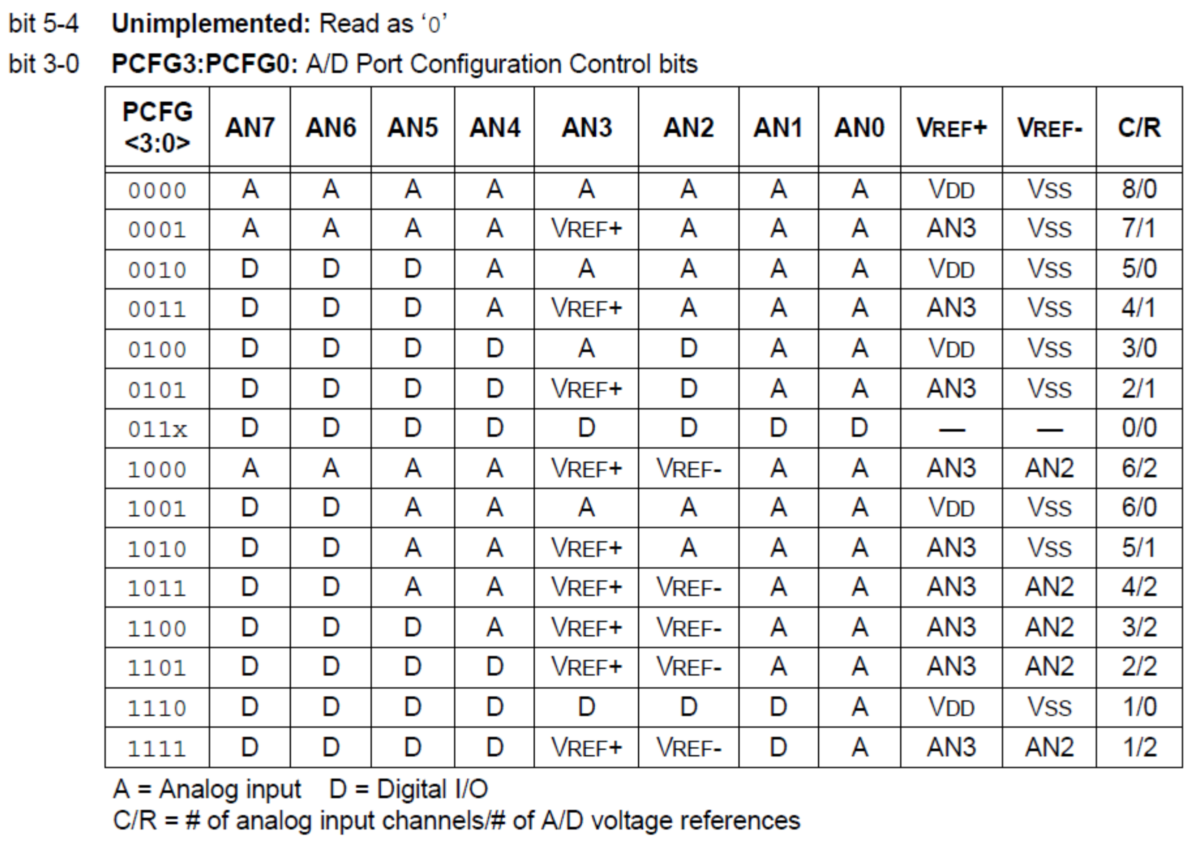


bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

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

Figure 7: the ADCON1 clock selection bit

Next is the ADCS2 bit. Earlier, we calculated that Fosc/8 is adequate, so that we selected it in ADCON0. But for Fosc/8, we also need to set the ADCS2 bit in ADCON1. According clock selection table for ADCON1, the value achieving Fosc/8 for ADCS2 bit alongside the value set in ADCON0 is zero resulting a value of ADCON1 to be **00xx xxxx**.



bit 5-4 **Unimplemented:** Read as '0'

bit 3-0 **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

Figure 8: the ADCON1 AD channel assignment bits

Lastly, the most important part of the ADC configuration is to select the mode for each Analog channel. As shown before, we have Analog Channels from 0 to 7. All these inputs can either be set to analog or digital. Referring to the table above, if we don't need any analog inputs and require more digital pins (let's say for a few LCDs), we can set the PCFG3:0 bits to be 011x. But in this case we do need the Analog inputs, so for simplicity we will set all of them to be in analog mode. Therefore, our final value for ADCON1 is **0000 0000**.

One important thing to note is that we've selected Vdd as the Vref+ and Vss as the Vref- that means that our conversion range is from 0 to 5 Volts. If you need it to be other than that, you can set a custom Vref value by choosing other configurations of PCFG3:0.

3.2 Configuring the Serial Communication

Serial communication is very important for micro devices. By using the serial communication one can communicate with computers (via COM port, USB, etc). Peripheral devices or some ICs (such as EEPROM, A/D converters, etc.).

The process of serial communication can be done by low level coding. You need to have a look at timing specification of the protocol and write a code to meet the requirements. However, most PIC microcontrollers come with built-in serial communication protocols such as USART, I2C, and SPI. You only need to configure it in order to starting transmitting/receiving data. In this experiment, we will use PC's COM port also called as RS-232 (Recommended Standard 232) and built-in USART module for the PIC microcontroller.

The word USART is the acronym of Universal Synchronous Asynchronous Receive Transmit. Standard COM port uses asynchronous receiver and transmitter. In this mode there are Tx (transmit) and Rx (receive) lines. There are two types of transmission **modes**: the full-duplex, where both transmitting and receiving data work simultaneously; and the half-duplex, where only one line (transmitter or receiver) work at a time. USART can be used to allow communication between PIC to PIC or between PIC to a personal computer.

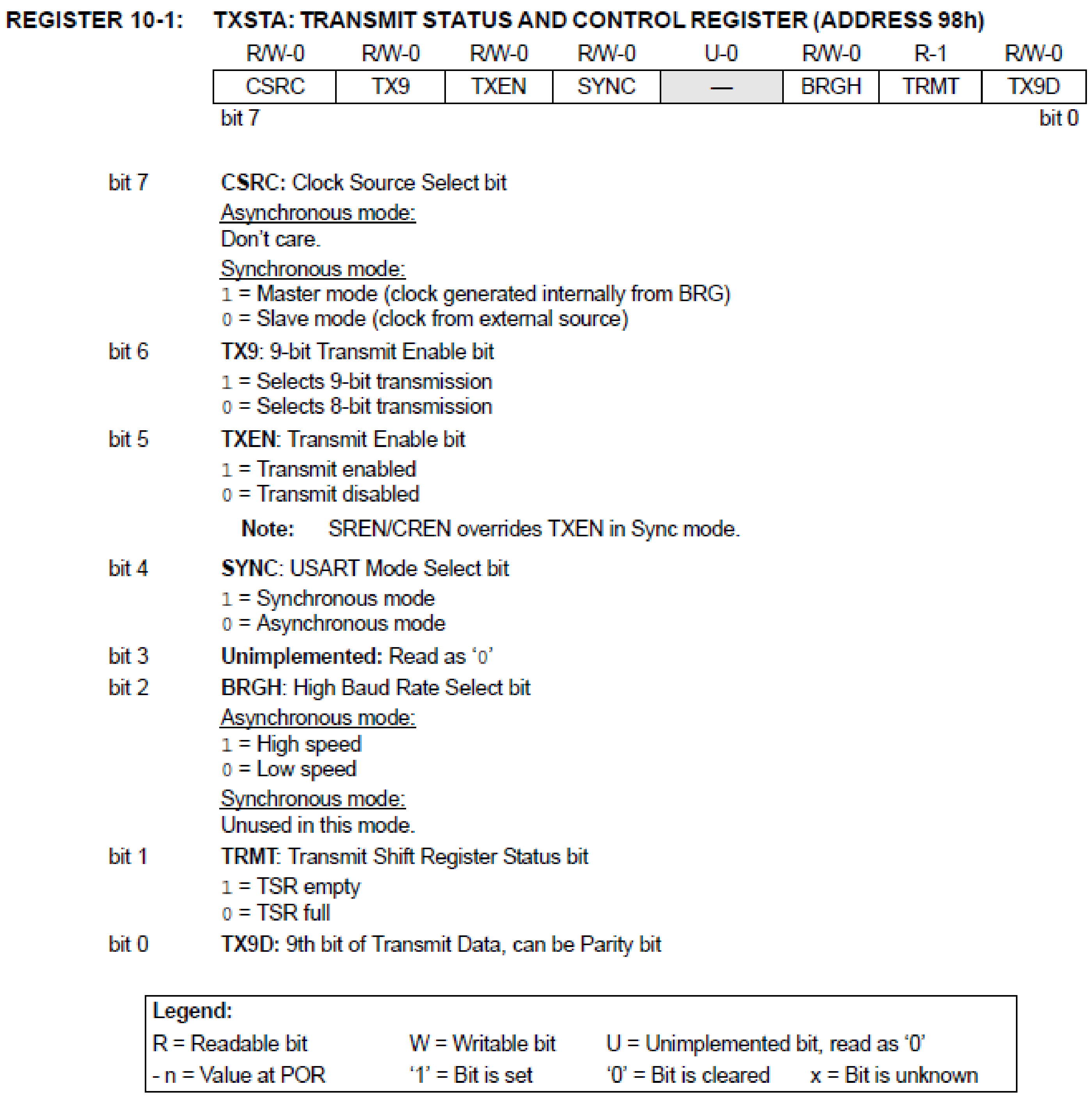
There are two **types** communication based on transmission lines: serial, where data bits are transmitted one bit at a time; and parallel, where a bulk of bits are transmitted simultaneously depending on the number of parallel lines. The trade-off in using serial vs. parallel is measured in hardware consumption vs. transmission speed. If one is interested in speed of transmission, parallel communication will be the right choice. However, if one is interested in hardware usage optimization, serial communication will do the job. Of course, there are variations in this trade-off depending on application itself.

There are two techniques in communication based on timing: the synchronous, where data is sent/received as a stream, bit-wise, with fixed baud rates and clock frequencies at the source and destination; asynchronous, where frequencies on source and destination are not equal where this requires bits to control the data flow in order to achieving consistent transmission of data. Data is sent in frames that are controlled by start and stop bits indicating the starting and ending of each frame. Synchronous technique is faster than the asynchronous since it constantly transmits steams of data without stops.

Within the serial communication module, there are three registers that take in role: the TXSTA transmission status register, the RCXSTA reception status register, and the SPBRG baud generation register.

3.2.1 The TXSTA register

According to the register specifications provided for the TXRSTA register. The transmission status bits function according to Figure 9.



REGISTER 10-1: TXSTA: TRANSMIT STATUS AND CONTROL REGISTER (ADDRESS 98h)

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

bit 7 **CSRC**: Clock Source Select bit
Asynchronous mode:
Don't care.
Synchronous mode:
1 = Master mode (clock generated internally from BRG)
0 = Slave mode (clock from external source)

bit 6 **TX9**: 9-bit Transmit Enable bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN**: Transmit Enable bit
1 = Transmit enabled
0 = Transmit disabled

Note: SREN/CREN overrides TXEN in Sync mode.

bit 4 **SYNC**: USART Mode Select bit
1 = Synchronous mode
0 = Asynchronous mode

bit 3 **Unimplemented:** Read as '0'

bit 2 **BRGH**: High Baud Rate Select bit
Asynchronous mode:
1 = High speed
0 = Low speed
Synchronous mode:
Unused in this mode.

bit 1 **TRMT**: Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

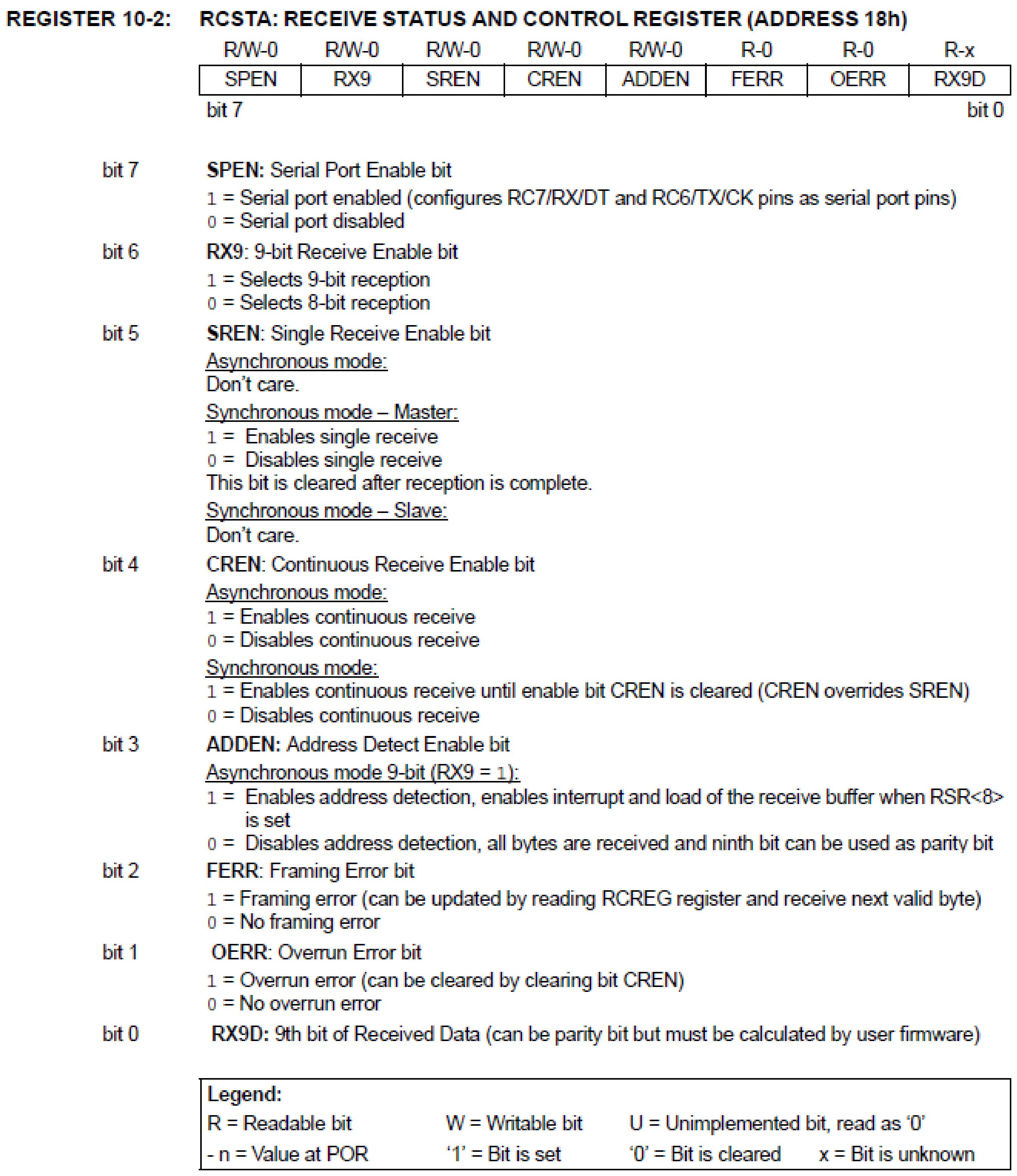
bit 0 **TX9D**: 9th bit of Transmit Data, can be Parity bit

Legend:			
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

Figure 9: the TXSTA register sheet

3.2.2 The RCSTA register

According to the register specifications provided for the TXRCTA register. The transmission status bits function according to Figure 10.



REGISTER 10-2: RCSTA: RECEIVE STATUS AND CONTROL REGISTER (ADDRESS 18h)

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

bit 7 **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

bit 6 RX9: 9-bit Receive Enable bit
1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 SREN: Single Receive Enable bit
Asynchronous mode:
Don't care.
Synchronous mode – Master:
1 = Enables single receive
0 = Disables single receive
This bit is cleared after reception is complete.
Synchronous mode – Slave:
Don't care.

bit 4 CREN: Continuous Receive Enable bit
Asynchronous mode:
1 = Enables continuous receive
0 = Disables continuous receive
Synchronous mode:
1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
0 = Disables continuous receive

bit 3 **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<8> is set
0 = Disables address detection, all bytes are received and ninth bit can be used as parity bit

bit 2 FERR: Framing Error bit
1 = Framing error (can be updated by reading RCREG register and receive next valid byte)
0 = No framing error

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

bit 0 **RX9D:** 9th bit of Received Data (can be parity bit but must be calculated by user firmware)

Legend:			
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

Figure 10: the RCSTA register sheet

3.2.3 The SPBRG register

This register holds the value of the baud rate associated with data transmission. The following presents the formula that both synchronous and asynchronous behaviors in calculating the baud rate depending on the SYNC and BRGH bits.

BAUD RATE FORMULA

SYNC	BRGH = 0 (Low Speed)	BRGH = 1 (High Speed)
0	(Asynchronous) Baud Rate = Fosc/(64 (X + 1))	Baud Rate = Fosc/(16 (X + 1))
1	(Synchronous) Baud Rate = Fosc/(4 (X + 1))	N/A

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

Example - Baud Rate Calculation: Calculate the SPBRG and error rate for an application that uses PIC16F877A with 4MHz external clock oscillator and BRH set to high where a baud percentage of 9600 is required for serial synchronous communication.

9600 = 4000000/ (16(X+1))

SPBRG= ((4000000/9600)-16)/16=25.041,67 [trim floating] ~ 25 (since SPBRG only holds integer values)

Baud Rate = 4000000/ (16(25+1))=9615

Error = (9615-9600)/9600=0.16%

3.2.4 The Transmission and Reception Modules

The following figures show how the bits associated with the three registers previously control hardware parts for the transmission and reception modules.

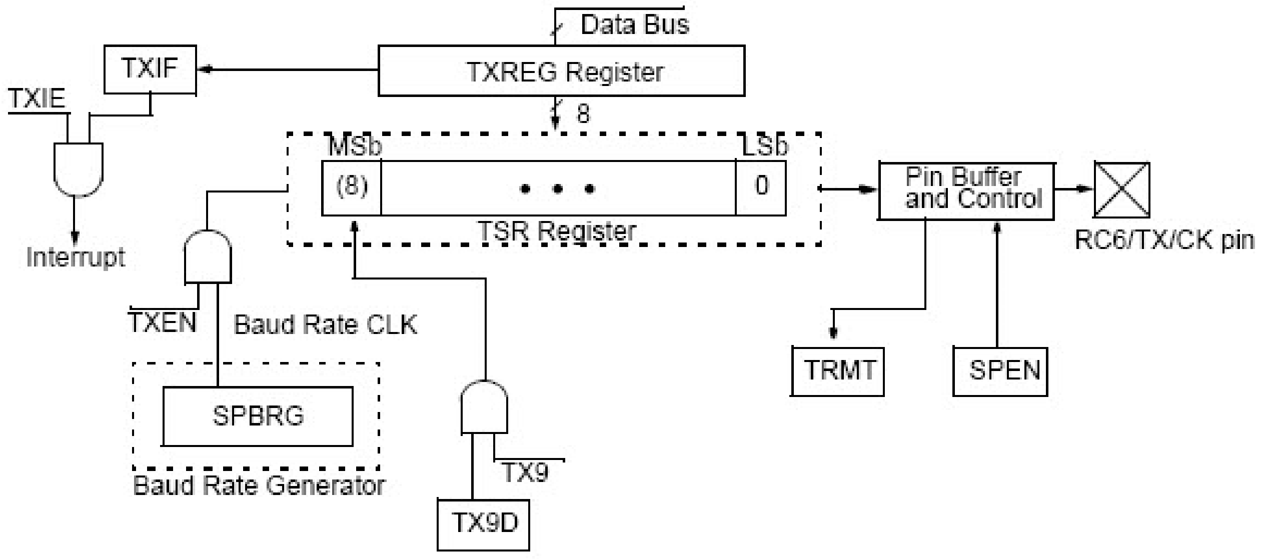


Figure 11: the transmission block diagram

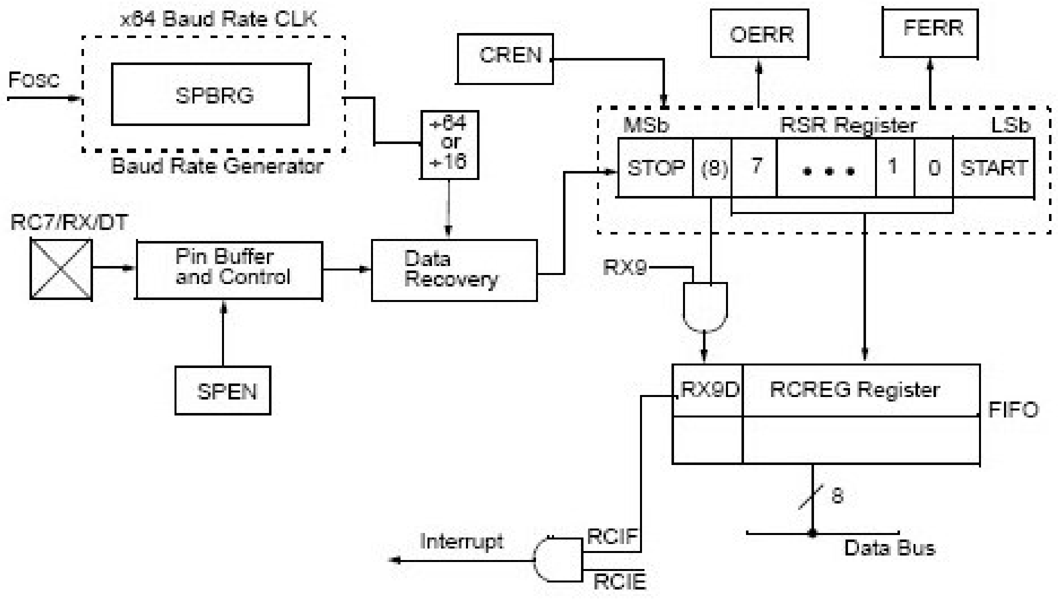


Figure 12: the reception block diagram

3.2.5 RS-232 Interfacing

For transmitting/receiving the information we use - USART. Although the USART successfully achieves PIC-PIC communication where both voltage levels on transmitter and receiver are the same, it still has a problem with PIC-PC communication since the voltage levels of the PIC and the PC are different.

Therefore, in order to establish a successful communication between a PIC and a PC, we have to add another component.

The RS232 uses voltages below -5 Volts to represent a logical level "1", and voltages above 5 Volts to represent a logical level "0". Therefore, to use this protocol we need voltage level conversion. This is possible using the device such as the MAX232. MAX232 is simple component, which operates on 5V.

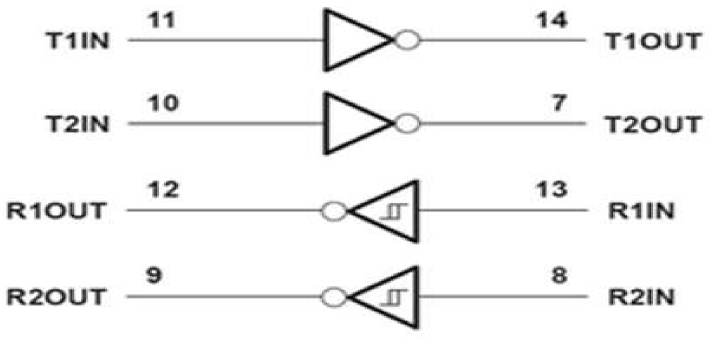


Figure 13: the MAX232 block diagram

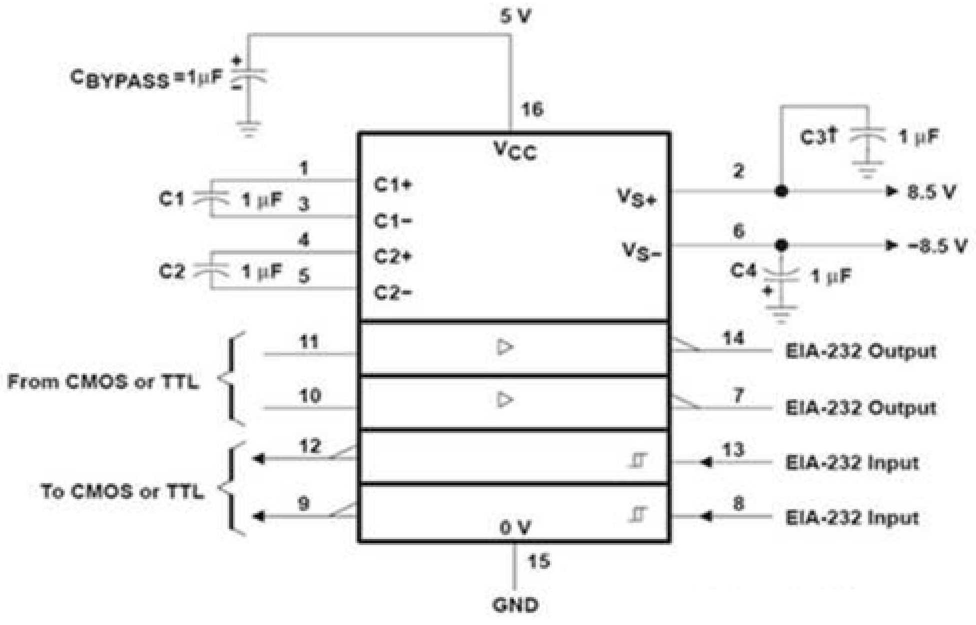


Figure 14: the MAX232 basic connection

The output of the USART (information transmitted to the computer) connects to pin 10 or 11. Levels of information are converting to voltage values that are suitable for RS232 and outputs from pins 7 or 14. From here the information advances to the computer.

The information that is transmitted from the computer connects to the pin 8 or 13 of the device. Here again there is conversion levels, but the opposite way, which will apply to USART. Converted signals are outputs through pin 9 or 12.

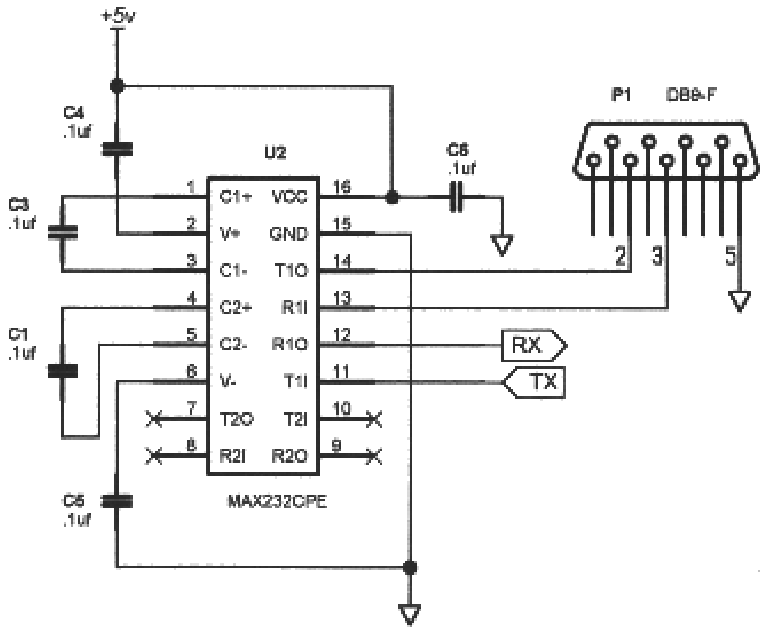


Figure 15: the MAX232-UART connection

4. Procedure

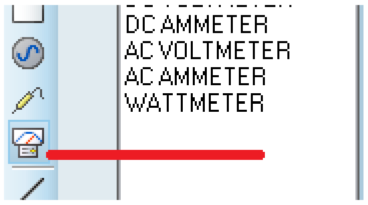
The basic idea of the project is summarized as follows:

1. The **PIC** reads an analog value on pin **AN0**.
2. The **PIC** converts this value from analog to digital using the analog-to-digital internal module.
3. The **PIC** sends the value to the **LCD** to be displayed.
4. The **PIC** decides what character to send to the computer via serial communication based on the analog value read. If it's greater than **2 Volts**, it sends the letter **"g",** else, it sends the letter **"l".**
5. The **computer** receives the character from the **PIC** via a hyper terminal and decides what letter to send in response. If it received a **"g"**, it sends the letter **"e"** (stands for enable). Else, it sends the letter **"d"** (stands for disable).
6. The **PIC** uses the letter received from the computer to apply an action of enabling or disabling a **DC motor** interfaced at some port (B). If the letter was **"e"**, the **PIC** enables the **DC motor**. Else, it disables it.

The experiment is outlined in the following steps:

4.1 Setting up the hardware

1. Open **MPLAB IDE**.
2. Make new project, and add the codes **delay.h**, **delay.c, lcd.h**, **lcd.c**, **string.h**, **string.c** and **USART_main.c** provided in the experiment codes section.
3. Compile the project using **HI-TECH UNVERSAL ANSI COMPILER** for **PIC16F877A**.
4. Open Proteus ISIS.
5. Under components, list the following: **PIC16F877A**, **POT-LN** or **POT-HG**, **RES**, **LM016L**, **CRYSTAL**, **CAP**, **VSOURCE**, **INVERTER**, **L298**, **MOTOR**, **BUTTON**, **MAX232**, and **COMPIM**.
6. Under terminal mode: use the **POWER** and **GND**.
7. Under instruments, pick a **DC VOLTMETER** and **VIRTUAL TERMINAL**. These devices results usually pop-up at simulation time. In case they didn't, you can always display them by clicking on **Debug**, then ticking the devices listed on the bottom of the list. All this happens only at simulation time.



8. Implement circuit provided in Figure 16 (for more clarity, check the attached image).
9. Set the values according to the Figure 16.
10. Double click on the **VIRTUAL TERMINAL**, set the **baud rate** to **9600**.

11. Double click on the **COMPIM**, set the port to **COM7**, **baud rates** to **9600**, **1 stop bit**, **8 physical data bits** and **no parity bits**.
12. Load the generated **HEX** onto the micro-controller. Set its external frequency to **4MHz**.
13. Click on **"play"** and observe the outputs you get. Again, in case the terminal output didn't pop-up at simulation time. During simulation, click on **Debug** then **Virtual Terminal**.
14. Stop the simulation and move on to the next part.

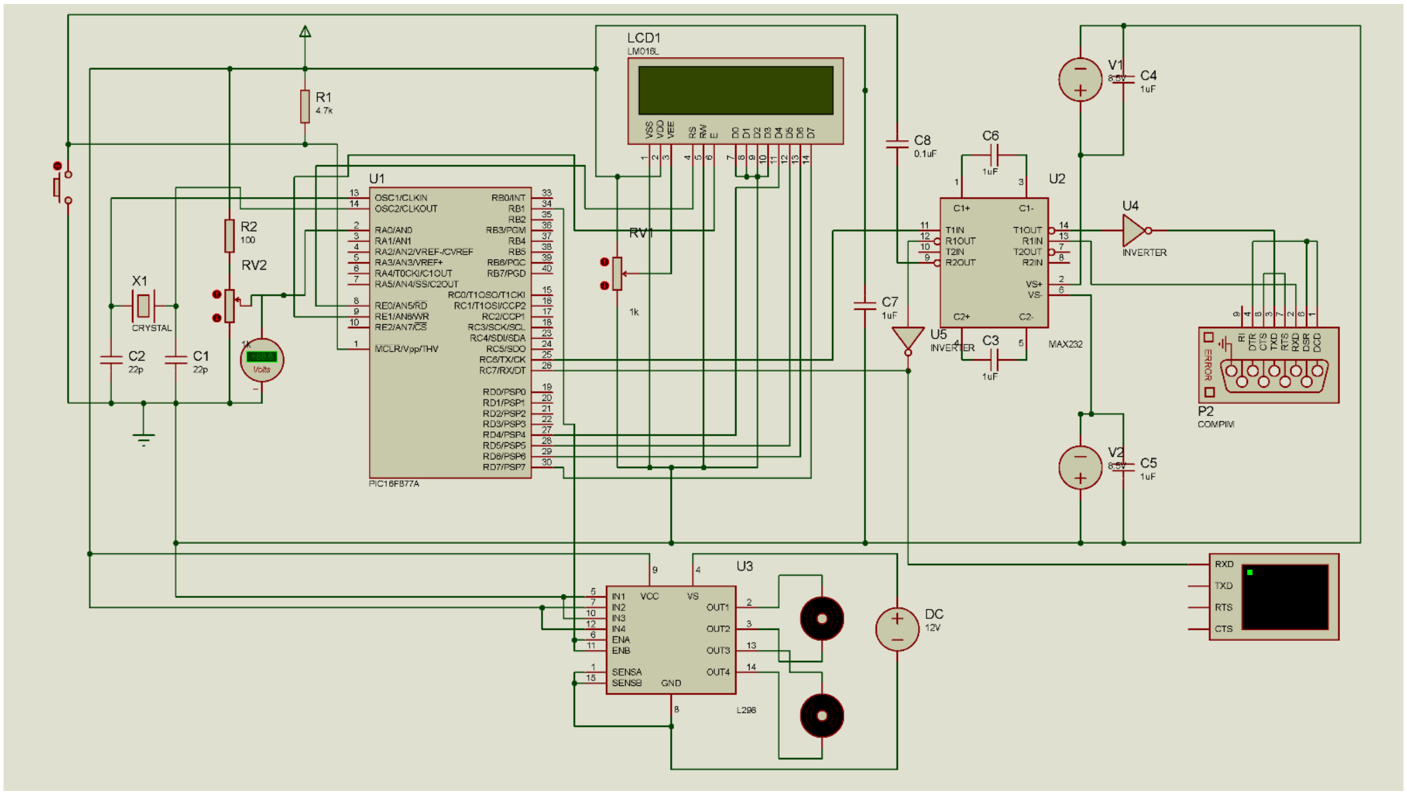


Figure 16: Hardware schematic

4.2 Setting up the Java serial communication library

1. Extract the provided **RAR** file that contains the **DLL** and **JAR** files for the serial communication.
2. Pick the folder that fits with your platform (64 or 32 bit machines).
3. Copy **"rxtxSerial.dll"** into "C:\Program Files\Java\jdk1.7.0_80\jre\bin".
4. Copy **"RXTXComm.jar"** into "C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext".

4.3 Setting up the Java communication program

1. Make a new **Java NetBeans** project.
2. Name it **"TwoWaySerialComm"**.
3. Include a main class within it.
4. Finish.
5. Copy **"RXTXComm.jar"** into **"lib"** folder within the project directory.
6. In case that the library still wasn't added, you can add it by clicking on Libraries within the project navigator then **"Add JAR/Folder…"**.

7. Compile the project.
8. Most probably that you will encounter an exception throw since **COM7** (where the program is assigned to communicate through) is still unavailable.
9. Ignore the warning about driver version mismatch at the run-time.
10. Stop the compilation and move on the next part.

4.4 Setting up a virtual connection using VSPE

This program will be used to establish virtual **COM** connection in a way that makes the communication between the Java program and the **Proteus ISIS** possible. This will provide a real-time simulation between the connected hardware and the software programmed on the computer.

1. Open **VSPE**.
2. Click on **"Create New Device"** icon as in Figure 17.

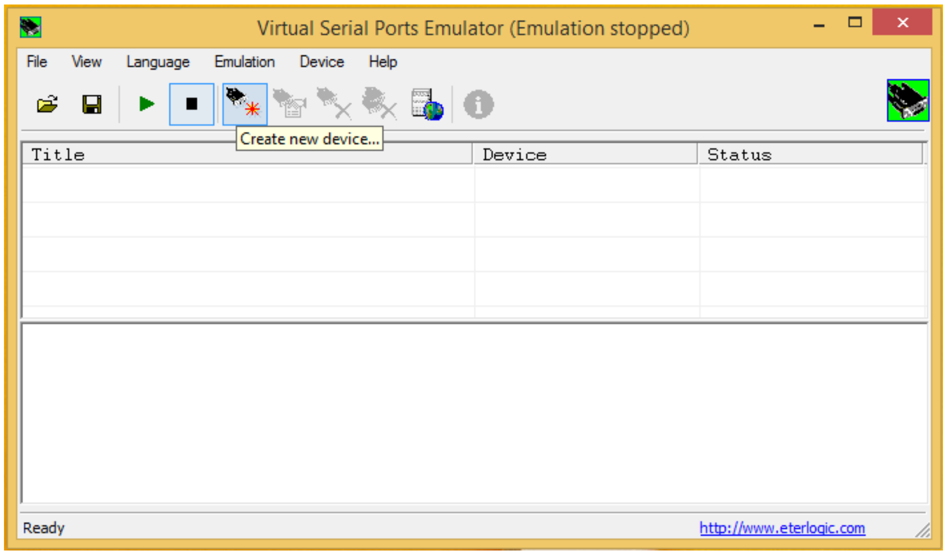
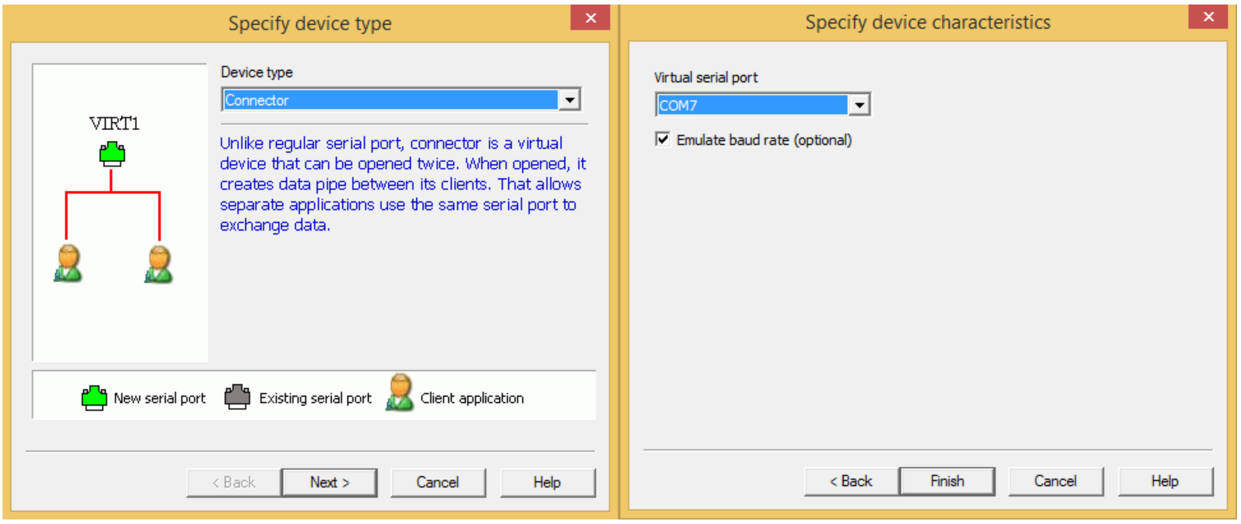
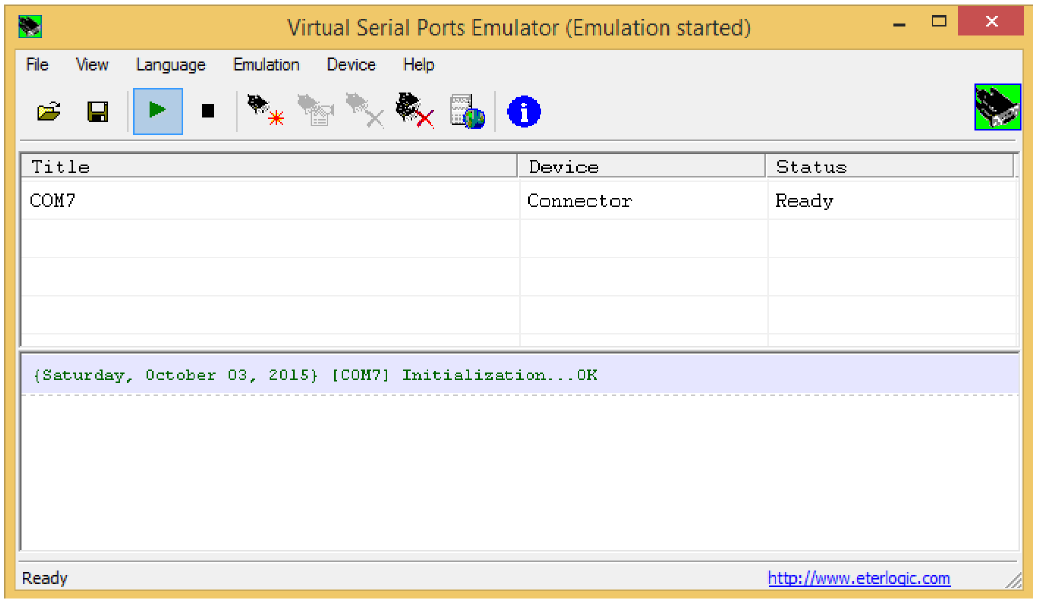


Figure 17: VSPE GUI



Unlike regular serial port, connector is a virtual device that can be opened twice. When opened, it creates data pipe between its clients. That allows separate applications use the same serial port to exchange data.

Figure 18: Setting up a virtual connection



Title	Device	Status
COM7	Connector	Ready

```
{Saturday, October 03, 2015} [COM7] Initialization...OK
```

Figure 19: Emulating connection on COM7

3. You can choose a pair of devices where you assign the terminal within Proteus a different port from that programmed in the Java application e.g. COM2 and COM3. For more simplicity, we will choose a device of type **"Connector",** where the two virtual devices that intend to communicate share the same **COMx** port. Set this port to **COM7** and click on **"Emulate"** as shown in Figure 19.

4.5 Integrating the project

1. Now that everything is ready for the integration, go back to **Proteus ISIS** and simulate the project.
2. Go back to **NetBeans IDE** and compile/run the program.
3. Within the console in NetBeans, you must see messages being sent and received. Each message sent can be a character of **"e"** or **"d"**. Each message received can be **"g"** or **"l".**
4. Get back to **Proteus ISIS**, adjust the **potentiometer** to have a values above **2 Volts** once, wait for the response from the computer and observe the changes.
5. Now try setting the value of the potentiometer to some value bellow **2 Volts**, wait for response time, and observe the outputs you get.
6. The behavior of the system must be exactly as that described in previous sections.

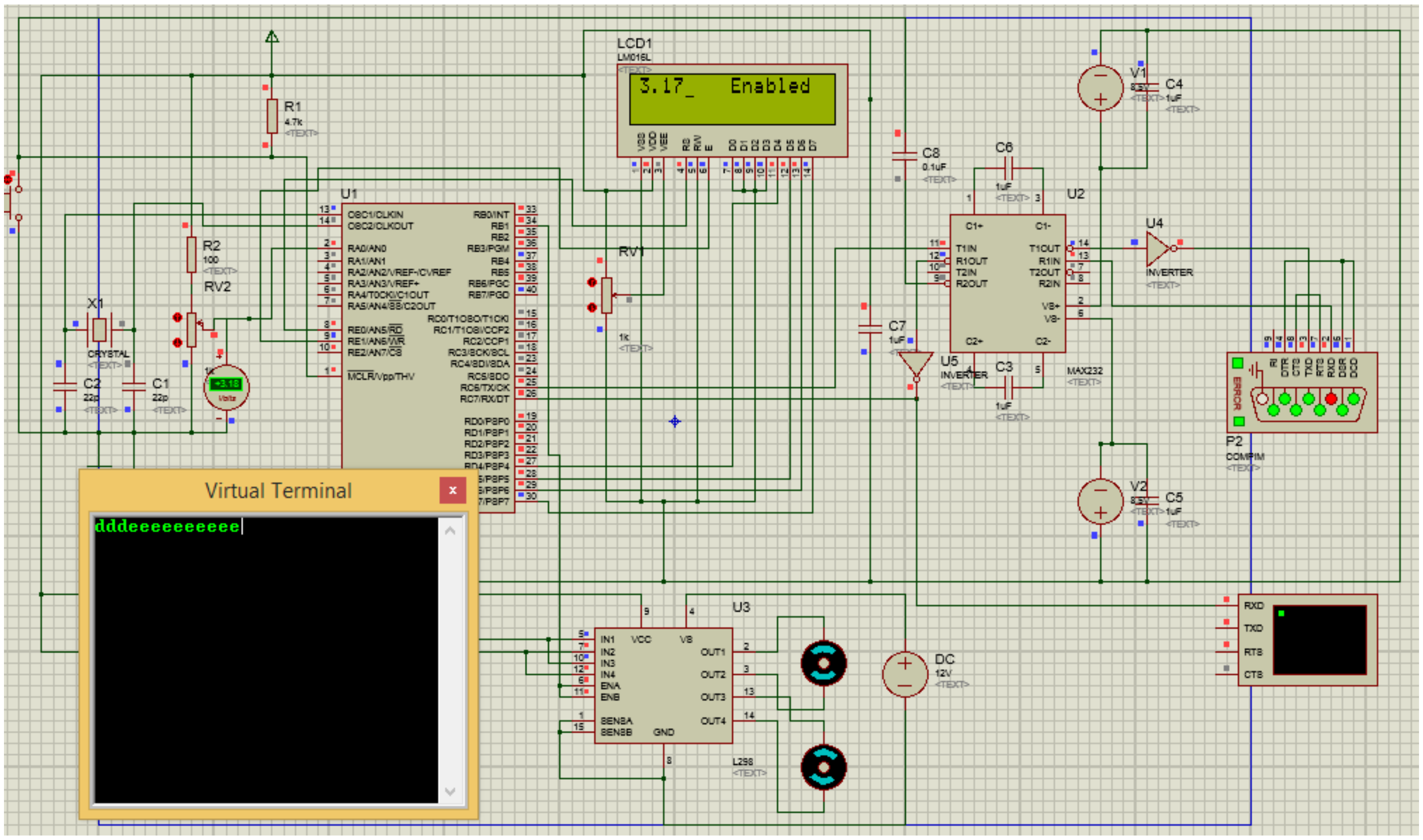
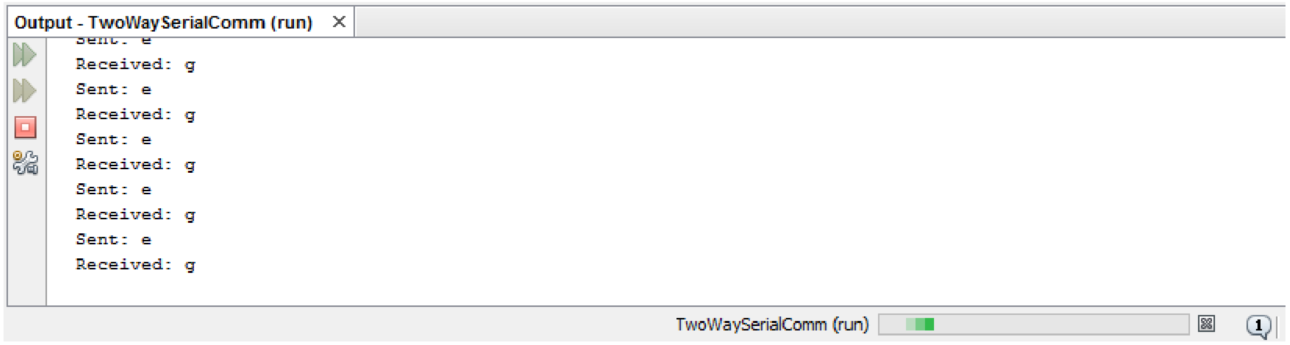


Figure 20: Hardware output



```
Sent: e
Received: g
Sent: e
Received: g
Sent: e
Received: g
Sent: e
Received: g
Sent: e
Received: g
```

Figure 21: Software output

5. Experiment Codes

Delay.h

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

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

      or
      picc -DXTAL_FREQ=100KHZ

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

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

 */

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

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

#if   XTAL_FREQ >= 12MHZ

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

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

extern void DelayMs(unsigned char);
```

Delay.c

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

#include "delay.h"

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

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

Lcd.h

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

Lcd.c

```
/*
*     the list of included files contains:
*     pic.h since we're going to use it with our pic microcontroller, duah
*     ldc.h which contains all the prototypes of the functions used for the
lcd
*     delay.h which contains the prototype of the melli-second delay and
the implementation
*     of the micro-second delay (which is used in the melli-second
implementation, so it has to be included)
*     delay.c contains the implementation of the delay_ms function, so it
has to be here
*/

#include <pic.h>
```

```
#include "lcd.h"
#include "delay.h"
#define LCD_STROBE ((RE1 = 1),(RE1=0))     //* The E bit on the lcd, where
it tells the lcd that we're writing data to it when it's set to 1*/
/*
*      the following write functions take a character inpute or 8 bits where
it takes the higher 4 bits and passes their values to the D port of the pic
*      then it shifts the character by 4 bits to the left in order to take
the values of the lower 4 bits and put them on the used D port bits. The
strobe
*      is used indicate that the LCD is receving data so that the values are
guaranteed to be passed in order and without interference
*
*      As for the RS bit which is connected to RE0. It is used to tell the
LCD to accept the character as a command when it's set to 0, or as a
character to
*      be displayed on the screen when it's set to 1
*/
void lcd_write(unsigned char c)
{
PORTD = (PORTD & 0x0F) | (c);
LCD_STROBE;
PORTD = (PORTD & 0x0F) | (c << 4);
LCD_STROBE;
DelayUs(40);
}
void lcd_clear(void)
{
RE0 = 0;
lcd_write(0x1);
DelayMs(2);
}
void lcd_puts(const char * s)
{
RE0 = 1;
while(*s)
lcd_write(*s++);
}
void lcd_putch(char c)
{
RE0 = 1;
PORTD = (PORTD & 0x0F) | (c);
LCD_STROBE;
PORTD = (PORTD & 0x0F) | (c << 4);
LCD_STROBE;
DelayUs(40);
}
void lcd_goto(unsigned char pos)
{
RE0 = 0;
lcd_write(0x80+pos);
}
void lcd_init(void)
{
RE0 = 0;
DelayMs(15); // power on delay
PORTD = (0x3 << 4);
LCD_STROBE;
DelayMs(5);
LCD_STROBE;
DelayUs(100);
LCD_STROBE;
DelayMs(5);
PORTD = (0x2 << 4);
LCD_STROBE;
DelayUs(40);
lcd_write(0x28); // 4 bit mode, 1/16 duty, 5x8 font
lcd_write(0x08); // display off
lcd_write(0x0F); // display on, blink curson on
lcd_write(0x06); // entry mode
}
```

String.h

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

String.c

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

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

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

    // If number of digits required is more, then
    // add 0s at the beginning
    while (i < d)
```

```
        str[i++] = '0';

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

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

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

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

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

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

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

USART_main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <pic.h>
#include "lcd.h"
#include "delay.h"
#include "string.h"
#define BAUD 9600 //Define baudrate
#define FOSC 4000000L //crystal OSC
#define DIVIDER ((int)(FOSC/(16UL * BAUD) -1)) // calculate baud rate
generator value
/*
*      since we're using 4MHz oscillator, we need to set the oscillator bit
to HS
*      the following are needed to make the LCD get to work:
*      BODEN, WDTDIS and WRTEN
*/

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

void putchOnSerial(unsigned char byte);
void pause(int d);
void init_a2d(void);
float read_a2d(unsigned char channel);
float x;
char str[10];
unsigned int send=1;

void main(void){
      unsigned char rec;
      TRISD = 0; /* to transmit characters to the LCD */
      TRISE = 0; /* to control the LCD */
      TRISB = 0; /* to control the motor */
      ADCON1 = 3;//set PORT A to Digital mode, Port E to digital
      TRISA = 0xFF; /* to take analog values */
      TRISC7=1;  /* set RC7 to input (RX) to receive bits */
      TRISC6=0;  /* set RC6 to output (TX) ro send bits */
      pause(1); //sleep for 1 sec

      /* Loading screen */
      lcd_init(); /* Initialize the LCD */
      lcd_clear(); /* in case of a reset */
      lcd_puts("Starting...");
      DelayMs(1000);

      nRBPU = 0; /* enable pullup resistors */

      SPBRG = DIVIDER; /* set baud rate generator value in SPBRG register
*/
      RCSTA = 0x90; /* set - Receive Status and Control Register */
      TXSTA =0x24; /* set Transmit Status and Control Register */

      init_a2d();  /* Initialize the A2D module */
      DelayMs(500);

      lcd_clear();
      GIE=0;  /* Disable global interrupts */
      while ( 1 ) {

            if(send==1) {
                  DelayMs(100); /* increase to 100 in case any problems
occured during the implementation */
                  lcd_goto(0);

                  /* Read analog value on AN0, adjust by coeficient to
obtain actual value */
                  x= read_a2d(0);
                  x = x/51.0; /* fraction to reflect actual analog value
(since it reaches within the range of the analog register */

                  ftoa(x, str, 2); /* Convert value to string with 2 digits
after the point */

                  lcd_puts(str); /* display value on LCD */

                  if(x>2.0){
                        putchOnSerial('g');
                        DelayMs(100);
                  }
                  else {putchOnSerial('l');
                        DelayMs(100);
                  }
                  send=0;
            }
            else {
                  rec = getch();
                  DelayMs(100);

                  if(rec!=NULL) { //check the PC response is zero or one to
rotate the motor
                        if(rec=='d') {
                              RB1=0;
                              lcd_goto(8);
                              lcd_puts("Disabled");
                        }

                        else if(rec=='e') {
                              RB1=1;
                              lcd_goto(8);
                              lcd_puts("Enabled ");
                        }
                  } /* end of if not null */
                  send=1;
            } /* end of else */
      }

}

void putchOnSerial(unsigned char byte) {
      /* output one byte */
      while(!TXIF) /* set when register is empty */
            continue;
      TXREG = byte;
}
unsigned char getch() {
      /* retrieve one byte */
      if(RCIF)
            return RCREG;
      else
            return NULL;
}

void init_a2d(void){
      ADCON0 = 0x41; // select Fosc/8
      ADCON1 = 0x0E; // select left justify result. A/D port configuration
0
      ADON=1; // turn on the A2D conversion module
      }

/* Return an 8 bit result */
float read_a2d(unsigned char channel){
      channel &=0x07; // truncate channel to 3 bits
      ADCON0 = 0x41; // select Fosc/8
      ADCON1 = 0x0E; // select left justify result. A/D port configuration
0
      DelayMs(10);
      GO_nDONE = 1;
      ADCON0 |=(channel<<3); // apply the new channel select
      while(GO_nDONE)
            continue;
      return( (float) ADRESH); // return 8 MSB of the result
      }
```

```
void pause(int d) {
      int i,j;
      for(i=0;i<4;i++)
            for(j=0;j<d;j++)
                  DelayMs(255);
      }
```

TwoWaySerialComm.java

```
package twowayserialcomm;

import gnu.io.CommPort;
import gnu.io.CommPortIdentifier;
import gnu.io.SerialPort;
import java.io.InputStream;
import java.io.OutputStream;

public class TwoWaySerialComm {
      public TwoWaySerialComm() {
      super();
      }

      void connect ( String portName ) throws Exception {
                boolean send=false;
            CommPortIdentifier portIdentifier =
CommPortIdentifier.getPortIdentifier(portName);
            if ( portIdentifier.isCurrentlyOwned() ) {
                  System.out.println("Error: Port is currently in use");
            }
            else { //outer else
            CommPort commPort =
portIdentifier.open(this.getClass().getName(),2000);
            if ( commPort instanceof SerialPort ) { //outer if
                  SerialPort serialPort = (SerialPort) commPort;

      serialPort.setSerialPortParams(9600,SerialPort.DATABITS_8,SerialPort.
STOPBITS_1,SerialPort.PARITY_NONE);
                  InputStream in = serialPort.getInputStream();
                  OutputStream out = serialPort.getOutputStream();
                   byte[] buffer = new byte[1024];
                   int len = -1;
                   byte cmd='d';
                  while(true) {
                        if(!send && (len = in.read(buffer))!=-1) { //start
of if
                                    cmd= buffer[0];
                                    send=true;
                                    System.out.println("Received: " +
(char) cmd);
                                    Thread.sleep(1000);
                        } //end of if
                                else if(send){
                                    if(cmd=='g') { out.write(cmd='e');
send=false; Thread.sleep(1000);}
                                    else { out.write(cmd='d'); send=false;
Thread.sleep(1000);}
                                    System.out.println("Sent: " + (char)
cmd);
```

```
                                }
                        } //end of while
                } //outer if
                else
                System.out.println("Error: Only serial ports are handled by
this example.");
                } //outer else
        }

        public static void main ( String[] args ) {
                try { (new TwoWaySerialComm()).connect("COM7");
                }
                catch ( Exception e ) {
                e.printStackTrace();
                }
        } //end of main
} //end of class
```