Chapter 5: Serial Communication

1. Serial Communication- Review

Unlike parallel communication which transmits or receives a byte of data, for an 8-bit parallel communication, simultaneously, serial communication sends or receives a single bit at a time. Although this is slower than parallel communication, which allows the transmission of an entire byte at once, it is simpler and can be used over longer distances.

Serial communication is a very common protocol for device communication that is standard on almost every PC. Most computers include two serial ports. Serial communication is also a common communication protocol that is used by many devices for instrumentation; numerous GPIB-compatible devices also come with a serial communication port. Furthermore, serial communication can be used for data acquisition in conjunction with a remote sampling device.

Serial communication has two types: asynchronous and synchronous communications.
Synchronous communication requires more complex interface and clock is sent along with data with higher rate than asynchronous communication. Synchronous communication requires that each end of an exchange of communication respond in turn without initiating a new communication. A typical activity that might use a synchronous protocol would be a transmission of files from one point to another. As each transmission is received, a response is returned indicating success or the need to resend.

On the other hand, asynchronous communication is usually for a situation where data can be transmitted intermittently rather than in a steady stream. For example, a telephone conversation is asynchronous because both parties can talk whenever they like. If the communication were synchronous, each party would be required to wait a specified interval before speaking.
The difficulty with asynchronous communications is that the receiver must have a way to find the end of talks of the other side so that one starts to speak. In computer communications, this is usually accomplished through special bits to indicate the beginning and the end of each piece of data. Asynchronous communication, therefore, has simpler interface and does not send clock, but requires `start` and `stop` bits. Asynchronous communication protocol is our main interest in this chapter.

<u>Serial Communication Specifications</u>
RS-232 (Recommended standard-232) is a standard interface approved by the Electronic Industries Association (EIA) for connecting serial devices. In other words, RS-232 is a long-established standard that describes the physical interface and protocol for relatively low-speed serial data communication between computers and related devices. EIA defined it originally for teletypewriter devices. In 1987, the EIA released a new version of the standard and changed the name to EIA-232-D. Many people, however, still refer to the standard as RS-232C, or just RS-232. RS-232 is the interface that your computer uses to talk to and exchange data with your modem and other serial devices. The serial ports on most computers use a subset of the RS-232C standard.

The electrical specification RS232C standard specifies many parameters such as :
1. A "Space" (logic 0) will be between +3 and +25 Volts.

2. A "Mark" (Logic 1) will be between -3 and -25 Volts.
3. The region between +3 and -3 volts is undefined.
4. An open circuit voltage should never exceed 25 volts. (In Reference to GND)
5. A short circuit current should not exceed 500mA. The driver should be able to handle this without damage. (Take note of this one!)

Above is no where near a complete list of the EIA standard. Line Capacitance, Maximum Baud Rates etc are also included. For more information please consult the EIA RS232-E standard. It is interesting to note however, that the RS232C standard specifies a maximum baud rate of 20,000 bps, which is rather slow by today's standards. Revised standards, EIA-232D & EIA-232E were released, in 1987 & 1991 respectively.

On the other hands, there are two similar standards similar to RS232: RS422 and RS485.

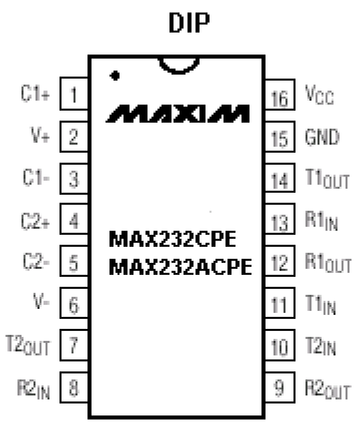
RS422 is a Standard interfaces approved by EIA, and designed for greater distances and higher Baud rates than RS232. In its simplest form, a pair of converters from RS232 to RS422 (and back again) can be used to form an "RS232 extension cord." Data rates of up to 100K bps and distances up to 4000 Ft. can be accommodated with RS422. RS422 is also specified for multi-drop (party-line) applications where only one driver is connected to, and transmits on, a "bus" of up to 10 receivers.

RS485 is an EIA standard for multipoint communications. It supports several types of connectors, including DB-9 and DB-37. RS-485 is similar to RS-422 but can support more nodes per line RS485 meets the requirements for a truly multi-point communications network, and the standard specifies up to 32 drivers and 32 receivers on a single (2-wire) bus.

A brief comparison of the above 3 standards is summarized in the table below.

Specifications	**RS232**	**RS422**	**RS488**
Mode of Operation	Single Ended	Differential	Differential
Max No. of Tx and Rx	1Tx, 1Rx	1Tx, 10Rx	32Tx, 32Tx
Max Cable Length	50 ft	4000 ft	4000 ft
Max Data Rate	20 kbps	100kbps	100kbps
Min Driver Output range	±5 to ±15 V	± 2 V	± 1.5 V
Max Driver Output Range	± 25 V	± 6V	± 6V

<u>RS232 Level Converter</u>
Almost all digital devices, including 16F877 and PIC board, require either TTL or CMOS logic levels. Therefore the first step to connecting a device to the RS-232 port is to transform the RS-232 levels back into 0 and 5 Volts. This level conversion is done by RS-232 Level Converters. The most common RS-232 Level Converter is MAX232 Multi-channel RS-232 Driver chip, which includes a charge pump which generates +10V and -10V from a single 5V supply, from Maxim Semiconductor. The package contains 2 drivers and 2 receivers: in other words, one MAX232 chip can support two serial communication ports. MAX232 needs electrolyte capacitors and whose values are determined by which device is used. For MAX232CPE, a DIP package, need 1.0 uF capacitors, while MAX232ACPE requires 0.1 uF capacitors. From the listed capacitors, capacitor C5 may be omitted. Now you better understand the minimum hardware in your PIC board especially around the MAX232 chip.



	CAPACITANCE (uF)				
	C1	C2	C3	C4	C5
MAX232CPE	1.0	1.0	1.0	1.0	1.0
MAX232ACPE	0.1	0.1	0.1	0.1	0.1

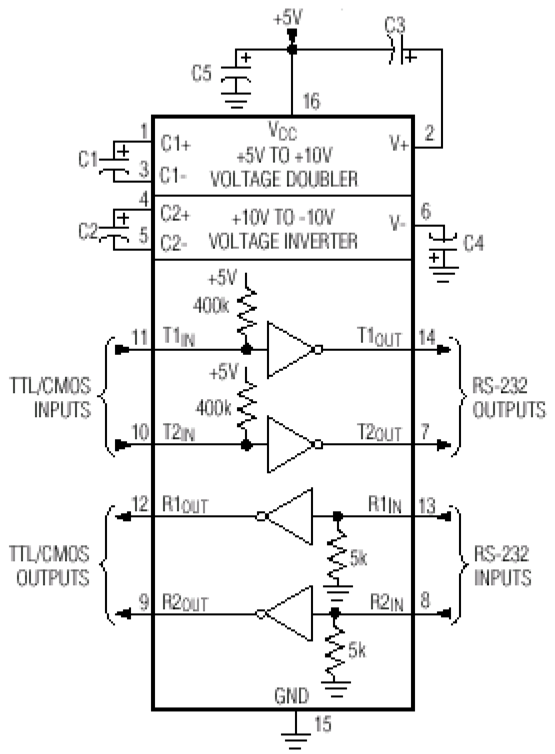


Fig. 19 DIP connection to capacitors

Hardware Properties
Devices which use serial cables for their communication are split into two categories: Data Communications Equipment(DCE) and Data Terminal Equipment (DTE). DCEs are devices such as your modem, plotter etc., while DTE is your PC or Terminal. Therefore, if you have a serial communication connection from a PIC board to a PC, your PIC board is a DCE device.

As we discussed above, a typical DTE is a computer and a typical DCE is a modem. Then, let's consider communication speed between DTE and DCE. However, the speed from DTE to DCE, referred to as *terminal speed*, must be faster than the speed between DCE and DCE, referred to as *line speed*, as in the link between modems.

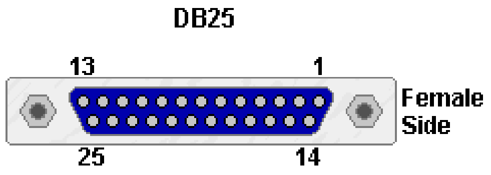
The most common modems, if they have in addition to Ethernet connection, today are 28.8Kbps or 33.6Kbps modems. Therefore we should expect the DCE to DCE speed to be either 28.8K or 33.6K. Then, what would be the terminal speed between DTE and DCE? In other words, if you have 28.8 Kbps, do you always have to set your serial communication software such as `Hyper Terminal` in `Windows` system at 28.8 Kbps as PC's terminal speed?

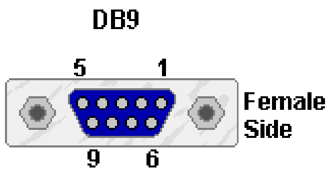
Let's look inside a modern modem. Today's Modems have Data Compression built into them which compresses and decompresses data. An 1 to 4 compression would be typical of a text file transmission. If we were transferring that text file at 28.8K (DCE-DCE), then when the modem compresses it you are actually transferring 115.2 Kbps between computers and thus have a DCE-DTE speed of 115.2 Kbps. This is why the DCE-DTE is usually set much higher than your modem's connection speed.
So if DTE to DCE speed is much faster than DCE to DCE speed, sooner or later data is going to get

lost as a small Modem buffer overflows, thus flow control is used. Flow control has two basic varieties: hardware or software control. Software flow control, sometimes expressed as Xon/Xoff, uses two characters Xon and Xoff. Xon is normally indicated by the ASCII 17 character where as the ASCII 19 character is used for Xoff. When the computer fills it up the buffer, the modem sends an Xoff character to tell the computer to stop sending data. Once the modem has room for more data it then sends an Xon character and the computer sends more data. This type of flow control has the advantage that it doesn't require any more wires as the characters are sent via the Transmit (TxD) and the Receive (RxD) lines. Hardware flow control is also known as (Request to Send)RTS/ (Clear to Send) CTS flow control. It uses two extra wires (i.e., RTS and CTS) for the control. The data is transmitted and received by the TxD and RxD lines. Thus hardware flow control will not slow down transmission times like Xon/Xoff does. When the computer wishes to send data it takes active the RTS line. If the modem has room for this data, then the modem will reply by taking active the CTS and the computer starts sending data. If the modem does not have the room then it will not send a CTS.

Serial Communication Ports

Serial Ports come in two "sizes" maybe three if we include the RJ-45 (Registered Jack – 45) ports: The RJ-45 connector is commonly used for network cabling and for telephony applications. It's also used for serial connections in special cases. However, our focus is on the typical two serial ports and connectors: D-Type 25 pin connector and the D-Type 9 pin connector.
Both types are male on the back of the PC, thus you will require a female connector on your PIC board. Below is a table of pin connections for the 9 pin and 25 pin D-Type connectors and RJ-45.





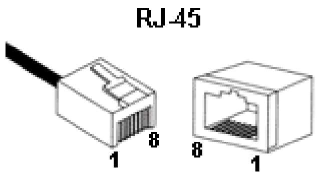


Fig. 20(a) D-Type 25 pin connector Fig. 20(b)D-Type 9 pin connector Fig. 20(c) RJ45

D-Type 25 Pin No.	D-Type 9 Pin No.	RJ-45 type 8 Pin No.	Abbreviation	Full Name
2	3	5	TxD	Transmit Data
3	2	6	RxD	Receive Data
4	7	7	CTS	Clear To Send
5	8	8	RTS	Request To Send
6	6		DSR	Data Set Ready
7	5	4	SG	Signal Ground
8	1	2	CD	Carrier Detect
20	4	3	DTR	Data Terminal Ready
22	9	1	RI	Ring Indicator

Pin description

- TxD: This pin carries data from a DTE (or a PC) to a DCE (or a modem or our PIC).
- RxD: This pin carries data from a DCE to a DTE.
- DTR (Data Terminal Ready): This pin is used by a DTE to signal that it is ready to communicate with a DCE.
- DSR (Date Set Ready): -Similarly to DTR, this pin is an indication from a DCE that a link is ready.
- CD (Carrier Detect): This pin is to indicate that a DCE has received carrier from a remote DCE.
- RTS (Request To Send): This pin is used for DTE to request clearance from a DCE to send data to the DCE.
- CTS (Clear To Send): This pin is used by the DCE to acknowledge the DTE's RTS Signal.
- RI (Ring Indicator): This pin is used to indicate an incoming call rings a phone. In an auto answer modem, this pin is used to signal a receipt of a telephone rign.

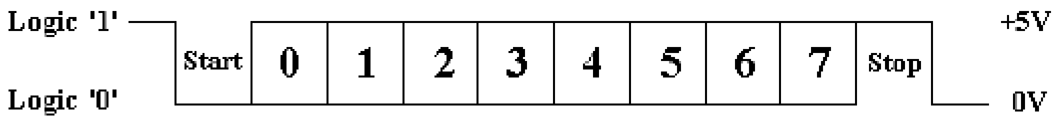
In general, the CD and the RI lines are only available in connections to a modem. Because most modems transmit status information to a PC when either a carrier signal is detected (i.e. when a connection is made to another modem) or when the line is ringing, these two lines are rarely used.

Universal Asynchronous Receiver/Transmitter (UART)

In order to do anything useful with derially transmitted data, we must convert it back to parallel. This series-to-parallel conversion is done by Universal Asynchronous Receiver/Transmitter (UART). One type of UART you can easily find insider your PC is the Intel 8250 and compatibles. These devices have configuration registers accessible via the data and address buses which have to be initialized before use. Most Microprocessors / Microcontrollers including 16F877 has build-in Serial Communication Interfaces (SCI). Therefore there is no need to connect a 40 pin 16550 or 8250 UART to your PIC 16F877 chip.

PIC connection via serial cable to PC
The connection of PIC 16F877's UART via a serial cable to the PC's COM port requires only 3 wires: TxD, RxD & SG. Go back to the hardware diagram and look for these 3 pins and their connections to 16F877 via RS-232 level converter (MAX232). Any data transmitted from the PC is received by the TxD pin of DB-9 connector and to the RXin pin of MAX232. Then, logic leveled data from RXout are fed to RX pin of 16F877. Similarly, any data in logic level transmitted from 16F877's TX pin arrives at TXin pin of MAX232 and, after their levels are converted to RS-232 standard, comes out from the Txout pin of MAX232, and is connected to Pin #2 (RxD) of DB-9 Connector to PC's COM port. The Signal Ground (SG) must be connected so both grounds of PC and PIC board are common to each other.

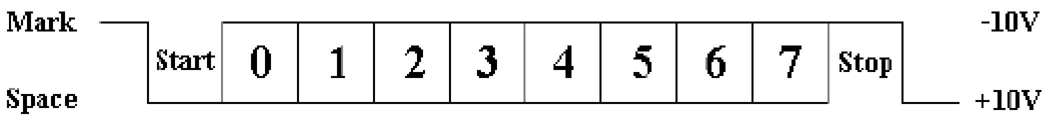
RS232 Protocol and Waveforms
RS-232 communication is asynchronous. That is a clock signal is not sent with the data. Each word is synchronized using it's Start bit, and an internal clock on each side keeps tabs on the timing.



The diagram above, shows the expected waveform from TxD pin of the serial communication port of 16F877 when using the common 8N1 format. 8N1 signifies 8 Data bits, No Parity and 1 Stop bit format. The RS-232 line, when idle, is in the Mark state (Logic 1).
A transmission starts with a start bit which is Logic 0. Then each bit is sent down the line, one at a time. The LSB (Least Significant Bit) is sent first. A Stop bit (Logic 1) is then appended to the signal to make up the end of a transmission.

The diagram shows the next bit after the Stop bit to be Logic 0. This indicates that another word is following, that is, the logic 0 following the Stop bit is the Start bit for the next transmission. If there is no more data, then the receive line will stay in it's idle state, logic 1. The data sent using this method, containing a data between a Start bit and a Sop bit, is said to be *framed*.

The above diagram of logic level coming out of 16F877 is converted to an RS-232 specified level via MAX232. Therefore, the actual input to the COM port of PC would look like the diagram shown below.



RS-232 logic levels uses +3 to +25 volts to signify a "Space" (Logic 0) and -3 to -25 volts for a "Mark" (logic 1). Any voltage in between these regions (i.e., between +3 and -3 Volts) is undefined.

By the way, who or which generates this serial communication bit information from a byte data? It is the internal UART module of 16F877. Therefore, for the waveforms and protocol, it is just for understanding how serial communication works. All you have to do is utilize the internal serial communication module by configuring and operating the registers of the module. The utilization of the UART module is discussed soon.

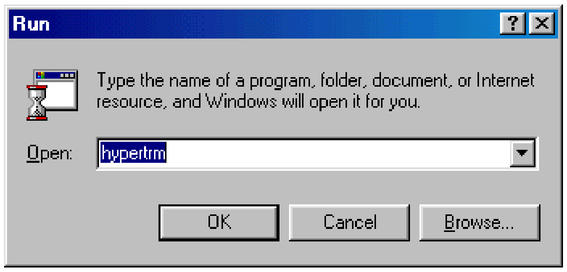
2. Terminal Program in Computer

To serially communicate your 16F877 with your PC, the PC must have a serial communication software (or terminal program) to make your serial port ready and serial communication operational. Fortunately, any PC with Windows operating systems comes with a terminal program: `Hyperterminal`. Apple Macintosh users will need to obtain a terminal program from a shareware web site, or from their software supplier. `ZTerm` is a good terminal program which can be obtained from most of the popular download sites for the Mac.

To run HyperTerminal for PIC communication with 8N1 format with 19200 bps, follow the steps.

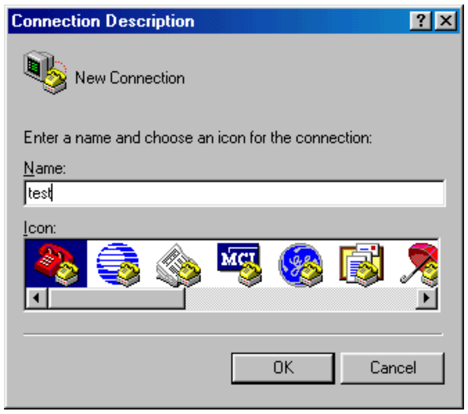
Step 1. Load HyperTerminal by Click Start > Run. In the box which appears type in **HYPERTRM** and click OK.

(Or click **Start>Programs>Accessories>Communications>HyperTerminal)**

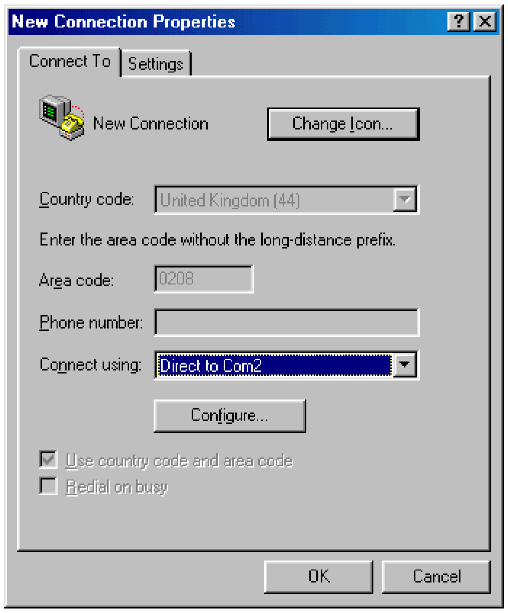

Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open: hypertrm
OK
Cancel
Browse...

If you see the error message 'Cannot find HYPERTRM' you will need to install HyperTerminal:
Click Start > Settings > Control Panel > Add / Remove Programs.
(a) In Add / Remove Programs select Properties and then click the Windows Setup tab.
(b) Double-click Communications
(c) Check the box for HyperTerminal
(d) Click OK, and then OK again to install.
Note: You may be asked for your Windows installation disk for this procedure.

Step 2. When HyperTerminal starts you will be presented with a '**Connection Description**' dialogue box, click **'Cancel'** to continue.



Step 3. Select your COMPort: Select '**File**' then '**Properties**' from the menu bar, and the '**New Connection Properties**' window will open. Click on the arrow for the '**Connect Using**' drop down box Select '**Direct to ComX**' (where X is the COMPort you are using to connect the modem. - e.g. COM1, or COM2, etc). Click OK.


New Connection Properties

Connect To | Settings

New Connection

Country code: United Kingdom (44)

Enter the area code without the long-distance prefix.

Area code: 0208

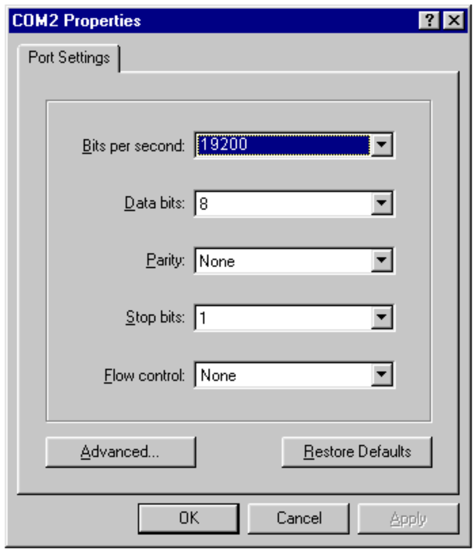
Phone number:

Connect using: Direct to Com2

☑ Use country code and area code

☐ Redial on busy

Step4: Port Setting: The COM port properties box comes up. Make sure that that you set the Baud rate to 19200, and No Stop Bit, 8 Data bits. And select **None** for Flow control. Click OK.


COM2 Properties

Port Settings

Bits per second: 19200

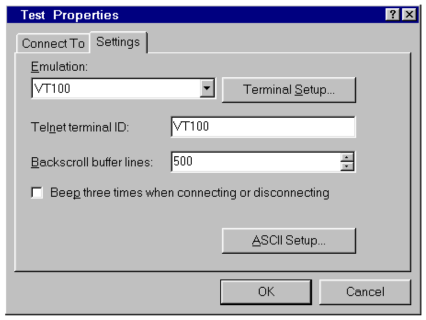
Data bits: 8

Parity: None

Stop bits: 1

Flow control: None

Step 5: Emulation Setup: Click on the Settings tab. Change the Emulation type to **VT100** .Now, click on **OK**.


Test Properties

Connect To | Settings

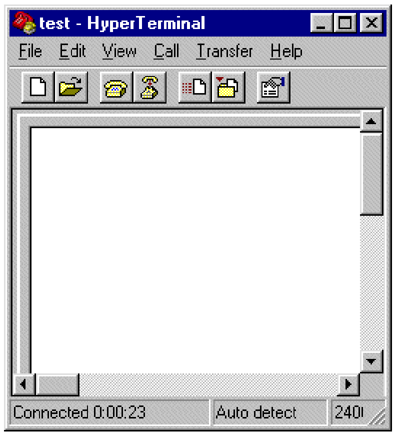
Emulation: VT100

Telnet terminal ID: VT100

Backscroll buffer lines: 500

☐ Beep three times when connecting or disconnecting

Step6: Save your file name as `test.ht`, and run it. Then, the following window will pop up and your computer is ready to communication with the PIC board. In the Serial Communication Module study for 16F877, our main focus is to display what 16F877 reads and processed for verification or display purpose. Also, we will receive command typed by the keyboard from the computer side.



The last concern in a terminal program involves the hardware configuration of your computer. Probably, you do not have to worry about the COM port address and Interrupt request (IRQ) setting to run a serial communication software. However, if your serial communication does not work and that stems from COM port of your PC, this information may help you to find a solution. Below is the standard port addresses and these should work for most PCs.

NAME	ADDRESS	IRQ
COM1	3F8	4
COM2	2F8	3
COM3	3E8	4
COM4	2E8	3

3. 16F877 Serial Communication Module (USART)

The Universal Synchronous Asynchronous Receiver Transmitter (USART) module is one of the two serial I/O modules provided by 16F877: the other is the SSP module. The USART is also known as a Serial Communications Interface (SCI). The USART can be configured as a full duplex asynchronous system that can communicate with peripheral devices such as CRT terminals and personal computers, or it can be configured as a half duplex synchronous system that can communicate with peripheral devices such as A/D or D/A integrated circuits, Serial EEPROMs etc. As mentioned earlier, the focus of the serial communication is centered on the asynchronous communication.

The configure and operation of the USART module is controlled by a few SFRs: RCSTA (Receive Status and Control Register in bank 0 at 18h), TXSTA (Transmit Status and Control Register in bank 1 at 98h), SPBRG (Baud Rate Generator in bank 1 at 99h), and PIR1 (Peripheral Interrupt Register 1 in bank 1 at 0Ch).
In addition, since the two serial ports TX (pin #25) and RX(pin #26) can also be used as PORTC pins,

while setting the SPEN (Serial Port Enable Bit) bit (RCSTA<7>), the TRISC bits must be configured: the two pins must be set.
Let's first examine the TXSTA register. As indicated below, for an synchronous communication with 8N1 format, we configure the register as follows:

TXSTA: Transmit Status and Control Register

bit 7 … bit 0

CSRC	TX9	TXEN	SYNC	----	BRGH	TRMT	TX9D
X							

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

TXEN: Transmit Enable bit
1 = Transmit enabled
0 = Transmit enabled

SYNC: Mode Select bit
1 = Synchronous mode
0 = Asynchronous mode

BRGH: 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 data
Can be parity bit

TXSTA<7> (CSRC): Don't care for this mode. Let's clear it.
TXSTA<6> (TX9): Since the format 8N1 dictates 8-bit data transmission, it should be cleared.
TXSTA<5> (TXEN): Set this bit to initiate data transmission.
TXSTA<4> (SYNC): Clear this bit to select asynchronous mode.
TXSTA<3> : Unused. Let's clear it.
TXSTA<2> (BRGH): This is to select not the Baud rate itself but to select high Baud or low Baud rate. For a given Baud rate like 19200 bps, whether setting or clearing this bit determines the content of SPBRG for the chose Baud rate. Set or clear works, but with proper calculation of the content for SPBRG. For the time being let's clear.
TXSTA<1> (TRMT):This bit is used to check the current status of data transmission. So this bit is not to be set or cleared. By the, this bit is not going to be used even for this purpose: we are going to use a flag of PIR1 register. Details on this subject will be discussed with PIR1 register explanation.
TXSTA<0> (TX9D): This pin is used only when 9-bit data transmission or 1 parity bit format is

selected. Since 8N1 format does not involve either one, this pin set is ignored. Then clear it.

So asynchronous communication mode with 8N1 format would make the content of TXSTA by the following instructions:

```
      banksel     TXSTA            ;B      B
      movlw       B'00100000'      ;76543210
                                   ;00100000
      movwf       TXSTA            ;Asyn mode, 8N1 format, low baud rate,
                                   ;TX enable
```

The next register we will investigate is RCSTA.

RCSTA: Receive Status and Control Register

bit 7 bit 0

SPEN	RXP	SREN	CREN	----	FERR	OERR	RX9D

SPEN: Serial Port Enable bit
1 = Serial port enabled
0 = Serial port disabled

RX9: 9-bit Receive bit
1 = 9-bit reception
0 = 8-bit reception

SREN: Single Receive bit
X

CREN: Continuous Receive bit
1 = Enables
0 = Disables

X

FERR: Framing Error
1 = Framing error
0 = No framing error

OERR: Overrun Error
1 = Overrun error
0 = No overrun error

RX9D: 9th bit of data,
can be parity bit

Now we configure the RCSTA register by setting or clearing each bit.
RCSTA<7> (SPEN): Set this bit to enable (start) reception.
RCSTA<6> (RX9): Since the format 8N1 dictates 8-bit data transmission, it should be cleared.
RCSTA<5> (SREN): Not used. So clear it.
RCSTA<4> (CREN): Set this bit to receive data continuously.
RCSTA<3> : Unused. Let's clear it.
RCSTA<2> (FERR): This bit is used to check the current status of data reception for frame error. In

configuration we gave nothing to do. For the time being let's clear.
RCSTA<1> (OERR): This bit is used to check the current status of data reception for overrun error . In configuration we gave nothing to do. For the time being let's clear.
RCSTA<0> (RX9D): This pin is used only when 9-bit data transmission or 1 parity bit format is selected. Since 8N1 format does not involve either one, this pin set is ignored. So clear it.

So asynchronous communication mode with 8N1 format would make the content of RCSTA by the following instructions:

```
        banksel     RCSTA           ;B      B
        movlw       B'10010000'     ;76543210
                                    ;10010000
        movwf       RCSTA           ;8N1 format and RX enabled
```

The next step is to configure SPBRG with the select Baud rate of 19200. The SPBRG is a dedicated 8-bit Baud rate generator. Therefore the value for SPBRG is 0 to 255 (0 – FFh). The SPBRG register controls the period of a free running 8-bit timer. In asynchronous mode, as explained before, bit BRGH (TXSTA<2>) controls the baud rate. The content of SPBRG is determined by the formula computation of the baud rate for different USART modes. Given the desired baud rate and f_{osc}, the nearest integer value for the SPBRG register can be determined.

When TXSTA<2>=0 (low rate selection), the formula for the value of SPBRG, N_{BRG}, is:

$$N_{BRG} = \frac{f_{osc}}{64 \times B_d} - 1,$$

where, f_{osc} is crystal oscillation frequency and B is a desired Baud rate.

If TXSTA<2>=1 (high rate selection), the formula for the value of SPBRG, N_{BRG}, is changed to:

$$N_{BRG} = \frac{f_{osc}}{16 \times B_d} - 1.$$

Therefore, with low rate selection with TXSTA<2> cleared, the value for N_{BRG} for 19200 bps for 20MHz crystal oscillation is:

$$N_{BRG} = \frac{f_{osc}}{64 \times B_d} - 1 = \frac{20{,}000{,}000}{64 \cdot 19200} - 1 = 15.27 \rightarrow 15 \text{ (the nearest integer).}$$

With the chosen value 16, the actual Baud rate we can get is:

$$B_c = \frac{f_{osc}}{64 \cdot (N_{BRG} + 1)} = \frac{20{,}000{,}000}{64 \cdot 16} = 19531$$

So there is $\frac{19531 - 19200}{19200} \times 100 = 1.7(\%)$ error, and it is acceptable.

The instructions to configure SPBRG are:

```
        banksel     SPBRG
        movlw       0x0F        ;15 in decimal
        movwf       SPBRG       ;[SPBRG]=0Fh for 19200bps
```

There are two more registers involved in the USART module for an asynchronous communication: the transmit buffer register (TXREG) and the receive buffer register (RCREG). The way these registers work is stemmed from the internal blocks for transmission and reception.

Let's first look at the USART transmission block diagram shown below. The data to be transmitted is, as you see, first placed to an 8-bit buffer (TXREG). The flag bit, TXIF (PIR1<4>), is to indicate the buffer is empty (TXIF=1) or full (TXIF=0). PIR1 is the Peripheral Interrupt Register located in bank 0 at 0Ch. In other words, by polling the TXIF bit, we know if we can send the next data or not. When the buffer is full, we have to wait until it is empty. The content inside TXREG is parallel transmitted to the transfer shift register (TSR), which is not accessible for us. The clock to shift one bit at a time, LSB first and MSG last, comes the Baud rate generator (SPBRG) but this must be enabled by the transmit enable bit (TXEN). When all the bits are shifted, its status is flagged by TRMT bit of TXSTA. In the TSR, additional bits such as `Start`, `Stop`, and `Parity` are inserted or concatenated. The final stage of the transmission if controlled and enabled by SPEN (serial communication enable bit) and bits are sent out to the TX pin (#26) of 16F877.

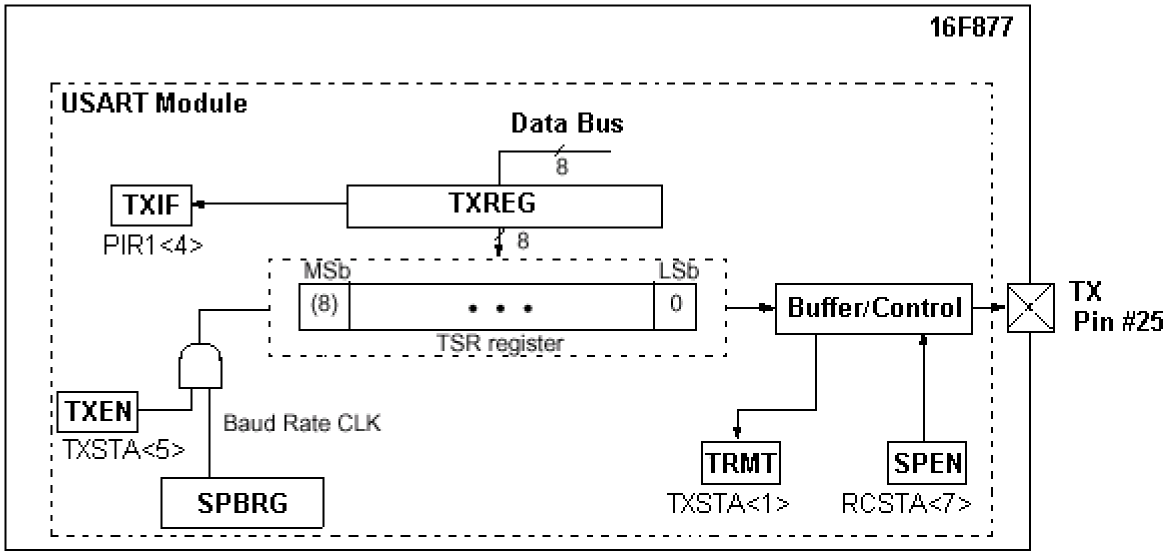


Fig. 21 USART transmission block diagram

Therefore, sending out a data can be done by the following instructions:

```
            movlw       'A'         ;character to send out
Txwait      btfss       PIR1, TXIF  ;is TXIF bit set? (or is TXREG empty?)
            goto        Txwait      ;No, it's still full
            ~~movlw~~movwf TXREG       ;yes. Place your data to TXREG
```

The reception of data to the USART receive block diagram is shown below. The bit stream arrived at the RX pin (#26) of 16F877 is entered only when the serial communication enabled by the SPEN bit of RCSTA. Then, the bit information is captured and recovered by the clocking of SPBRG. So if the input data is not in the same Baud rate as in the receive block, the captured and recovered data cannot be the same data sent. The recovered data is fed to the internal Receive Shift Register (RSR). In

RSR, frame errors and overrun errors are flagged, and the frame bits (`Start`, `Stop`, and/or `Parity`) are removed before transferred to the Receive Buffer Register (RCREG). The status of RCREG is flagged by the RCIF bit (PIR1<5>): 1 when buffer is full and 0 when empty. Therefore, when in reception mode, we have to monitor the RCIF bit if we can safely move the content of RCREG to `W` register.

Therefore, reception can be done by the following instructions:

```
Rxwait      btfss       PIR1, RCIF  ;is TXIF bit set? (or is RCREG full?)
            goto        Rxwait      ;No, it's empty
            movf        RCREG, 0    ;yes. Move the data to W
```

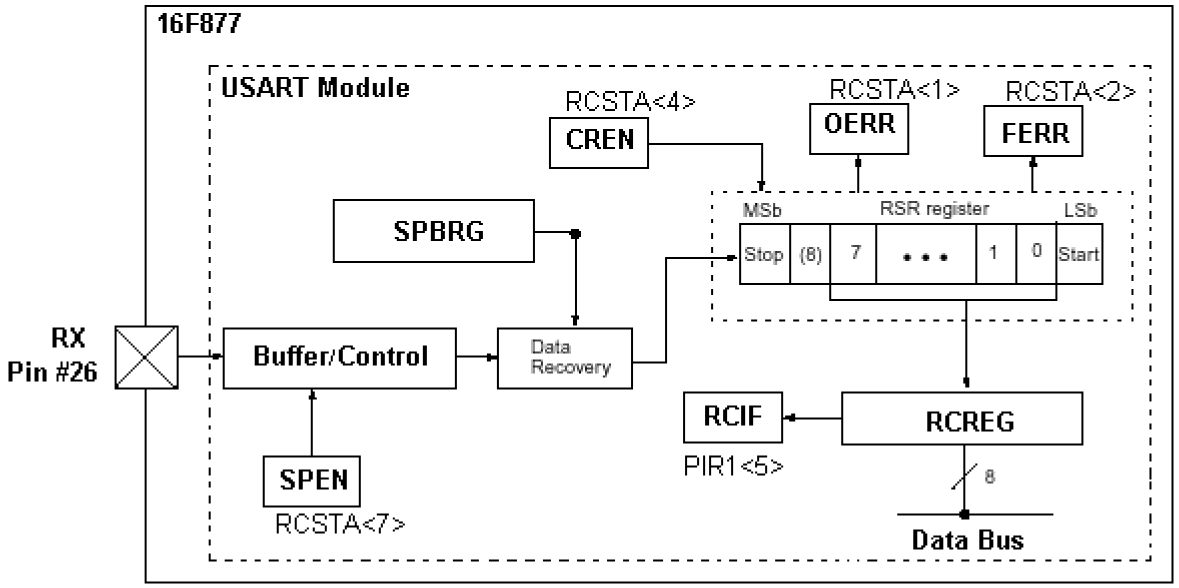


Fig. 22 USART receive block diagram

4. PIC16F877 Serial Communication Coding Examples

Key Echoing

As the first example for serial communication, we want to write a code to echo the keyboard. What it means is display the character, on the monitor of your PC, of a key typed by your keyboard. It sounds odd to beginners. Why do we need this kind of program? Isn't any character I type on my keyboard automatically displayed on the monitor, as I write a Word file? When you run the `Hyperterminal`, your computer is ready for serial communication via a COM port with other devices. When you type a key, it is an action of transmission of the key to the COM port. When data is received from the COM port, it is displayed on the monitor. In other words, under Hyperterminal, your keyboard is not connected to the monitor to display the key you type.

In other words, to display a key on your monitor must involve the other device, which is our 16F877. What involves is like this: when we type a key, the data is sent from the COM port in a speed set by Baud rate setting say 19200bps, via a serial cable, to the DB-9 pin of the PIC board. This RS-232 level data is converted to a logic level data by MAX232 level converter. The logic level data now arrives at the RX pin of 16F877. If the USART module and an asynchronous communication are enabled with the same Baud rate of 19200bps by the configuration of the RCSTA, TXSTA, and PSBRG, then the data would arrive at the RCREG register. As soon as the RCREG is full, RCIF flag is set. Then, we move the content of RCREG to W to store the key character sent by the keyboard. As we know, this reception does not display the key character on the monitor. We have to sent back the key character to the computer.

So we place the content of W register to TXREG when TXREG is empty, which is flagged by TXIF bit of PIR1. The data in TXREG is eventually transmitted back bit by bit to the computer through the same path, and the computer upon reception of the data display it on the monitor. It involves much more than one thinks it would.

Therefore the following echo program thus consists of mainly three parts: configuration of registers, reception, and transmission. Examine the code by carefully following comments. Note that the MPLAB directive `banksel` does not go with any label. In other words, there should not be a label at the same line with `banksel` directive.

```
;rxtx-v1.asm
;
;This program uses USART moddule and RX & TX pins of 16F877
;
;The program echoes on PC screen a character typed from keyboard
;
;Asynchronous mode of serial communication
;
;
;PC's Hyper Terminal Set-Up: 8N1 19200
;Baud:           19200
;Data Bit:       8
;Parity:         None
;Stop Bit:       1
;Control:        None

        list P = 16F877

STATUS      EQU   0x03        ;Declaration of registers and bits
TXSTA       EQU   0x98        ;TX status and control
RCSTA       EQU   0x18        ;RX status and control
SPBRG       EQU   0x99        ;Baud Rate assignment
TXREG       EQU   0x19        ;USART TX Register
RCREG       EQU   0x1A        ;USART RX Register
PIR1        EQU   0x0C        ;USART RX/TX buffer status (empty or full)
RCIF        EQU   0x05        ;PIR1<5>: RX Buffer Status 1-Full  0-Empty
TXIF        EQU   0x04        ;PIR1<4>: TX Buffer Status 1-empty 0-full

;
;==========================================================
      org         0x0000            ;line 1
      goto        START             ;line 2

      org         0x05
START
;Configuration Part
      banksel     SPBRG
      movlw       0x0F              ;15 for 19200bps
      movwf       SPBRG

      banksel     TXSTA
      movlw       B'00100000'        ;Asynchronous mode, 8N1 format

      banksel     RCSTA
      movlw       B'10010000'        ;USART enabled with async mode
      movwf       RCSTA

;Reception Part

      banksel     PIR1
Rxwait
      btfss       PIR1, RCIF         ;RX Buffer Full?  (i.e. Data Received?)
      goto        Rxwait
      movf        RCREG,0            ;mode received data  to W
;
; Now the key is stored in W register

;Echo Back (Transmission Part)
      banksel     PIR1
Txwait
      btfss       PIR1, TXIF         ; Check if TX buffer is empty
      goto        Txwait
      movwf       TXREG              ; Place the character (in W) to TX buffer

      goto        Rxwait             ;for next key

      END
; end of code
```

Since serial communication is frequently used for display and value check, it would be much better to make the reception and transmission parts into subroutines. Even the configuration part can be made into a subroutine.

A subroutine starts with the subroutine's name as label, and ends with `return`. The reception subroutine `Rxpoll` goes like below.

```
RXPOLL                        ;The result is placed in W register
      banksel     PIR1
      btfss       PIR1, RCIF  ;RX Buffer Full?  (i.e. Data Received?)
      goto        RXPOLL
      banksel     RCREG
      movf        RCREG,0            ;received data  to W
      return
```

The subroutine for transmission `Txpoll` looks like this:

```
TXPOLL                        ;Send data placed in W register
      banksel     PIR1
      btfss       PIR1, TXIF  ; Check if TX buffer is empty
      goto        TXPOLL
      banksel     TXREG
      movwf       TXREG       ; Place the character in W to TX buffer
      return
```

The configuration part for asynchronous communication `Async` can go like this:

```
Async                                ;Async mode with 8N1 format of 19200bps
      banksel     SPBRG
      movlw       0x0F               ;15 for 19200bps
      movwf       SPBRG

      banksel     TXSTA
      movlw       B'00100000'       ;Asynchronous mode, 8N1 format

      banksel     RCSTA
      movlw       B'10010000'       ;USART enabled with async mode
      movwf       RCSTA
      return
```

Then the code for key echo can be rewritten like below.

```
;rxtx-v2.asm
;
;This program uses USART module and RX & TX pins of 16F877
;
;The program echoes on PC screen a character typed from keyboard
;
;Asynchronous mode of serial communication
;
;
;PC's Hyper Terminal Set-Up: 8N1 19200
;Baud:            19200
;Data Bit:        8
;Parity:          None
;Stop Bit:        1
;Control:         None

        list P = 16F877

STATUS      EQU   0x03        ;Declaration of registers and bits
TXSTA       EQU   0x98        ;TX status and control
RCSTA       EQU   0x18        ;RX status and control
SPBRG       EQU   0x99        ;Baud Rate assignment
TXREG       EQU   0x19        ;USART TX Register
RCREG       EQU   0x1A        ;USART RX Register
PIR1        EQU   0x0C        ;USART RX/TX buffer status (empty or full)
RCIF        EQU   0x05        ;PIR1<5>: RX Buffer Status 1-Full  0-Empty
TXIF        EQU   0x04        ;PIR1<4>: TX Buffer Status 1-empty 0-full

;
;==========================================================
      org         0x0000            ;line 1
      goto        START             ;line 2
;=======================================================
      org         0x05
START
;Configuration Part
      call        Async             ;async mode
Again
      call        Rxpoll            ;receive a key
      call        Txpoll            ;transmit the key
      goto        Again             ;repeat


;Subroutines
RXPOLL                        ;The result is placed in W register
      banksel     PIR1
      btfss       PIR1, RCIF  ;RX Buffer Full?  (i.e. Data Received?)
      goto        RXPOLL
      banksel     RCREG
      movf        RCREG,0           ;received data  to W
      return
```

```
;
TXPOLL                        ;Send data placed in W register
      banksel     PIR1
      btfss       PIR1, TXIF  ; Check if TX buffer is empty
      goto        TXPOLL
      banksel     TXREG
      movwf       TXREG       ; Place the character in W to TX buffer
      return

;
Async                               ;Async mode with 8N1 format of 19200bps
      banksel     SPBRG
      movlw       0x0F              ;15 for 19200bps
      movwf       SPBRG

      banksel     TXSTA
      movlw       B'00100000'       ;Asynchronous mode, 8N1 format

      banksel     RCSTA
      movlw       B'10010000'       ;USART enabled with async mode
      movwf       RCSTA
      return
;
      END                           ;End of Code
```

Hex Number Display

The next example is to display a measured value by the PIC board to the monitor of the computer. Consider a situation that you are monitoring room temperature, voltage, or current using the Analog-to-Digital module of 16F877, then you are displaying the measured value on the monitor. This case you are actually posting a hexadecimal numbers. Consider another case that you are going to find the command code for a TV remote controller using an Infrared (IR) receiver. The code 16F877 received via the IR receiver must be displayed on a monitor to decode (or match) a button pressed to a code the receiver received. In either case, it would be better to know how we send a hexadecimal value to the monitor.

Here we have to remember that when we send a hex number to the computer, the computer will displace the number on the monitor as ASCII (American Standard Code for Information Interchange) data. An ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Below is the ASCII character table and this includes descriptions of the first 32 non-printing characters. To get a hex equivalent for an ASCII character, use a corresponding row number from the left column as the first hex digit, and a corresponding column number from the top row for the second digit. For example, a hex equivalent for the character 'C' is 43h.

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

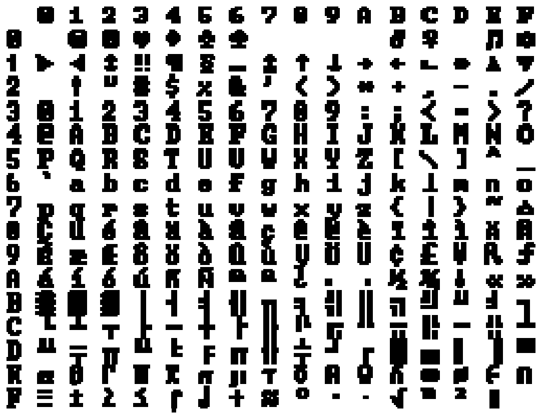
Therefore character 'P' in ASCII code is 50h or 01010000b. Similarly, character 'I' in ASCII code is 49h or 01001001b.

ASCII was actually designed for use with teletypes and so the descriptions are somewhat obscure. The following is a more detailed description of the first 32 non-printing (or special) ASCII characters, often referred to as control codes.

- NUL (null)
- SOH (start of heading)
- STX (start of text)
- ETX (end of text)
- EOT (end of transmission): Not the same as ETB
- ENQ (enquiry)
- ACK (acknowledge)
- BEL (bell): Causes teletype machines to ring a bell. Causes a beep in many common terminals and terminal emulation programs.
- BS (backspace): Moves the cursor move backwards (left) one space.
- TAB (horizontal tab): Moves the cursor right to the next tab stop.
- LF (line feed) - Moves the cursor (or print head) to a new line.
- VT (vertical tab)
- FF (form feed): Advances paper to the top of the next page (for a printer).
- CR (carriage return): Moves the cursor all the way to the left, but does not advance to the next line.
- SO (shift out): Switches output device to alternate character set.
- SI (shift in) : Switches output device back to default character set.
- DLE (data link escape)
- DC1 (device control 1)
- DC2 (device control 2)
- DC3 (device control 3)
- DC4 (device control 4)
- NAK (negative acknowledge)
- SYN (synchronous idle)
- ETB (end of transmission block): Not the same as EOT
- CAN (cancel)
- EM (end of medium)
- SUB (substitute)
- ESC (escape)
- FS (file separator)
- GS (group separator)
- RS (record separator)

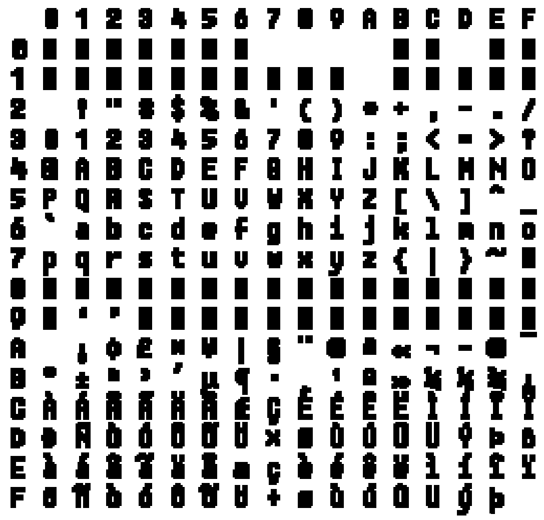
- US (unit separator)

As you see the ASCII table, the ASCII character set only includes values up to 127 decimal (7Fh). However, when the IBM PC was developed, the video card contained one byte for each character in the 80x25 character display, they could use the extra bit per character to invent 128 new characters, for line-drawing and special symbols. The result, of course, was the extended ASCII character set for the IBM PC. The chart below shows (most of) the characters that can be generated by the display in the original IBM PC.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺	☻	♥	♦	♣	♠					♂	♀		♫	☼
1	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
F	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	

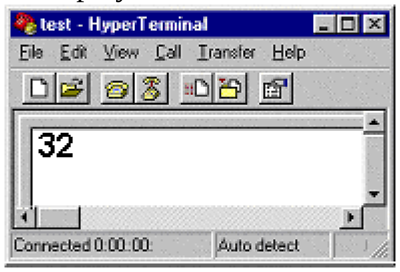
On the other hand, Microsoft Windows ® has a different notion about what the high-order (upper 128) characters are, as shown in the table below.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	■	■	■	■	■	■	■					■	■		■	■
1	■	■	■	■	■	■	■	■	■	■		■	■	■	■	■
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	■
8	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
9	■	'	'	■	■	■	■	■	■	■	■	■	■	■	■	■
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·		¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	

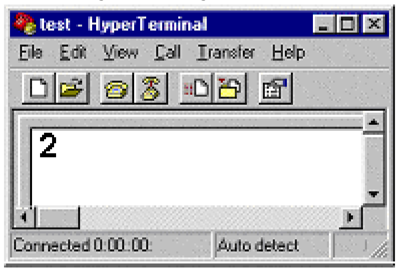
We had enough discussion on ASCII code, and now let's go back to our original intention of sending a hex value and displaying it on a monitor. As you probably knew or noticed from the lengthy introduction of ASCII code in our discussion on serial communication, displaying a hex code not as simple as the echoing code we programmed before. It's because any number we sent via the serial cable to the computer is interpreted as an ASCII character.

For example, after reading an ADC register or IR command, assume that the result, say 32h, is stored in **W** register. If we send the data by calling a transmission subroutine, what would you expect will be displayed on the monitor? Do you expect to see as shown below, with 32?



```
32
```

Actually, what you see would be like this:



```
2
```

So you would erroneously guess that there is a bug in the program. But your code is perfect, only your thinking and expectation have a bug: the ASCII code for 32h is the character '2'.

If the contents of **W** were FFh, the result would be more dramatic and vexing: there would be no display at all. Check the extended ASCII code for Windows and IBM.

So we need a supplementary code to change a byte data in **W** in to two ASCII characters. In other words, the byte data must be split into two nibbles (i.e., 4-bit data) and their ASCII hex equivalents must be sent to the monitor for correct display of the result.

Let's discuss about splitting the byte data into two nibbles. Since we need temporary storage places, for W content and two nibbles, let's allocate a few file registers: `RESULT` (for the hex result), `TEMP` (for temporary space), `NH` (for higher nibble), and `NL` (for lower nibble).

Let's first store the original hex result (in **W**) by:

```
      movwf        RESULT              ;[result]=W
```

Then, store the content again to another temporary location, by:

```
      movwf        TEMP                ;[TEMP]=NH|NL
```

Since the first nibble must be displayed first, we would get the higher nibble first from the result. To do that we swap the content of `TEMP`:

```
        swapf      TEMP,0              ;[TEMP]=W=NL|NH --->W
```

Now **W** has the swapped nibbles. Since NH is our interest for the first hex digit, we want to make the `NL` part of **W** zero. The following does that job. If you `AND` a bit and 0, the result is 0 no matter what value the bit may be.

```
        andlw     0x0F        ;Mask off upper nibble so that W=0|NH
```

Now let's discuss how we convert a nibble with preceding 4 bits of zero into an ASCII equivalent. If you check the ASCII table again, you may see some patterns for the 16 different hex digits, `0 - 9` and `A - F`.

hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ASCII	30	31	32	33	34	35	36	37	38	39	41	42	43	44	45	46

From the pattern, we can see that a number between 0 – 9 can be easily converted to an ASCII equivalent by being added with a hex number 30h. For the hex symbols, the addendum would be 37h, instead (41h-0Ah=37h). Therefore, the only thing left is to know if the content of the nibble is a number (0 – 9) or a symbol (A –F). To do that, let's examine the 16 possible hex numbers in a binary format.

```
Hex   Binary Equivalent
===   =================
      B3    B2    B1    B0
0     0     0     0     0
1     0     0     0     1
2     0     0     1     0
3     0     0     1     1
4     0     1     0     0
5     0     1     0     1
6     0     1     1     0
7     0     1     1     1
8     1     0     0     0
9     1     0     0     1
A     1     0     1     0
B     1     0     1     1
C     1     1     0     0
D     1     1     0     1
E     1     1     1     0
F     1     1     1     1
```

It's very clear that if B3=0, the data is a number. For the symbols, B3 must be 1 and at least one bit of B2 & B1 must be 1. This is a good pattern observation, since the numbers 8 and 9 has all zeros for B2 & B1. So here is a pseudo-code for a hex to ASCII conversion subroutine.

```
1. Get a nibble for a hex digit
2. Check if B3=0. If it is add 30h to the nibble.
3. If not, then check if B2 & B1 are both zero. If they are zero, add 30h to the nibble.
4. If B2 & B1 has at least one 1, add 37h to the nibble.
```

So here goes the hex to ASCII subroutine:

```
; === hex to ascii conversion subroutine
;move a nibble to W before call this routine
;final result will be stored back to W
HTOA
      movwf       ATEMP
```

```
;check 0-9 or A-F
      btfsc       ATEMP, 0x03        ;if B3=0, it is a number in 0 - 7
      goto        RECHK              ;
THIRTY            movlw 0x30         ;then, add 30h
      addwf       ATEMP
      movf        ATEMP,0            ;store the ASCII back to W
      return

RECHK andlw       0x06               ;ANDed with 00000110 to see B2 and B1
      btfsc       STATUS,ZERO        ;
      goto        THIRTY             ;if both are zero, it is a number 8 or 9
      movlw       0x37               ;if at least one is not zero, add 37h
      addwf       ATEMP
      movf        ATEMP,0            ;store the ASCII symbol back to W
      return
```

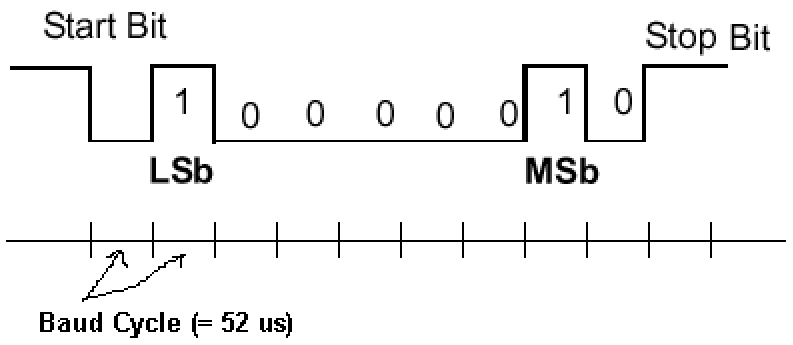
Now let's combine everything into a hex display code. Here again we assume that a result is stored in **W** register. For a simple test of this code would be accomplished, without ADC or IR decoding, by putting a hex number to **W** and see if the hex code in correctly displayed on the monitor.

```
;your routine for measurement or decoding
;here
;your result is now in W
      movwf       RESULT             ;[result]=W
      movwf       TEMP               ;[TEMP]=NH|NL
;for the first hex digit
      swapf       TEMP,0             ;[TEMP]=W=NL|NH --->W
      andlw       0x0F               ;Mask off upper nibble so that W=0|NH
      call        htoa               ;ASCII conversion
      call        Txpoll             ;Transmission to monitor
;for the second hex digit
      movf        RESULT, 0          ;W=[RESULT]=NH|NL
      andlw       0x0F               ;By masking the upper nibble, we have only NL
      call        htoa
      call        Txpoll

;subroutine
HTOA
      movwf       ATEMP
;check 0-9 or A-F
      btfsc       ATEMP, 0x03        ;if B3=0, it is a number in 0 - 7
      goto        RECHK              ;
THIRTY            movlw 0x30         ;then, add 30h
      addwf       ATEMP
      movf        ATEMP,0            ;store the ASCII back to W
      return

RECHK andlw       0x06               ;ANDed with 00000110 to see B2 and B1
      btfsc       STATUS,ZERO        ;
      goto        THIRTY             ;if both are zero, it is a number 8 or 9
      movlw       0x37               ;if at least one is not zero, add 37h
      addwf       ATEMP
      movf        ATEMP,0            ;store the ASCII symbol back to W
      return
;end of code
      END
```

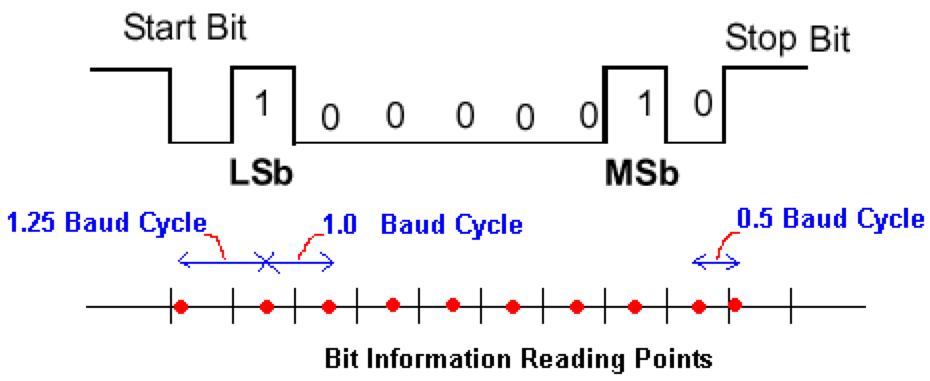
5. Serial Communication without using the USART module

There are many occasions that we need more than one serial port and communication. Consider a situation that we want to use one port for hex data display to a monitor or serial LCD, and another one for serial communication with another PIC chip or any other serial communication device. However, in 16F877, there is only one USART module. In other words, it supports only one serial communication. Some PIC chips does not have even 1 USART.
This extended discussion on serial communication is to introduce a software implementation of an asynchronous serial communication for 16F877. You can use this additional serial port for serial LCD connection or other serial device. With this software serial communication scheme, you can use any two I/O pins for TX and RX for the communication.

Reception
The coding for software serial communication should start with the review of RS-232 communication protocol. Let's consider serial reception first. For reception, in 8N1 format, there would be 8 bits to be tested for their values. For example, the character 'A' (ASCII code of 41h=01000001b) typed from a keyboard would generate a start bit (Low) followed by 8-bit data stream. The data stream is the nominal LSb first format (100000010). The 8-bit data is followed by a Stop bit (High). Then, the communication terminates. So the reception should begin with the start bit monitoring, once the start bit is detected, it regularly monitors the values for the next 8 bits. Then, after the high Stop bit, it terminates the reception process. Since the data bit duration is dependent upon selected Baud rate, we continue our discussion on reception with 19200 bps. With 19200 bps, the data bit duration is $1{,}000{,}000 / 19200 \approx 52$ [μs]. Let's call this 52 μs duration per bit as one "Baud cycle." Note that this is valid only for 19200 bps setting. Then, we proceed the discussion in terms of the Baud cycle.



This is the way our software serial communication works:
1. Monitor the serial input (TX) port for the Start bit.
2. If Start bit is detected, wait for 1.25 Baud cycle for detecting the next bit. The reason for 1.25 Baud cycle delay is to have our monitoring timing at the middle of the next data to minimize any error caused by drift. If there is no Start bit detected, go to 1 and keep monitoring.
3. After the first data bit (LSb) is detected, for the rest of the data bits, we wait only 1 Baud cycle to read the values. It's apparent that after first reading, we can at the center of the next data bit only after 1 Baud cycle.
4. After all 8 bits are read, for Stop bit detection, we wait only for 0.5 Baud cycle since the Stop bit is the last one and we do not care if it lasts long or short.



The next job now is to come up with time delays for 1.25, 1.0, and 0.5 Baud cycles. Since we have not studies on Timer block of 16F877, we keep using the instruction cycle based time dealy as we did in Chapter 3 Instruction Sets. With 20MHz oscillation, each instruction takes 0.2 µs. Therefore, 1.25 Baud cycles correspond to 325 instruction cycles, 1.0 Baud cycle to 260 instruction cycles, and 0.5 Baud cycle to 130 instructions. As before, we are going to use `decfsz` instruction to have repetition loop, the basic building block for instruction cycles is k*3, where K is the number of repetition to get a desired time delay, or Baud cycle. Therefore, we can have the following relationships for k's and corresponding Baud cycles:

Number of Baud cycles	Time in [µs]	Number of Instructions needed	Formula	Repetition Count K's
1.25	65	325	325=107*3 + 4	K125=107 (6Bh)
1.0	52	260	260=84*3 + 8	K100=84 (54h)
0.5	26	130	130=42*3 + 4	K050=42 (2Ah)

Then, we can make a subroutine to obtain three Baud cycle time delays. `BC050`, `BC100`, and `BC125`.

```
;===SUBROUTINES ====
BC050 movlw        K050         ;The value 0x2A must be declared at top
      goto         SAVE

BC100 movlw        K100         ;The value 0x54 must be declared at top
      goto         SAVE

BC125 movlw        K125         ;The value 0x6B must be declared at top
      goto         SAVE

SAVE  movwf        Kount
RPT   decfsz       Kount
      goto         RPT
      return
; End of Subroutines
```

The following is an example code section of Start bit detection. When Start bit is not detected, it waits for 0.5 Baud cycle, and then resume its detection of the bit:

```
WAITSTART
      btfss  PORTD, RXPin             :Is Start bit arrived?
      goto   RECEPTION                ;Yes, Start bit detected,
                                    ;go to reception mode

      call  BC050                   ;No.  Then 0.5*BC second wait
      goto  WAITSTART               ;check for START bit again
```

Since we have desired time delays in terms of Baud cycles, we can now write a code to read the serially transmitted data from the keyboard. But before, we proceed with code, there is one minor thing to discuss. Since in serial communication we receives LSB first and MSB last, we have to change the order of the bits we receive. The instruction `rrf` will do the job for us. `rrf` rotates the bits in a file register to the right. In this rotation the carry bit (Carry bit is 0th bit of STATUS register, STATUS<0>) is included. Specifically, first we clear the Carry bit, and rotate right one bit so that the MSB in the file register becomes 0. Then, we read the first bit of the received data. If the bit is High, the we set the MSB bit of the file register, otherwise, we keep the MSB. Then, we clear the Carry bit again and rotate a bit to the right for the detection and determination of the second bit. The following code section shows how it reads one bit of data and rotate to the right:

```
RECEPTION
            call  BC125             ;After Start, wait for 1.25*(BC)
RXNEXT      bcf   STATUS, CARRY     ;Clear the Carry Bit
            rrf   Rreg              ;rotate to the right. so Rreg<8>=0
            btfsc PORTD, RXPIN      ;check the bit
            bsf   Rreg, MSB         ;1? Then set the MSB -->Rreg<8>=1
            call  BC100             ;0? Do nothing --->Rreg<8>=0
                                    ;Wait for the next bit reading
```

This "clearing Carry bit, Rotate right a bit, read a received bit, and set/clear MSB, then rotate the file register" continues for 8 times to read the 8-bit data stream.

After this we wait for a Stop bit to terminate the reception. When Stop bit is detected we move the content of the data, stored in a file register, to **W** register to retransmit.

```
WAITSTOP
            btfss PORTD, RXPIN      ;Stop bit received?
            goto  WAITSTOP          ;Not yet.
; Save the content to TXREG for an echo
            call  BC100             ;Yes. Then give 1*BC wait
            movf  Rreg,0            ;And move the content: Rreg -> W
```

Transmission: Next job is to send the received data back to the monitor of the computer. This part is actually simpler than the reception part. Before sending the data, we send the Start bit by setting the TX pin to low for one BC seconds:

```
      bcf   PORTD, TXPin
;delay for 1*(BC) cycles
      call  BC100
```

Then, we send the data, one bit at a time with the same 1 BC second delay, Txpin low when the bit is 0, and Txpin high if the bit is 1. Since we have to send LSB first, we have to do the similar rotation, which includes the carry bit. The idea is to move the LSB of the file register where the data is stored to Carry bit, and check the status of the bit. If the carry bit is 1, then we send 1 to Txpin, for 0, then 0 to Txpin, for 1 BC seconds. See below for a code section of one bit (LSB first) transmission:

```
      bcf    STATUS, CARRY     ;Clear carry
      rrf    Treg              ;Now Carry = Treg<0>, LSB of the data
      btfsc  STATUS,CARRY      ;Is Carry=1?
      bsf    PORTD, TXPin      ;Then send High to TXpin
      btfss  STATUS,CARRY      ;IS Carry=0?
      bcf    PORTD, TXPin      ;Then send Low to Txpin
      call   BC100             ;The pulse width of 1.0 Baud Cycle
```

We do the above process of 8 times to send all 8 bits of the data. After data transmission, we send Stop bit to terminate the transmission session:

```
      bsf    PORTD, TXPin
      call   BC100                   ;High for 1 BC for STOP bit
```

Hardware and Example Code

One hardware set-up we need is to add one more set of RX and TX lines. If you examine the minimum hardware we implemented in Chapter 4. Since MAX232 supports two set of serial communication level conversions, and only one set was used for serial communication using USART, we can use the other set for serial communication with software. As you can see from the schematic diagram below, the RX and TX ports for software implementation of serial communication are connected to RD3 and RD2 ports, respectively. As said before, any port pins can be connected for this software based serial communication.

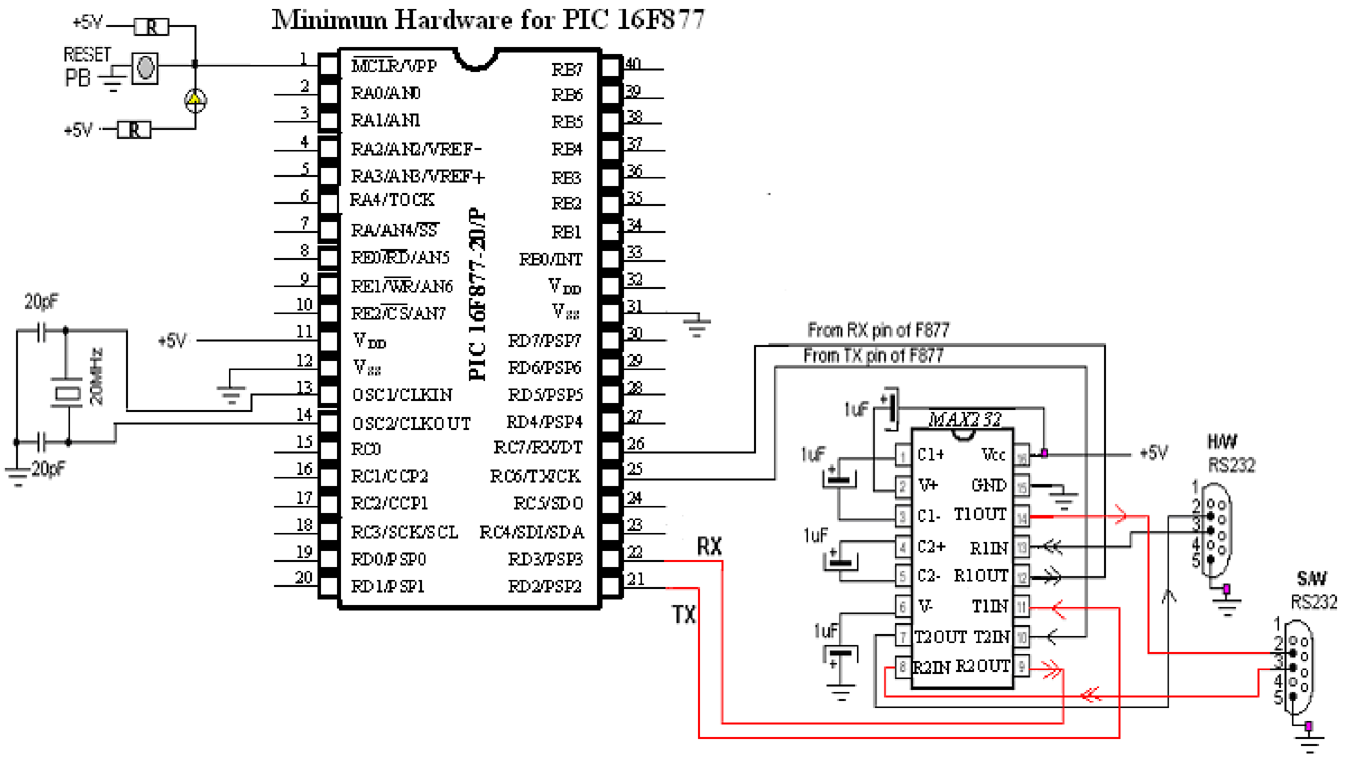


Fig. 23 Software based Serial Communication

Now we are ready to have an example code for receiving a byte from the keyboard and echoing it on the monitor. So the code functions as the echoing program we studied before, but with USART module. Read the comment lines for detailed explanation of the code.

```
;echo-v2.asm
;This program is asynchronous communication using software method
;In other words, using this, any two pins of F877 can be used
;for RX and TX pins, respectively.
;
;This program reads(receives) RS-232 transferred data and
;echo it by transmitting the data
;
;F = 20 MHz
;B = Baud Rate
;For B=19200, one Baud cycle (BC) is about 52uS
;
;
;RECEIVE MODE
;RX pin is polled approximately every 0.5*(BC) seconds, i.e., 26uSec for 19200
;to detect the START bit (which is LOW when detected)
;If a START bit is detected, then the first data bit is checked for after
1.25*(BC) seconds
;From then on, the other data bits are checked every (BC) seconds.
;The data received is assumed to be transmitted LSB first (Normal mode)
;
;TRANSMIT MODE
;First START bit is sent by setting the TX pin to LOW for (BC) seconds
;And, from then on, the TX Pin is Set/Cleared corresponding to the data bit
;every (BC) seconds.
;
;
; By the way, 1 instruction cycle = f (CLK)/4
; In 20 MHZ clock case: 1 instruction cycle takes (1/20)*4=0.2 uS
;
;Terminal set up: 8N1 19200
;
;
; Number of instruction cyles for 1(BC) = (52/0.2)=260 Cycles
;            260   =84*3 + 8 ====>K100 = 84 (0x54)
; Number of instruction cyles for 0.5*(BC) = (0.5)(52/0.2)=130
;           130 = 42*3 + 4 ====>K050 = 42 (0x2A)
; Number of instruction cyles for 1.25*(BC) = (1.25)(52/0.2)=325
;            325 = 107*3 + 4 ====>K125 = 107 (0x6B)
;
;Asynchronous mode
;
;TX Pin = RD2
;RX Pin = RD3
;
       list P = 16F877

STATUS      EQU   0x03
CARRY       EQU   0x00        ;Carry bit STATUS<0>
TRISD       EQU   0x88
PORTD       EQU   0x08
K100        EQU   0x54        ;for 1BC delay
K050        EQU   0x2A        ;for 0.5 BC delay
K125        EQU   0x6B        ;for 1.25 BC delay
TXPIN       EQU   0x02        ;RD2
RXPIN       EQU   0x03        ;RD3
MSB         EQU   0x07

;
```

```
;RAM for RX/TX Buffer
      CBLOCK      0x20
            KEYIN       ;keyin from a keyboard
            Rreg        ;Temporary register for reception
            Treg        ;Temporary register for transmission
            Bitcount    ;data bit count
            Kount       ;Delay count (number of instr cycles for delay)
      ENDC


;==========================================================
      org         0x0000            ;line 1
      goto        START       ;line 2 ($0000)

;=======================================================
      org         0x05
START

      banksel     TRISD
; Port setting (1 for input and 0 for output)
;RD2 - TXPin (out) cleared
;RD3 - RXPin (in)  set
; 0000 1000

      movlw       0x08
      movwf       TRISD

TALK                          ;Start of the session
      banksel     Rreg
      clrf        Treg        ;clear the TX and RX buffers (registers)
      clrf        Rreg

; Check for START bit
WAITSTART
      btfss       PORTD, RXPin
      goto        RECEPTION
      call        BC050       ;0.5*BC cycle wait
                              ;Then, check for START bit again
      goto        WAITSTART
;
;
;RX for Data bits begins here
RECEPTION
      call        BC125       ;Delay for 1.25*(BC) cycles
      movlw       0x08
      movwf       Bitcount    ;8 data bits
RXNEXT
      bcf         STATUS, CARRY     ;Clear the Carry Bit
      rrf         Rreg              ;rotate to the right
      btfsc       PORTD, RXPIN      ;check the bit
      bsf         Rreg, MSB         ;1? Then set the MSB
      call        BC100
      decfsz      Bitcount          ;Repeat 8 times
      goto        RXNEXT
;Check for STOP Bit
WAITSTOP
      btfss       PORTD, RXPIN
      goto        WAITSTOP
; Save the content to Treg for an echo
      call        BC100
      movf        Rreg,0            ;Rreg -> W
      movwf       Treg              ;W --> Treg  For transmit back

;TX ROUTINE
```

```
      movlw       0x08
      movwf       Bitcount          ;8 data bits
;send a START bit
      bcf         PORTD, TXPin
;delay for 1*(BC) seconds
      call        BC100
TXNEXT
      bcf         STATUS, CARRY
      rrf         Treg              ;LSB first mode (normal)
      btfsc       STATUS,CARRY
      bsf         PORTD, TXPin
      btfss       STATUS,CARRY
      bcf         PORTD, TXPin
      call        BC100
      decfsz      Bitcount          ;Repeat 8 times
      goto        TXNEXT
;send STOP bit
      bsf         PORTD, TXPin
      call        BC100
;wait until the end of STOP bit
      goto        TALK                    ;Get the Keyin again


;
;===SUBROUTINES ====
BC050
      movlw       K050        ;0x2A
      goto        SAVE
BC100
      movlw       K100        ;0x54
      goto        SAVE

BC125
      movlw       K125        ;0x6B
      goto        SAVE

SAVE  movwf       Kount
RPT   decfsz      Kount
      goto        RPT
      return

;End of Subroutine ===

;End of code
        END
```

Of the example code above, in many applications, only the part for transmission of serial data may be needed for displaying results and others.