Receiving RS-232 Serial Data

Introduction. This application note presents a simple program for receiving asynchronous serial data with PIC microcontrollers. The example program, written using TechTools assembly language, displays received bytes on a bank of eight LEDs.

Background. Many controller applications involve receiving data or commands from a larger system. The RS-232 serial port is a nearly universal means for this communication. While the PIC lacks the serial receive function found on some more expensive chips, it can readily be programmed to receive serial data.

A byte of serial data is commonly sent as a string of 10 bits; a start bit, eight data bits, and a stop bit, as shown in figure 1 below. The start and stop bits help the receiver to synchronize to the incoming data bits. In some cases, a serial transmitter will lengthen the stop bit to 1.5 or 2 times the duration of the data bits in order to ensure proper sync under noisy conditions.

The speed of a serial transmission is expressed in baud or bits per second (bps). Since a complete transmission is 10 bits long, the number of bytes per second is one-tenth the baud rate. A 1200-baud signal conveys 120 bytes per second. The bit duration is 1 second divided by the baud rate. For instance, each bit of a 1200-baud signal is 833 microseconds long.

RS-232 is an electrical standard for signals used in serial communication. It represents a binary 1 with a level of -5 to -15 volts, and a 0 with +5 to +15 volts. In order for a 5-volt device like the PIC to interface with this signal, additional circuitry must convert the RS-232 signal to logic levels. The 1489 quad line receiver used in the circuit (see schematic, figure 3) can convert four RS-232 signals to 5-volt logic levels; the example circuit uses only one section of the device.

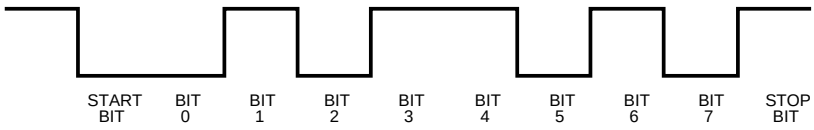

Figure 1. A byte of serial data. The byte depicted is 01011010, the ASCII code for Z.

Where cost or space is a problem, the PIC can accept the RS-232 signal through a 22k resistor, as shown in the inset to figure 3. The resistor limits the input current, while the PIC's internal clamping diodes (intended to protect against static electricity) clip the voltage to logic levels. The resistor method does not invert the RS-232 signal, so three minor changes to the program are required as shown in comments to listing 1. The method also gives up the noise rejection built into the 1489, and should not be used in noisy environments or over long cable runs.

The example presented here does not use any of the RS-232 handshaking lines. These lines help when a fast computer must communicate with (for instance) a slow printer. When the receiving device does not use the handshaking lines, it is necessary to loop them back as shown in figure 2. That way, when the computer asks for permission to send, the signal appears at its own clear-to-send pin. In effect, it answers its own question.

Although it is the most common, RS-232 is not the only serial-signaling standard. RS-422 is becoming increasingly popular because of its resistance to noise, high speed, long allowable wire runs, and ability to operate from a single-ended 5-volt power supply. Since the only difference between RS-232 and RS-422 is the electrical interface, conversion requires only the substitution of an RS-422 line receiver chip.

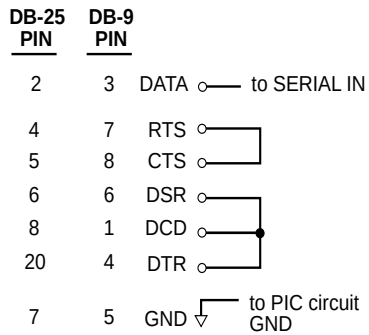

Figure 2. Hookups for standard 9- and 28-pin connectors. Connecting RTS to CTS disables normal handshaking, which is not used here.

How it works. The example program in listing 1 is a no-frills algorithm for receiving serial data in the popular N81 format; i.e., no parity bit, eight data bits, and one stop bit. Listing 2 is a BASIC program for sending individual bytes to the circuit.

Listing 1 begins by setting up the input and output bits. It then enters a loop waiting to detect the start bit. Once the start bit is detected, the program waits one-half bit time and checks to see whether the start bit is still present. This helps ensure that the program isn't fooled by a noise burst into trying to receive a nonexistent transmission. It also makes sure that subsequent bits are read during the middle of their time slots; another precaution against noise.

Once it detects and verifies the start bit, the program enters another loop that does the actual job of receiving the data. It works like this:

- Wait one bit time.
- Copy input bit to carry bit.
- Rotate the receive byte right.
- Decrement the bit counter.
- Is the counter zero?
  - > No, loop again.
  - > Yes, exit the loop.

If you are unfamiliar with the rotate right (rr) instruction, you may not see how the input bit gets from the carry bit into the receive byte. Performing an rr on a byte moves its bits one space to the right. Bit 7 goes to bit 6, bit 6 to bit 5, and so on. Bit 0 is moved into the carry bit. The carry bit moves into bit 7.

Once the byte is received, the program waits a final bit delay (until the middle of the stop bit), copies the received byte to the output port to which the LEDs are connected, and goes back to the beginning to await another start bit.

The program can be set up for most standard data rates. The table lists PIC clock speeds and values of the bit time constant *bit_K* (declared at the beginning of listing 1) for a wide range of common rates. For other combinations of clock speed and data rate, just replace the delay

routines with ones that provide the appropriate timing. The footnote to the table gives general guidance.

One final hardware note: Although timing isn't overly critical for receiving this type of serial data, resistor/capacitor timing circuits are inadequate. The PIC's RC clock is specified to fairly loose tolerances (up to ±28 percent) from one unit to another. The values of common resistors and capacitors can vary substantially from their marked values, and can change with temperature and humidity. Always use a ceramic resonator or crystal in applications involving serial communication.

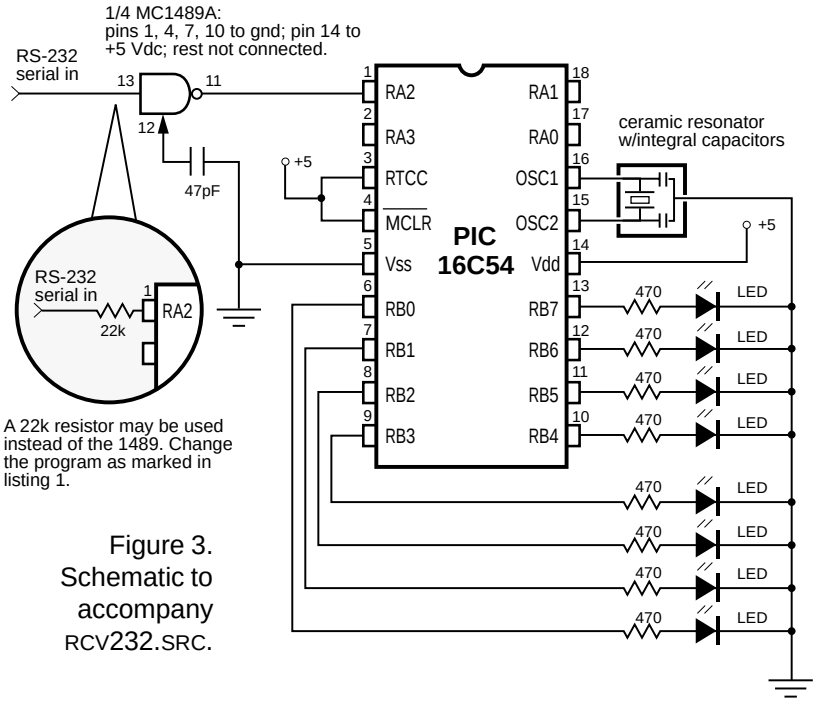

Figure 3. Schematic to accompany RCV232.SRC.

Program listing. This program may be downloaded from our Internet ftp site at ftp.tech-tools.com. The ftp site may be accessed directly or through our web site at http://www.tech-tools.com.

Values of Timing Constant Bit_K for Various Clock Speeds and Bit Rates

Clock Frequency	Serial Bit Rate (bit time)						
	300 (3.33 ms)	600 (1.66 ms)	1200 (833 µs)	2400 (417 µs)	4800 (208 µs)	9600 (104 µs)	19,200 (52 µs)
1 MHz	206	102	50	24	Ñ	Ñ	Ñ
2 MHz	Ñ	206	102	50	24	Ñ	Ñ
4 MHz	Ñ	Ñ	206	102	50	24	Ñ
8 MHz	Ñ	Ñ	Ñ	206	102	50	24

Other combinations of clock speed and bit rate can be supported by changing the bit_delay and start_delay subroutines. The required bit delay is $\frac{1}{\text{bit rate}}$. For example, at 1200 baud the bit delay is $\frac{1}{1200}$ = 833µs. The start delay is half of the bit delay; 416µs for the 1200-baud example. Calculate the time delay of a subroutine by adding up its instruction cycles and multiplying by $\frac{4}{\text{clock speed}}$. At 2 MHz, the time per instruction cycle is $\frac{4}{2,000,000}$ = 2µs.

;PROGRAM: RCV232

```
; This program receives a byte of serial data and displays it on eight LEDs
; connected to port RB. The receiving baud rate is determined by the value of the
; constant bit_K and the clock speed of the PIC. See the table in the application
note
; (above) for values of bit_K. For example, with the clock running at 4 MHz and a
; desired receiving rate of 4800 baud, make bit_K 50.

; Remember to change device info if programming a different PIC. Do not use RC
; devices. They are not sufficiently accurate or stable for serial communication.

                device      pic16c54,xt_osc,wdt_off,protect_off
                reset       begin

bit_K           =           24              ; Change this value for desired
                                            ; baud rate as shown in table.
half_bit        =           bit_K/2
serial_in       =           ra.2
data_out        =           rb

; Variable storage above special-purpose registers.
                org         8
delay_cntr      ds          1               ; counter for serial delay routines
bit_cntr        ds          1               ; number of received bits
rcv_byte        ds          1               ; the received byte

; Org 0 sets ROM origin to beginning for program.
                org         0
```

```
; Set up I/O ports.

begin       mov     !ra, #00000100b     ; Use RA.2 for serial input.
            mov     !rb, #0             ; Output to LEDs.

:start_bit  snb     serial_in           ; Detect start bit. Change to sb
                                        ; serial_in  if using 22k resistor
                                        ; input.

            jmp     :start_bit          ; No start bit yet? Keep watching.
            call    start_delay         ; Wait one-half bit time to the
                                        ; middle of the start bit.

            jb      Serial_in, :start_bit ; If the start bit is still good,
                                        ; continue. Otherwise, resume
                                        ; waiting.

                                        ; Change to jnb Serial_in,
:start_bit
                                        ; if using 22k resistor input.

            mov     bit_cntr, #8        ; Set the counter to receive 8 data
                                        ; bits.

            clr     rcv_byte            ; Clear the receive byte to get
                                        ; ready for new data.

:receive    call    bit_delay           ; Wait one bit time.
            movb    c,Serial_in         ; Put the data bit into carry.
                                        ; Change to movb  c,/Serial_in if
                                        ; using 22k resistor input.

            rr      rcv_byte            ; Rotate the carry bit into the
                                        ; receive byte.

            djnz    bit_cntr,:receive   ; Not eight bits yet? Get next bit.
            call    bit_delay           ; Wait for stop bit.
            mov     data_out, rcv_byte  ; Display data on LEDs.

            goto    begin:start_bit     ; Receive next byte.

; This delay loop takes four instruction cycles per loop, plus eight instruction cycles
; for other operations (call, mov, the final djnz, and ret). These extra cycles become
; significant at higher baud rates. The values for bit_K in the table take the time
; required for additional instructions into account.

bit_delay   mov     delay_cntr,#bit_K
```

```
:loop       nop
            djnz    delay_cntr, :loop
            ret

; This delay loop is identical to bit_delay above, but provides half the delay time.

start_delay mov     delay_cntr,#half_bit
:loop       nop
            djnz    delay_cntr, :loop
            ret
```

BASIC Program for Transmitting Bytes via COM1

```
10  REM  Open the serial port com1. Substitute the desired baud rate
20  REM  for 9600 in this line. The parameters CD0, CS0, DS0, and OP0
30  REM  serve to disable hardware handshaking. They may be omitted if
40  REM  these lines are looped back as shown in figure 2.
50  OPEN "com1:9600,N,8,1,CD0,CS0,DS0,OP0" FOR OUTPUT AS #1
60  CLS
70  REM  At the prompt, enter a value between 0 and 255 representing a
80  REM  byte of data to send out the serial port.
90  INPUT "ASCII code to send:  ", A%
100 PRINT #1, CHR$(A%);
110 GOTO 60
120 END
```