A Virtual Peripheral UART: Implementing RS232C Serial Protocol

Introduction

This application note presents programming techniques for receiving and transmitting asynchronous data with the SX microcontroller via RS232C using the SX's internal interrupt feature to allow background operation of the code as a virtual peripheral. This implementation uses the Parallax demo board and takes advantage of their *SX demo* software user interface features to allow the SX to act as a Universal Asynchronous Receiver Transmitter (UART) which communicates simply and directly with a personal computer.

How the circuit and program work

The circuit is a simple implementation of RS232C and requires the use of only two pins on the SX: an RX pin (Port A, bit 2) for receiving serial data and a TX pin (Port A, bit 3) for transmitting data. Other RS232C functions, such as handshaking signals CTS, RTS[1] are bypassed in order for this basic two-pin configuration to work. Various signals from the RS232C connector[2] must be jumpered together: namely, the RTS and CTS pins and, separately, the DSR, DCD, and DTR pins. To enable conversion between RS232C line levels (±5−15 volts) and the 0-5 volt levels used by the SX, a line-level converter chip (such as a LT1181 or MAX232 charge pump) is used.

The code in this example implements the popular N-8-1 data format (No parity (N), eight (8) data bits, and one (1) stop bit), and makes use of the SX's internal RTCC rollover interrupt[3] to operate at a transfer rate of 19.2k baud. Since operation of the routine is timing-critical, the period between interrupts must remain constant because it is inversely proportional to the baud rate of the UART. The duration of this period, in seconds, depends upon oscillator frequency, the RTCC pre-scaler (i.e. what clock divider we are using), how many instruction cycles there are between interrupts (the value loaded into the W register upon leaving the interrupt), and whether the SX is operating in turbo mode[4]. The baud rate depends, in turn, upon the period of each interrupt and also how many times the interrupt executes between each bit sent or received (as determined by the values in *tx_divide* and *rx_divide*), as shown below:

period (sec) = mode * prescaler * RETIW value / frequency, where mode=1 (turbo) or =4 (normal)

Baud rate = 1 / (period * # *tx/rx_divide* value)

---

[1]When the host system has data that is Ready-To-Send, it asks the receiving device for permission to send by pulling the RTS line high. It then awaits a Clear-To-Send acknowledgment which involves the receiving device pulling the CTS line high when it is ready to receive data. By shorting these two pins together, the host system (in this case the PC) essentially answers its own RTS request and immediately sends the data. For more information on handshaking, see http://www2.sco.com:1996/HANDBOOK/RTS_CTS.html

[2]On a **9-pin (DB-9) connector:** pins 7 & 8 (RTS & DTS) must be jumpered and,separately, pins 6,1 & 4 (DSR, DCD, and DTR) must also be wired together. On a **25-pin (DB-25) connector:** pins 4 & 5 (RTS & DTS) must be jumpered and, separately, pins 6,8 & 20 must also be wired together.

[3]The interrupt is triggered each time the RTCC rolls over (counts past 255 and restarts at 0). By loading the RTCC Prescaler in the OPTION register with the appropriate value, the RTCC count rate is set to some division of the oscillator frequency (in this case they are equal), which is the external 50 MHz crystal in this case. At the close of the interrupt sequence, a predefined value (255-period desired) is loaded into the W register using the RETIW instruction which determines the period of the interrupt in RTCC cycles.

[4]Most instructions (except calls, returns, jumps, skips and the *IREAD* command) execute in 1 instruction cycle which, in turbo mode, corresponds to a single clock count. In compatible (non-turbo) operation mode, this corresponds to four external clock counts.

So, in turbo mode, with a 50 MHz oscillator, no prescaler (i.e. prescaler=1), and tx/rx_divide = 16 interrupt passes per data bit sent/received,

Baud rate = 1 / (period * 16) = 50 MHz / (1 * 1 * RETIW value * 16) = 3.125 x 10^6 / RETIW value

At a baud rate of 19.2k baud, we get: RETIW value = 3.125 x 10^6 / 19200 = 163

The value of 255-163=92 must be loaded into W and seeded into the RTCC at the end of the interrupt sequence in order for the UART to function properly in terms of timing issues. By subtracting the value from the RTCC maximum, there is a shortcut method used in this code to set the correct timing (using RETIW). The RETIW value has a minimum threshold value (see *Modifications and further options: selecting different baud rates*).

The UART interrupt code starts with the transmit routine. It begins by selecting the *serial* register bank, and then increments, *tx_divide*, the count of the number of passes through the interrupt routine since the last bit operation, checking if it has reached 16 (*baud_bit*=4), which corresponds to a serial 19.2k baud[5] transmission rate. It then checks to see if any data is currently being transmitted, and if so, it rotates the 10-bit *tx_high* + *tx_low* transmission buffer (1 start bit + 8 data bits + 1 stop bit) to prepare the next data bit, sends it out on the TX pin, and decrements, *tx_count*, the number of bits are left to send.

To access the transmission routines from the main program code, the byte to be sent is loaded into W, and then a call is made to the s*end_byte* subroutine. *Send_byte* first waits until any earlier transmissions are complete. Then the byte to be sent is negated (because inverse logic is used on the RS232 port) and given a start bit (the high bit of *tx_high* is set). The stop bit is taken care of by the *CLC* instruction in the transmit code (by clearing the trailing bits which follow the last data bit).

The *send_hex* subroutine converts a hex value stored in *number_low* into ascii form and sends it out through the UART. The *send_string* subroutine requires an address[6] in W that points to a text string which is terminated by a 00h as the last character of the string, and transmits the entire string out through the UART.

Receiving data is slightly more involved, since the best precaution against noise and timing variations is to read bits in the middle of their time slots. The receive routine starts by loading the status of the RX line into the carry flag. It then tests to see if a byte is currently being received. If not, it checks the incoming bit to see whether it is a start bit (i.e. has it dropped low?), and if so, loads the bits-to-receive counter, *rx_count*, with 9 (8 data bits and 1 stop bit) and loads *rx_divide*, the number of interrupt passes to wait before reading the next bit, with a value which corresponds to one and a half bit times[7] in order to skip over the start bit, and to begin reading data bits in the middle of their bit times.

If a byte *is* currently being received, it jumps to *:rxbit* and tests whether the middle of the next bit time has been reached by decrementing and testing *rx_divide* for 0. If true, it resets *rx_divide* to wait for the next bit, and rotates the bit read into the *rx_byte* receive buffer. When the number of bits left to receive, *rx_count*, reaches zero, indicating that the last bit received was a stop bit (which doesn't get rotated into the *rx_byte* buffer), the routine flags (using *rx_flag*) that the new byte is stored in the receive buffer and then begins awaiting the next data byte.

To access the receive routines from the main program code, a call is made to the g*et_byte* procedure, which returns with the incoming data byte value stored in *byte*. G*et_byte* simply monitors *rx_flag* until it signals

[5]To select different serial rates, see the *modifications* section

[6]The address of the text string must be within the lower half of the first page of memory.

[7]This value is calculated by the formula: (2^baud_bit)*1.5 + 1, the "+1" portion being due to the location of the are-we-receiving? test and branch instructions, which precede the *:rxbit* code that actually receives a data bit.

that a byte has been received. It then clears *rx_flag* (to prepare for the next byte), and loads the received byte from the *rx_byte* buffer.

The *get_hex* subroutine allows the reading of from 0-4 ascii hex digits which are receieved via the serial port and converted to a 16 bit hex value stored in *number_low* & *number_high*.

Modifications and further options

To select **different baud rates**, a few factors must be taken into account. The first is that the length of the entire interrupt routine (including the UART and all other interrupt functions) must be less than the value loaded into W before returning from the interrupt with the RETIW command. If this is not so, the code will simply malfunction. Ideally, this value, which corresponds to the number of instruction cycles between interrupts, is considerably longer than the length of the entire interrupt procedure, otherwise most of the SX processing time will be used up on constant interrupts, allowing a small slice of CPU time for main program code execution.

The simplest method to adjust baud rate is to select the *baud_bit* parameter value within the range from 1-7, which corresponds to the following baud rates: 1=153600, 2=76800, 3=38400, 4=19200, 5=9600, 6=4800, 7=2400. For slower rates, either a slower crystal can be used, or the RTCC prescaler can be changed by loading a different value into the !OPTION register. For quicker or custom rates, alterations can be made to *int_period* (along with *baud_bit*) to allow the appropriate bit times. If baud bit is changed, the *start_delay* value must also be adjusted appropriately, where *start_delay* = (2^*baud_bit*) * 1.5 + 1.

If it is important that no incoming data be lost (because the main program code hasn't the time to be continually checking *rx_flag*), then **handshaking** may be used to control the flow of incoming data. This will require two more SX pins, one each for the RTS and CTS signals. The receive routine should then check the RTS line (rather than look for a start bit) while awaiting a new byte. Once it recognizes a valid Ready To Send signal (i.e. a high on the corresponding port pin), it sets the Clear To Send line to valid (i.e. high), loads the appropriate initial bit times and counts in *rx_divide* and *rx_count*, and begins receiving the byte as normal, toggling the CTS line off once the stop bit is received, in which case it starts waiting for the next RTS signal.