Serial Communications (RS232) in background

Submitted by:

Glenn Clark - Protean Logic Inc.

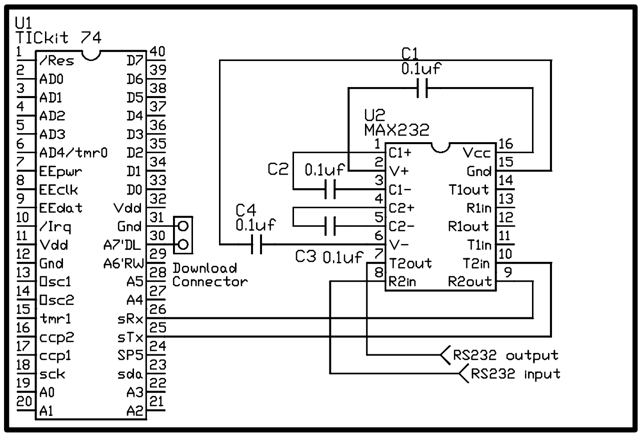
Introduction

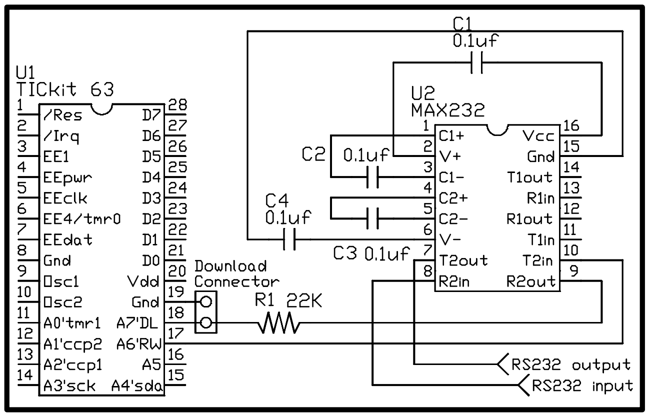
The RS232 standard for serial communications has proven to be a very versatile standard. This basic protocol standard has found uses in countless micro-controller applications. Earlier versions of micro-interpreters, like the TICkit 62 or Parallax's BS2, have provided basic RS232 support through simple emulation functions. This works well for transmitting data, but presents a problem for receiving data. The emulation functions can only see RS232 data when the function is actually being executed. If the data reception is completely asynchronous, it is very likely some part of the data will be missed while the processor is doing something else. Designers had to become very creative to invent ways of synchronizing with the data transmitter. However, there are still a large number of projects which simply require the ability to receive RS232 data independent of program execution.

The TICkit 63 and 74 solve this problem by using an internal hardware block called the SCI (Serial Communications Interface). This hardware contained inside the raw micro-controller can detect and buffer received serial data up to 3 bytes while the processor is doing other things. Provided the main process can respond to interrupts or periodically poll the condition of the receive buffer, large amounts of data can be received and processed completely independent of the process running on the TICkit. This opens up many project possibilities in communication or control for micro-interpreters.

Hardware Considerations and Wiring

The SCI hardware requires that received and transmitted serial data go through specific pins. On the TICkit 74 there are pins dedicated to this function. On the TICkit 63 the two SCI pins connect to general purpose pins A7 and A6. This presents a little bit of a problem for the TICkit 63 because pin A7 is the primary download pin and pin A6 is the Read/Write output for the buss functions. Fortunately, these problems can be overcome for the TICkit 63 without too much difficulty.





If you will be using the SCI hardware on a TICkit 63 you should remap the debug and console pins away from A7. The assignment of the pins does not matter but the following example remaps them to pin A4.

```
FUNC none main
BEGIN
    ; remap the debug and console pins away from pin_a7
    rs_bdparam_set( rs_invert | rs_19200 | pin_a4 )
    rs_conparam_set( rs_invert | rs_19200 | pin_a4 )
    debug_on()
```

```
        ; You should continue to have the ability to download through A7
        ; just in case the program gets corrupted, but you can download
        ; program modifications, debug your program, and receive
        ; console I/O through pin_a4 at this point and on.

        ; whenever the processor resets, the initial console connect
        ; occurs then execution at BEGIN of main occurs. When
        ; debug_on() occurs, another console connect occurs but on
        ; pin_a4. If no debug console is found, execution continues.
    ENDFUN
```

Controlling the SCI and getting data

The SCI hardware is controlled through some registers. By setting certain bits in these registers you select the baud rate, enable the SCI, clear error conditions, and indicate if an interrupt should be generated when an SCI event takes place.

The registers used are:

Register Name	Description	Reading Functions	Writing Functions
sci_rxsta	SCI receiver Status	sci_rxsta_get	sci_rxsta_set
sci_txsta	SCI transmitter Status	sci_txsta_get	sci_txsta_set
sci_baud	SCI Baud Rate Divisor	sci_baud_get	sci_baud_set
sci_reg	SCI receive and xmit data buffer	sci_reg_get	sci_reg_set

An SCI polled method of serial input

Polling is similar to the emulated RS232 methods for receiving data in the sense that you only read the hardware when a function executes. It is more powerful, though, because the hardware can receive up to three bytes before being read without loosing any data. In most cases, this will be plenty of buffering as most micro-control projects have a very fast control loop. To give you a sense of how much time the hardware method gains over the emulated receive method consider the following timing.

An emulated mode has the time between the last data bit sample and the falling edge of the start bit to process data without missing any characters. At 9600 baud, a stop bit is 104 micro seconds, because the last data bit is sampled somewhere in the middle of that bits time, you actually have somewhere between 208 and 104 microseconds to do whatever processing and get back into the serial reception program. If you have stop bit checking enabled, you only have between 104 microseconds and 0 microseconds to do processing. This is very, very little time to do anything since it takes a minimum of 50 microseconds to execute the simplest line of FBasic.

Now, if you are using the SCI hardware, from the point in time that a byte is received into the SCI data register, you have the time of two more serial byte transmissions to read the first byte without loosing the first byte through an overrun error. At 9600 baud, that comes out to be 2008 microseconds which is somewhere between 10 and 20 times the read window of a strictly emulated approach. In most cases, this is plenty of time to receive examine the byte compare it against certain values and alter the program's flow for conditional input parsing.

```
    ; Uses the SCI hardware to receive rs232 data
    FUNC byte rs_sciget
    BEGIN
        REP ; wait until a character is received
            IF b_test( 0y00000110b, sci_rxsta_get())
                ; clear the mess if an overrun or framing error is detected
                sci_rxsta_set( sci_rxstat_port ) ; clear receive status
                sci_rxsta_set( sci_rxstat_cenbl | sci_rxstat_port )
            ENDIF
    UNTIL b_test( int_flag_rx, int_flag_get())

    =( exit_value, sci_reg_get())
ENDFUN

; wait for a stream that matches the string parameter
FUNC none wait_string
     PARAM word str_pntr
     LOCAL word str_pntrtemp
BEGIN
     =( str_pntrtemp, str_pntr )
     =( each_strchar, ee_read( str_pntrtemp ))
     WHILE each_strchar
         ;=( each_rsinchar, rs_receive( 0b, 0b, each_rsinerror))
         =( each_rsinchar, rs_sciget())
         IF ==( each_rsinchar, each_strchar )
             ++( str_pntrtemp )
             =( each_strchar, ee_read( str_pntrtemp ))
         ELSE
             =(str_pntrtemp, str_pntr )
             =( each_strchar, ee_read( str_pntrtemp ))
         ENDIF
     LOOP
ENDFUN
```

The routines above demonstrate a simple polling routine for the SCI that is used by an input stream matching function. The matching function alone does not require as sophisticated approach and could easily be handled by the BS2 SERIN function or the TICkit 62's rs_scanf() function. However, if program flow needs to be controlled on the basis of fields within an input stream. The SCI speed is required. This allows the first fields in a string to influence how the later fields in the stream are handled without loosing any of the data.

If you write an interrupt handling routine, a special subroutine does the polling and either processes the received serial data or stores it for later processing. What is special about the interrupt routine is that it is called automatically by the SCI hardware whenever their is data to be read. This frees the main loop from having to pool the hardware.

An SCI interrupt routine

```
FUNC none global_int    ; Internal Global Interrupt
BEGIN
    ; this routine is used to capture serial data
    IF rec_state
        =( xmit_string[ rec_state ], sci_reg_get())
        ++( rec_state )
    ELSE
        IF ==( sci_reg_get(), header_value )
            =( last_head, rtc_timer_get())
            ++( rec_state )
        ELSE
            IF b_test( sci_rxsta_get(), sci_rxstat_over )
                sci_rxsta_set( sci_rxstat_port )
                sci_rxsta_set( sci_rxstat_cenbl | sci_rxstat_port )
            ENDIF
        ENDIF
    ENDIF
ENFUN
```

This is an example of an SCI interrupt routine. What happens in this program, is that the interrupt routine is automatically flagged when a character is received. As soon as the current executing token finishes, which is normally very soon, the interrupt routine executes by virtue of an interrupt call. An interrupt call is just like a subroutine or function call. Once the interrupt routine is finished, your program will resume execution at the point immediately after the interrupt call. To the main program, it seems as though nothing happened except for a little loss of time.

Using interrupt routines in this way, you can either process the data as it comes in, or store it for later analysis. Either way, you can overcome the 3 byte buffer limitation of the SCI hardware. A common buffering scheme is called a ring buffer where two pointers, a read pointer and a write pointer, are maintained in a circular buffer. The ap-note will not go into the ring buffer concept, but it is a useful way extending the buffer size without adding any additional management concerns for the main program.

clears the overrun flag if it set. The array xmit_string[] holds the received data. The main body of the program uses the following code to initialize the SCI. The main body monitors the value of rec_state to determine if a full block has been received. If rec_state is equal to 17, then the packet has been fully received and can be analyzed.

```
; initialize serial port
sci_baud_set( 255b )   ; 1200 baud
sci_txsta_set(0b)
int_mask_set( int_flag_rx )
int_cont_set( int_con_periphe )
sci_rxsta_set( sci_rxstat_cenbl | sci_rxstat_port )
=( rec_state, 0b )
irq_on()

; main loop fragment
REP
    ; test to see if a new message has been received
    IF >=( rec_state, 17b )
        command_handle() ; decode and execute packet
    ENDIF

    do_controls() ; do the primary function of the controller
LOOP
```

Concluding Remarks

As you can see, using the SCI is a powerful way to handle RS232 data reception when the exact timing of the data stream is unknown. This asynchronous reception is the normal situation in the real world, which is why the SCI is so essential. Simple CAN (controller area networks) devices, RF links, data acquisition are all examples of applications that benefit greatly from asynchronous data reception.