AN044 - Implementing Ring Buffers for Background RS232 Reception and Transmission

Submitted by:

Glenn Clark
Protean Logic Inc.

The Asynchronous Nature of Communications

Serial communications often have an asynchronous nature. The receiver has very little control over the character timing of the data it is receiving. This requires the receiver to be able to capture the input stream independently of any other functions it might be performing. Furthermore, bandwidth is often a premium, which means that character to character idle time during transmission should be minimized. How can these goals be achieved with the TICkit line of processors?

Previous versions of the TICkit, or Parallax Basic Stamp, provided RS232 emulation routines that used cycle timing to generate and interpret RS232 signals on normal I/O lines. These methods proved very useful, but limited in the real world of asynchronous communications. With the development of the TICkit 63, new tools are now available to the Basic interpreter model. The TICkit 63 has a hardware block called the “SCI” or serial communications interface. This dedicated hardware takes care of the bit timing and start bit detection for receiving and transmitting RS232 bytes of data. Furthermore, the shift register portion of the SCI is double buffered for transmitting and triple buffered for receiving. This provides all the basis required to implement some conventional methods for receiving and transmitting RS232 data more or less continuously.

The SCI takes care of receiving or transmitting a byte of data in background while the TICkit performs other tasks in the foreground. This provides a great deal more time for the functions to be performed and still have time to process a continuous input stream. For example, at 9600 bps, there is 104 microseconds per bit time and 1040 micro seconds per byte. When using emulated routines, the receiver only had one half of a bit time to perform processing before needing to watch for the next incoming start bit. Using the SCI hardware, the receiver has a full 1040 microseconds to perform processing before the next character must be read. This represents a full 20 times increase in available processing time.

However, a problem still exists. What if the foreground processing of an application requires more than a single character time on an occasional basis? This is where the interrupt capability of the TICkit 63 comes into play. Remember the double and triple buffering mentioned above? The TICkit 63 can flag an interrupt whenever a byte is received in the first buffer. Provided the application services this interrupt before another byte is received (actually two bytes since the receive channel is triple buffered), no data is lost. Likewise, the transmit channel of the SCI will flag an interrupt whenever the output shift register is empty. This flag occurs as the double buffer’s data is moved into the shift register. Provided the application places another output character into the output register before the output shift register is again empty, transmission is continuous with no idle time between characters.

How are these flagged interrupts serviced?

Interrupts are very much like hardware generated subroutine or function calls. Between every token fetch, the TICkit 63 examines the interrupt flags. If one is set, the TICkit 63 applies rules of priority and, instead of executing the next token of the foreground sequence, it pushes the current program count pointer onto the stack and the function that was assigned to the highest priority flagged interrupt. These functions are designated simply by referring to them by the interrupt name. These functions are compiled just as any other function in the program with the exception that their beginning address is stored in a special area of the EEprom. This area contains “vectors” that correspond to the possible interrupt sources. When an interrupt is detected, the functions address is read from the proper vector and program execution jumps to that location. For the TICkit 63 the function names that are vectored are as follows:

IRQ - Executes when the IRQ line is low.

STACK_OVERFLOW - Executes when the program stack overflows

GLOBAL_INT - Executes whenever any of the internal hardware systems request an interrupt (such as the SCI)

TIMER_INT - Executes whenever the RTC (real time clock) period interval expires

Of course, any interrupt source can be turned off either by masking out the specific interrupt source or by disabling all interrupts with the irq_off() function. In fact all interrupts are initially disabled and must be enabled using the irq_on() function.

So, what is available is this: Whenever the SCI receives a new character or finishes sending one of two buffered characters, the TICkit can automatically execute a function to deal with the situation. In the case of SCI reception, the TICkit can automatically fetch the character that was received and store it or process it depending upon the application. Likewise, when the SCI signals a character has been sent, the application can load up the next character to be sent provided the character is available. There is a common method of buffering both received and transmitted data which can simplify this process quite a bit. This method employs a buffering technique called ring buffers and it makes RS232 reception and transmission appear to be synchronous to the application.

So What Exactly is a Ring Buffer?

The name "ring buffer" refers primarily to the nature of the input and output pointers into the buffer area. You can think of this arrangement like a ring or maybe a dog racing track. In the case of a receive buffer, the input pointer is like the rabbit, and the data read pointer is like the dog. When the dog catches the rabbit (input and read pointer are the same) there is no more data that has been received. If the rabbit catches the dog, there is trouble. This is referred to as a buffer overrun and it means your application did not process data fast enough to keep up with the input stream. The analogy of a ring is probably not as good as the analogy of say a staircase. The rabbit runs up the stair case then jumps off and goes around and back up the stairs. The dog follows the same path. To step away from the dog and rabbit symbols, what we are really dealing with is two pointers into memory that are always increasing. When a pointer reaches the end of the storage array, it simply resets to point to the bottom of the array. So there is really no "beginning" or ending point of the array, just an empty condition and an overrun condition that can occur at any point in the storage array when the pointers are equal.

The transmit ring buffer is similar to the receive buffer except it has an output pointer and a write pointer. Whenever a character finishes physically being sent, the data at the output pointer is placed into the transmit register and the output pointer is incremented. Whenever the application wishes to send data it places the data at the location pointed to by the write pointer and increments the write pointer. If the output pointer "catches" the write pointer, all data has been sent. If the write pointer "catches" the output pointer, the application is generating data too fast for the communications baud rate. In the condition where the application is generating too much data too fast, either the transmit ring buffer must be made larger, or the application must test the ring buffer condition to see if the ring buffer is full. If it is full, the application must delay generating further output.

The ring buffer methodology ends up looking like just two routines to the application interface. A get_rs232_input() and a send_rs232_output() routine. Additional functions can be added for testing an input buffer empty condition and for testing an output buffer full condition.

The actual ring buffer code:

The code for the ring buffers is pretty straight forward. The vector function, "GLOBAL_INT" is called whenever an internal hardware block flags an interrupt on a source whose mask has enabled interrupts. These masks and flags are controlled by the int_cont_set(), int_flag_set(), int_flag2_set(), int_mask_set(), int_mask2_set() functions. These five internal registers control the steering logic for all internal hardware interrupt sources in the TICkit 63. The interrupt routine is responsible for interrogating the interrupt flags to determine which of the sources created the interrupt condition. The interrupt routine is also responsible for clearing the source of the interrupt, resetting the flag (if necessary) and re-enabling interrupts. An important note here: Any interrupt automatically disables all interrupts. This implementation prevents a single interrupt from calling itself repeatedly. Furthermore, the irq_on() functions effect is delayed for exactly one token. So, the irq_on() function must be executed by all interrupt routines as a general rule to catch all interrupt conditions. However, the irq_on() function should only be called after the source of the interrupt just handled has been cleared and the flag has been reset. Also, the irq_on() function should only appear as the line immediately before an EXIT or ENDFUN directive. Some interrupt conditions can exist where, even after clearing the interrupt source, the interrupt flag is immediately set again by another source or even the current source. The irq_on() effect is delayed one token to allow a return to the main program sequence before another interrupt is handled. This prevents the stack from being overwhelmed by return addresses from within the interrupt routines themselves. Okay, here is the code:

```
DEF tic63_k
LIB fbasic.lib

DEF MAX_REC_ARRAY 10b
DEF MAX_XMIT_ARRAY 10b
GLOBAL byte rec_array[ MAX_REC_ARRAY ]  ;  storage for received data
GLOBAL byte xmit_array[ MAX_XMIT_ARRAY ] ; storage for data to be sent

GLOBAL byte rec_input_ptr 0b     ; receive ring buffer input pointer
GLOBAL byte rec_read_ptr 0b      ; receive ring buffer read pointer
GLOBAL byte xmit_output_ptr 0b   ; xmit ring buffer output pointer
GLOBAL byte xmit_write_ptr 0b    ; xmit ring buffer write pointer

GLOBAL byte ring_buffer_stat 0y00000110b ; status of ring buffer
DEF ring_stat_over 0y00000001b   ; receive ring buffer input pointer
DEF ring_stat_empty 0y00000010b  ; receive buffer empty
DEF ring_stat_under 0y00000100b  ; xmit buffer underrun
DEF ring_stat_full 0y00001000b   ; xmit buffer full


; The four standard interrupt handling functions follow
FUNC none irq    ; Executes when the int line goes low
BEGIN
ENDFUN

FUNC none global_int    ; Internal Global Interrupt
BEGIN
    IF b_test( int_flag_get(), int_flag_rx )
        ; just received a character
        =( rec_array[ rec_input_ptr ], sci_reg_get())
        ++(  rec_input_ptr )
        IF >=( rec_input_ptr, MAX_REC_ARRAY )
            =( rec_input_ptr, 0b )
        ENDIF

        b_clear( ring_buffer_stat, ring_stat_empty )
        IF ==( rec_input_ptr, rec_read_ptr )
           b_set( ring_buffer_stat, ring_stat_over )
        ENDIF

        irq_on()
        EXIT

    ELSEIF b_test( int_flag_get(), int_flag_tx )
        ; no characters transmitting
        IF ==( xmit_output_ptr, xmit_write_ptr )
            IF b_test( ring_buffer_stat, ring_stat_full )
                ; handle the character normally
            ELSE
                ; do nothing, no more to xmit
                b_set( ring_buffer_stat, ring_stat_under )
                int_mask_set( b_and( int_mask_get(), b_not( int_flag_tx )))
                ; clear the interrupt flag to prevent any more services
                irq_on()
                EXIT
            ENDIF
        ENDIF

        sci_reg_set( xmit_array[ xmit_output_ptr ])
        b_clear( ring_buffer_stat, ring_stat_full )
        ++( xmit_output_ptr )
        IF >=( xmit_output_ptr, MAX_XMIT_ARRAY )
            =( xmit_output_ptr, 0b )
        ENDIF

        irq_on()
        EXIT
    ELSE
        ; unknown interrupt source so ignore
        irq_on()
        EXIT
    ENDIF
ENDFUN


FUNC none rsSCI_send             ; Routine to send a single byte out the SCI port
    PARAM byte output_byte

BEGIN
    WHILE b_test( ring_buffer_stat, ring_stat_full )
    LOOP

    irq_off()
    =( xmit_array[ xmit_write_ptr ], output_byte )
    ++( xmit_write_ptr )
    IF >=( xmit_write_ptr, MAX_XMIT_ARRAY )
        =( xmit_write_ptr, 0b )
    ENDIF

    IF ==( xmit_output_ptr, xmit_write_ptr )
        b_set( ring_buffer_stat, ring_stat_full )
    ENDIF

    IF b_test( ring_buffer_stat, ring_stat_under )
        b_clear( ring_buffer_stat, ring_stat_under )
        int_mask_set( b_or( int_mask_get(), int_flag_tx ))
    ENDIF

    irq_on()
ENDFUN


FUNC byte rsSCI_receive          ; Function to receive a single byte from SCI port
    PARAM byte wait
BEGIN
    IF wait
        WHILE b_test( ring_buffer_stat, ring_stat_empty )
        LOOP
    ELSE
        IF b_test( ring_buffer_stat, ring_stat_empty )
            =( exit_value, 0b )
            EXIT
        ENDIF
    ENDIF

    irq_off()
    =( exit_value, rec_array[ rec_read_ptr])
    ++( rec_read_ptr )
    IF >=( rec_read_ptr, MAX_REC_ARRAY )
        =( rec_read_ptr, 0b )
    ENDIF

    IF ==( rec_read_ptr, rec_input_ptr )
        B_set( ring_buffer_stat, ring_stat_empty )
    ENDIF

    irq_on()
ENDFUN


FUNC none rsSCI_string           ; send a string of characters through the SCI
    PARAM word pointer
    LOCAL word tpointer
```

```
        LOCAL byte data
    BEGIN
        =( tpointer, pointer )
        =( data, ee_read( tpointer ))
        WHILE data
            rsSCI_send( data )
            ++( tpointer )
            =( data, ee_read( tpointer ))
        LOOP
    ENDFUN



    FUNC none timer_int    ; Executes when timer interval expires
    BEGIN
    ENDFUN

    FUNC none stack_overflow    ; Executes when stack overflows
    BEGIN
        reset()    ; only sure way to recover
    ENDFUN



    FUNC none main    ; First Function to execute on startup
    BEGIN
        ; enable and setup the SCI hardware for the ring buffer
        sci_baud_set( 32b )   ; 9600 baud
        sci_rxsta_set( sci_rxstat_cenbl | sci_rxstat_port )
        sci_txsta_set( sci_txstat_enabl )
        int_mask_set( int_flag_rx )
        int_cont_set( int_con_periphe )

        ;pin_low( pin_a6 )

        irq_on()
        rsSCI_send( 'H' )
        rsSCI_send( 'e' )
        delay( 200 )
        rsSCI_send( 'l' )
        delay( 200 )
        rsSCI_send( 'l' )
        rsSCI_send( 'o' )
        REP
            rsSCI_send( rsSCI_receive( true ))
        LOOP
    ENDFUN
```