Column #117, January 2005 by Jon Williams:

Timing is Everything

The dreaded "I" word ... yes, everybody talks about it ... there's lots of bloviating about it ... but what can we actually do with interrupts? Well, quite a lot actually – if we're patient and work carefully. Thankfully, SX/B makes interrupt programming more manageable that we thought it could be.

It wasn't very long after the BASIC Stamp and other BASIC-language microcontrollers appeared that advanced users started asking about using interrupts. Well, neither the BASIC Stamp family nor – to my knowledge – any of the micros in the same class support true interrupts; it's just not practical in BASIC and I'm about to explain why. Please, please, please ... don't fill my e-mail basket with flame mail telling me that your favorite BASIC-language controller does do interrupts; let me qualify my statement.

Let's back up a bit for those that may be a bit new. An interrupt is just that: an event or condition that suspends (interrupts) a program, forces the code into special section (usually called an ISR, for Interrupt Service Routine), then goes back to what it was doing when the interrupt occurred. Sounds pretty simple and straightforward, right? Well, not quite.

Here's why the BASIC Stamp and similar microcontrollers don't support true interrupts (as I just described): What happens if we're doing a bit-bang serial input (as most micros in the

BASIC Stamp class do) and we get an interrupt? Well, if we process the interrupt our serial timing is going to get trashed and we will corrupt the data – this could lead to a very big problem. The same problem holds true for any time-oriented function; things like SERIN, SEROUT, PULSIN, PULSOUT, PAUSE, OWIN, OWOUT, etc. You get the idea.

How is this handled then? Well, the BS2p family has the ability to do what is called "pin polling." When enabled, the BASIC Stamp 2p will check pin states in between high-level instructions and act in accordance with the polling setup configuration (there are several options). This pseudo-interrupt process can be very useful. Now, I realize that some BASIC-language microcontrollers use hardware UARTs and timers, and this does help alleviate the interrupt issue I just described. That said, the use of internal hardware occasionally limits design flexibility as specific IO points on the micro are required. I'm not saying that any of this is bad ... it just is.

Give Me an "I"

Okay, now that you know why the BASIC Stamp doesn't support true interrupts, what about SX/B? For those of you who have checked it out you've no doubt seen that there is indeed interrupt support. And yes, I'm going to show you how to use one type of interrupt this month – and to do two things with it: receive and buffer serial data (coming from a BASIC Stamp host) and to multiplex an eight-digit, 7-segment LED display.

The warnings I gave about interrupts above apply to SX/B – the difference is that SX/B allows interrupts any time you configure them. So, if you're going to be using interrupts in an SX/B program, you should not be using any of the time-oriented functions I mentioned earlier (note that SX/B does not have the 1-Wire commands of the BS2p).

As this is going to be a bit of a ride, let's get right to it. I was in my favorite store the other day (yes, Tanner Electronics in Dallas) and found a surplus eight-digit, 7-segment LED display that cost a dollar; that's right, one dollar. How could I not buy it? The question now was control. It's a common-cathode display, which means that the segment cathodes for all eight digits are tied together. The only way to use such a display properly is with a multiplexing controller. Of course I could use MAX7219, but they're not cheap and not easy to find anymore. Why not roll up my sleeves and create my own controller?

The idea was to create a serial LED controller that is AppMod protocol compatible, which means it could controlled from one BASIC Stamp pin, and even share that pin with the line follower we created last month. While on the surface this seems a simple task, it does present a serious challenge. To use the display properly, each active digit has to be refreshed at a

regular rate. If we weren't doing anything else and the display was static we could handle this in a program loop, especially in a compiled language like SX/B. The "problem" is that we want to be able to receive and buffer serial data at the same time. What this means for us, then, is that we will create an interrupt-driven program that multiplexes the display and handles the serial input.

Bit-Bang Serial – Interrupt Style

Take a look at Figure 117.1; this shows the structure of a serial byte in True (input idle state is high) mode. The stream begins with a start bit; this actually lets the receiver sync up and get ready for the incoming data bits, which will arrive LSB to MSB. At the end of the stream is a Stop bit period. Under non-interrupt conditions the processor will simply loop until the serial line goes low to indicate a start bit. A timer is set for 1.5 bit periods so that the first bit is sampled in the middle of its period. After that, the timer is reloaded with the bit period and the rest of the sampling happens in a loop. If you want to see this for yourself, use SERIN in an SX/B program and look at the assembly code that gets generated.

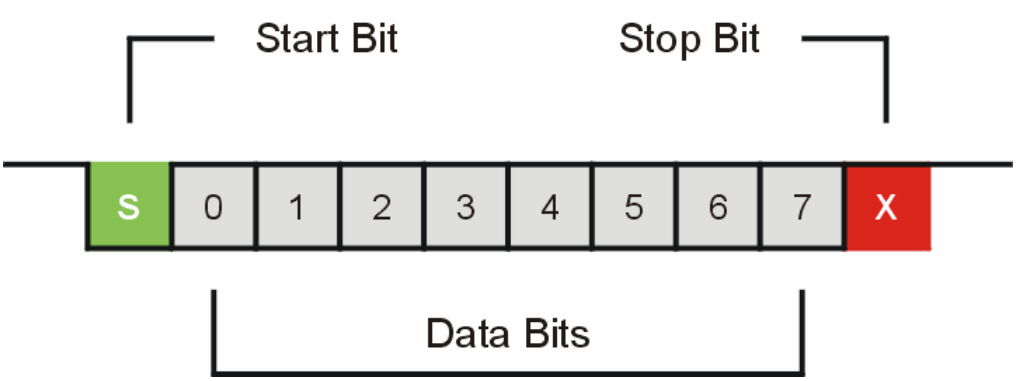

Figure 117.1; Structure of a serial byte in True mode

But in our project we can't sit around waiting for the bit to come in as we've got to update the display periodically. What we have to do is sample the serial line at a rate that will let us accurately capture the incoming data. So how fast do we sample? Well, I actually checked with programmers that are much better at this stuff than me, and the consensus was that when doing interrupt-driver serial input, one should sample the serial line at least four times per bit period.

Okay, decision time. In order to know how fast to sample the serial line we need to know what baud rate we want to support. For this project I decided to go with 9600 baud as this is somewhat standard for serial accessories and supported by most micros. And if we can sample at 9600 baud, then lower baud rates will be no problem; they'll simply have longer bit timing periods.

At 9600 baud the bit period is 104 microseconds (1 / 9600). If we want to sample four times per bit period, we have to do that every 26 microseconds. So, that's our first hurdle: setup the interrupt so that it activates every 26 microseconds. To do this we're going program the SX to create a periodic interrupt based on an internal value called the RTCC (Real Time Clock/Counter). This eight-bit value can be incremented by a change of state on an external pin, or by the oscillator that runs the SX. Since we can have a range of speeds to run the SX, we also have the ability to divide the oscillator frequency before sending it to the RTCC. This is called the prescaler, and usually comes into play when we're running the SX at very high speeds (e.g., 50 MHz).

For this project we'll run at 4 MHz so the prescaler won't be required. What we'll have to do is setup the OPTION register in the SX to enable RTCC updates on the internal clock without being divided by the prescaler. Here's how:

```
OPTION = %10001000
```

This configuration allows RAM address $00 to access the RTCC (bit 7 = 1), enable interrupt when the RTCC rolls over to zero (bit 6 = 0), increment RTCC on internal instruction cycle (bit 5 = 0), and set the divide rate for the RTCC to 1:1 (bit 3 = 1). The SX28 documentation (download from Ubicom) goes into all the details of the OPTION register.

Okay, now that we've enabled interrupts, how do we make that happen at the desired interval of 26 microseconds? Here's what an empty ISR block looks like in SX/B:

```
INTERRUPT
  ' ISR code
RETURNINT 104
```

The key is actually at the end, the value following RETURNINT. This tells the SX how many cycles to run before generating an interrupt. How then, did we come up with 104? We start with the clock frequency of our project: 4 MHz. At this rate, each instruction cycle takes 0.25 microseconds. Since we want our interrupt to trigger every 26 microseconds, we divide the instruction cycle speed into that. So, 26 divided by 0.25 is 104. This works because it is less than 255 (the maximum value of the RTCC). If you're ever doing a project where your

interrupt cycles calculate to greater than 255, you either have to reduce the oscillator speed or enable the RTCC prescaler.

At this point our program will be interrupted every 26 microseconds, a rate that we've determined is fast enough to sample the serial input line enough to accurately capture data at 9600 baud Okay, let's do it.

```
ISR_Start:
  ASM
    BANK $00
    MOVB C, Sin
    TEST rxCount
    JNZ RX_Bit
    MOV W, #9
    SC
    MOV rxCount, W
    MOV rxTimer, #BitTm15

RX_Bit:
    DJNZ rxTimer, Multiplex
    MOV rxTimer, #BitTm
    DEC rxCount
    SZ
    RR rxByte
    SZ
    JMP Multiplex

RX_Buffer:
    MOV FSR, #rxBuf
    ADD FSR, rxHead
    MOV IND, rxByte
    BANK $00
    INC rxHead
    CLRB rxHead.4
  ENDASM
```

Even though SX/B allows high-level code in the ISR, we're not going to do that for the serial input. Why not? Well, there are two reasons: with assembly language we can be a tiny bit more code-efficient, and – even more importantly – the code was already written and working, so why not just use it?

Let me pause for a second and suggest that if you're serious about programming the SX, you should consider the books that Parallax makes available: *Exploring the SX Microcontroller* by

Al Williams (no, we're not related but he lives in Texas too), and *Programming the SX Microcontroller* by Guenther Daubach. Both authors are great guys and very active in the Parallax support forums. You can get an SX starter kit that includes both books, and if you're on a budget, Al's book is available as a free PDF download.

Okay, back the code. On entering the ISR we want to make sure that we're pointing at the serial variables so we issue a BANK $00 statement to do that. Then we sample the serial line by copying it into the SX Carry bit. When the serial line is idle the Carry bit will now hold a value of 1. Let's continue to go through the code as if we're in the idle state. The TEST instruction will set the Zero bit if the register tested holds a value of zero. In our program, the variable rxCount is used to count-down the bits as they're coming in; when rxCount is zero we are not currently receiving a byte. The next instruction, JNZ, will force the program to jump to RX_Bit when the Z flag is not set – this happens when we are receiving (rxCount > 0). Since the Z flag is currently set, we will fall through the JNZ to where we load the value of 9 (start bit plus eight data bits) into the W register. After that we will check the Carry bit; if it is one (and it currently is), we will skip the loading of rxCount and load the bit timer (rxTimer). With rxTimer loaded with drop into RX_Bit where the timer is decremented and if not zero, the serial routine jumps to the label called Multiplex.

This process will repeat every interrupt cycle until a start bit is received. You may be wondering – as I did – why the rxTimer gets loaded when there is no start bit. Well, the reason there is no bail-out on a no start bit condition is that it actually adds more code than simply allowing the rxTimer to be loaded and the routine to exit.

Now a start bit arrives, let's see what happens. This time through we will move 0 into the Carry bit. As rxCount is still zero, we will not jump to RX_Bit, but we will end up moving 9 into rxCount (via W). Now we load the rxTimer with 1.5 bit periods, decrement the timer for this interrupt cycle and exit. On the next interrupt we will have 9 in rxCount so the code will jump right to RX_Bit after sampling the serial line, and then the rxTimer will be decremented again. This will continue until rxTimer is zero.

At this point we're actually in the middle of the first data bit (the LSB). We will reload the rxTimer with the bit timing, and then decrement the rxCount to account for the start bit. The program will drop through the SZ (skip if zero) instruction since rxCount is at eight, and then move the data bit (currently sitting in Carry) into rxByte with the RR (rotate right) instruction. Finally, the program will drop through another SZ instruction and jump out of the serial routine to the Multiplexer.

This process will continue for eight bits. After the final bit arrives rxCount will be zero, and the code will end up skipping the JMP Multiplex instruction and move to RX_Buffer. This

code will save the incoming byte to a 16-byte circular buffer. This will let our foreground program handle important business while bytes are streaming in. That said, it's a circular buffer, and if we don't pull data from it before it fills, it can end up overwriting itself. That won't be a problem with our display.

The code at RX_Buffer uses indirect addressing via the FSR (File Select Register) to update the circular buffer. We start by moving the location of the first byte of the buffer into the FSR, then adding the head pointer (rxHead) to that. The MOV IND instruction takes the value of rxByte and puts it into the location being pointed to by the FSR. Then we update the position of the head pointer and make sure that it stays within a 0 to 15 range by clearing bit 4. At the end of our serial section we can terminate the assembly code block of our ISR with the ENDASM instruction.

Did you just take a big breath? I did. There will come a point when this all seems trivial, but until you get to that point you might want to review it over a few times. It wouldn't hurt to map the position of the counters and bits on paper so that you make sure you understand it. By understanding how this works you'll be able to modify it to suit your needs for a different application.

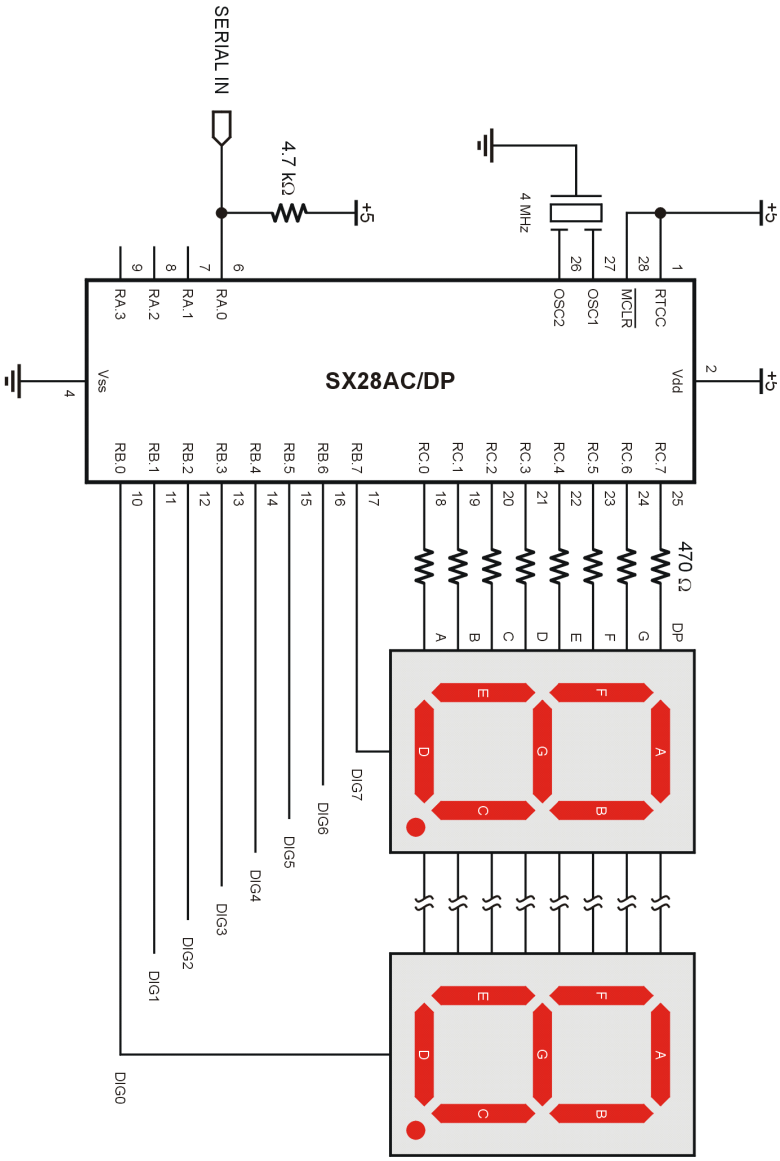

Figure 117.2: SX28AC/DP with Multiplexed LEDs

Taking the Mystery Out of Multiplexing

Remember that our project has another important task: we have to multiplex the LED display which means selecting the active column (cathode) and then activating the appropriate segments (anodes) to create the desired pattern. We will handle this "in the background" via the ISR. This is actually much easier than the serial code though, and can be done with SX/B instructions.

```
Multiplex:
  INC digPntr
  IF digPntr <= 7 THEN Next_Digit
  digPntr = 0

Next_Digit:
  Cathodes = NoDig
  IF digPntr > limit THEN ISR_Exit
  Anodes = anoBuf(digPntr)
  IF digBlank = 1 THEN ISR_Exit
  READ DigCtrl + digPntr, Cathodes

ISR_Exit:
  RETURNINT 104
```

The first step is to increment the variable called digPntr which points at the current active column. The next line will compare the value of digPntr to seven (last legal column value), and if it digPntr is less than or equal to seven then we will move on to Next_Digit. Once digPntr hits eight we will reset it before moving on. If you modify the program for a smaller display, be sure to update this section of code.

The code at Next_Digit actually updates the display. We start by turning it off – this will prevent ghosting when we change the anode (segments) values. Next we're going to check a couple of values that can be set by the user via serial commands (more on that later). The first is the blanking bit, which turns the display off without affecting the contents of the display buffer. When blanking is enabled we jump right out of the ISR before enabling the current column. The next value checked is the column limit. This lets us decide who many columns to activate (starting from the rightmost position). If the column pointer is beyond the column limit we jump out of the ISR before activating the current column.

Finally, when blanking is off and the current column is active we will move the contents of the anodes buffer for that column to the display. Then we activate the column by setting its cathode control line to zero and we're done.

Take another breath. The really cool thing about all this is that the multiplexing code was written in BASIC – SX/B BASIC. That's really neat. Now, before you get too excited there is something very important to keep in mind: You must keep the longest path through the ISR to less than the number of cycles assigned to the ISR activation, minus 3 cycles (101 for this project). If we go over, what will happen is that an interrupt cycle may get ignored if it occurs while the current interrupt is still running (the SX disables interrupt while running the ISR), and this could be catastrophic for programs that require specific interrupt timing. You can check the length of the ISR by looking at the assembly output – using Ctrl-L in the SX-Key Editor is a quick way to do this. In this program the final address of the ISR is $0056 (86), so we're in good shape.

Back to Easy Street

With the interrupt routine coded and working the rest of the program is downright simple. Let's go through the important parts. In the beginning, we want to wait for the proper header string before processing any commands – this keeps us AppMod compatible. The header for the LED controller is "!SS8" and will be followed by a command, and one or more data bytes.

```
Main:
  GOSUB Get_Byte, @cmd
  IF cmd <> "!" THEN Main
  GOSUB Get_Byte, @cmd
  IF cmd <> "S" THEN Main
  GOSUB Get_Byte, @cmd
  IF cmd <> "S" THEN Main
  GOSUB Get_Byte, @cmd
  IF cmd <> "8" THEN Main
```

This code looks very similar to what we did in the line follower program. It simply goes through the input until the sequence "!SSR" is received. Remember that our serial input is being placed into a circular buffer by the ISR, so we need to write a routine to retrieve the first available byte.

```
_Get_Byte:
  IF rxTail = rxHead THEN _Get_Byte
  regAddr = __PARAM1
  temp1 = rxBuf(rxTail)
  INC rxTail
  rxTail = rxTail & $0F
  __RAM(regAddr) = temp1
  RETURN
```

Just as we did last time, we can pass the desired variable address by using the "@" preface.

In the Get_Byte routine this causes the address of that byte to be saved. Then the routine compares the value of the tail pointer (where we will get the byte) to the head pointer (where the next incoming byte will be saved). If these values are equal, the buffer is empty and we'll loop to the top of the routine until something arrives.

When the buffer isn't empty we will move the byte currently sitting in the tail position to a temporary variable. As we did with the head pointer in the ISR, we have to update the position of the tail pointer and force it to stay within the 0 to 15 range of valid buffer addresses. Finally, we move the serial byte (sitting in temp1) to the variable specified by the caller by using the system __RAM() address. This is new in SX/B version 1.1, and makes it easy to modify or retrieve any SX RAM address.

Once we have the header we will grab the command byte and then jump to a routine that takes care of any data or processing required by the command.

```
Get_Cmd:
  GOSUB Get_Byte, @cmd
  IF cmd = "R" THEN Do_Reset
  IF cmd = "C" THEN Do_Config
  IF cmd = "X" THEN Do_Blanking
  IF cmd = "W" THEN Do_Write
  IF cmd = "B" THEN Do_Block
  IF cmd = "<" THEN Do_ShiftL
  IF cmd = ">" THEN Do_ShiftR
  GOTO Main
```

To some this structure may look a bit clunky, but keep in mind that SX/B is designed to be very close to assembly language. This lets the code compile very cleanly, and more importantly, it lets us learn from the compiled code. In many instances you'll see that there is a 1-to-1 relationship between SX/B instructions and SX instructions. SX/B is built for fancy – it's built for speed.

Let's have a look at the valid instructions, starting with "R" for reset. The purpose of this command is to clear the serial buffer, clear the display buffer, and set the display mode for each column.

```
Do_Reset:
  GOSUB Get_Byte, @colMode
  rxHead = 0
  rxTail = 0
```

```
  limit = 0
  colEnable = %11111111
  FOR idx = 0 TO 7
    digBuf(idx) = 0
  NEXT
  GOSUB Update_Anodes
  GOTO Main
```

Note that the reset command allows us to specify the column mode bits. Since our display is eight digits wide, a single byte works perfectly. A "0" bit (default) indicates that the column is decoded, that is, the value in the data buffer will be translated to the appropriate patterns for the values 0 to F (15). A "1" bit in the mode byte will cause the raw bits value to be transferred to the display. This feature allows us to define other alpha characters, and special patterns that may be used in animations (the BS2 demo program shows off this feature).

The rest of the reset code clears the serial input buffer, sets the display limit to one column, enables all columns (up to the column limit), and clears the display buffer (digBuf). After these changes are made we have to call the Update_Anodes subroutine as the anodes buffer is what gets transferred to the display in the ISR.

```
_Update_Anodes:
  FOR temp1 = 0 TO 7
    temp2 = 0
    temp3 = colEnable >> temp1
    IF temp3.0 = 0 THEN _Put_Dig
    temp2 = digBuf(temp1)
    temp3 = colMode >> temp1
    IF temp3.0 = 1 THEN _Put_Dig

_Decode_Dig:
    temp2 = temp2 & $0F
    READ SegMaps + temp2, temp2

_Put_Dig:
    anoBuf(temp1) = temp2
  NEXT
  RETURN
```

This subroutine probably looks a bit more complicated than it is. The code loops through eight columns, first checking to see if a column is enabled. If it isn't, the anodes buffer for that column is cleared. If the column is enabled, then we need to check the mode for that column. When the mode bit is "0" we will take the low nibble of the column value and use it

as an index into the patterns table that make up the shapes for the numbers 0 through F. If the mode bit for a column is "1" the raw value is transferred to the anodes buffer.

After the display is reset, we may want to change the configuration. Let's say that we wanted to enable the rightmost three columns in decoded mode. Here's how we could do that using a BASIC Stamp:

```
  SEROUT Sout, Baud, ["!SS8C", 2, 0, $FF]
```

The first byte in the stream limits us to the third column (column 2), the next byte specifies that all columns are decoded (all bits are 0), and that all visible columns are enabled. Let's look at the code that processes the "C" (configuration) command:

```
Do_Config:
  GOSUB Get_Byte, @limit
  GOSUB Get_Byte, @colMode
  GOSUB Get_Byte, @colEnable
  limit = limit MAX 7
  GOSUB Update_Anodes
  GOTO Main
```

As you can see, there is no magic here – we simply grab the bytes coming in and move them to their respective variables. The only byte of concern is the column limit which has a maximum value of seven. The MAX operator handles this for us. Since the configuration command can change column display modes and enable bits, we need to call Update_Anodes again to refresh the anodes buffer.

Before we run out of space, let's actually put a value into the display, shall we? We're going to use the "W" (write) command that will let us specify a column and a value to write to it.

```
Do_Write:
  GOSUB Get_Byte, @idx
  GOSUB Get_Byte, @cmd
  IF idx > 7 THEN Main
  digBuf(idx) = cmd
  GOSUB Update_Anodes
  GOTO Main
```

After retrieving the column and data values, we just need to make sure that a column value beyond our display has not been specified. If this happens we exit to Main and leave the raw digits buffer alone. If the column index is good, then we update the digits buffer, and as we did before we update the anodes buffer as well – this updates the display.

Before we go, let's look a bit of PBASIC code that can run the project – it will make sense of some of the main features.

```
idx2 = 0
FOR cntr = 1 TO 100
  FOR idx = 0 TO 2
    SEROUT Sout, Baud, ["!SS8W", idx, cntr DIG idx]
  NEXT
  SEROUT Sout, Baud, ["!SS8W", 7, 1 << idx2]
  idx2 = idx2 + 1 // 6
  LOOKDOWN cntr, <[10, 100, 1000], last
  LOOKUP last, [$FE, $FC, $F8], cMode
  LOOKUP last, [$C1, $C3, $C7], cEnable
  SEROUT Sout, Baud, ["!SS8C", 7, cMode, cEnable]
  PAUSE 100
NEXT
```

The purpose of this code is to display a 3-digit counter in the display, as well as run a little animated "bug" on the left. The main loop handles the counter. At the top of the main loop is a smaller inner loop that uses the Write command to send the counter digits to the display. Notice how convenient the DIG operator is for us in this application. The next section animates the outside segments of the left-most display. It's a very simple attention getter.

Now that we have data in the SS8 buffer, we need to configure the display so that digits are shown on the right, and the animated "bug" on the left. With LOOKDOWN we can determine how many columns the current count value occupies, and with that value LOOKUP will give us the proper column mode and enable bytes. This lets us blank leading zeros and create a more professional looking display.

One of the things that you probably noticed is that the column mode and column enable bytes can – in some cases – be used to accomplish the same thing. To be honest, the column enable feature was a late addition to the project, and this came after a lot of display experimenting. One technique I experimented with while developing the code was pre-writing to the display, then revealing the display column by column by updating the column enable byte.

Well, it's up to you now. I will admit that programming the SX – even with SX/B – can be challenging, but the rewards are really worthwhile. Using this display project as a guide, you can build any number of serial accessories that required buffered input. Be sure to download the SX documentation from Ubicom and do checkout the books I told you about; they will make your journey into SX mastery far easier.

By the way, Happy New Year! And until next time ... Happy Stamping.

```
' =============================================================================
'
'   File..... SS8_Test.BS2
'   Purpose... Demonstrates SS8 Terminal module
'   Author.... Jon Williams -- Parallax, Inc.
'   E-mail.... jwilliams@parallax.com
'   Started... 11 NOV 2004
'   Updated... 15 NOV 2004
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =============================================================================


' -----[ Program Description ]---------------------------------------------
'
' Test program for the SS8 LED terminal.


' -----[ Revision History ]------------------------------------------------


' -----[ I/O Definitions ]-------------------------------------------------

Sout            PIN     15


' -----[ Constants ]------------------------------------------------------

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    T1200       CON     813
    T2400       CON     396
    T4800       CON     188
    T9600       CON     84
    T19K2       CON     32
    TMidi       CON     12
    T38K4       CON     6
  #CASE BS2SX, BS2P
    T1200       CON     2063
    T2400       CON     1021
    T4800       CON     500
    T9600       CON     240
    T19K2       CON     110
    TMidi       CON     60
    T38K4       CON     45
#ENDSELECT

Inverted        CON     $4000
Open            CON     $8000
```

```
Baud            CON     Open + T9600            ' AppMod compatible

DispOn          CON     0                       ' display blanking control
DispOff         CON     1


' ******************
' 7-Segment Patterns
' ******************

_Spc            CON     %00000000
_DQuote         CON     %00100010
_SQuoteL        CON     %00000010
_SQuoteR        CON     %00100000
_Dash           CON     %01000000
_DP             CON     %10000000
_Fslash         CON     %01010010

_0              CON     %00111111
_1              CON     %00000110
_2              CON     %01011011
_3              CON     %01001111
_4              CON     %01100110
_5              CON     %01101101
_6              CON     %01111101
_7              CON     %00000111
_8              CON     %01111111
_9              CON     %01100111

_Equ            CON     %01000001
_Qmark          CON     %00000000

_A              CON     %01110111
_C              CON     %00111001
_E              CON     %01111001
_F              CON     %01110001
_H              CON     %01110110
_I              CON     %00000110
_J              CON     %00011110
_L              CON     %00111000
_O              CON     %00111111
_P              CON     %01110011
_S              CON     %01101101
_U              CON     %00111110

_BrktL          CON     %00111001
_Bslash         CON     %01100100
_BrktR          CON     %00001111
_Uline          CON     %00001000

_b              CON     %01111100
```

```
_c2             CON     %01011000
_d              CON     %01011110
_h2             CON     %00000000
_i2             CON     %00000000
_n              CON     %01010100
_o2             CON     %00000000
_r              CON     %01010000
_t              CON     %01111000
_u2             CON     %00000000

_DegSym         CON     %01100011



' -----[ Variables ]------------------------------------------------------

cntr            VAR     Byte
last            VAR     Byte
cMode           VAR     Byte
cEnable         VAR     Byte
idx             VAR     Nib
idx2            VAR     Nib
char            VAR     Byte


' -----[ EEPROM Data ]-----------------------------------------------------


' -----[ Initialization ]---------------------------------------------------

Reset:
  SEROUT Sout, Baud, ["!SS8R", $FF]             ' clear bufs, segment mode
  PAUSE 1


' -----[ Program Code ]-----------------------------------------------------

Main:
  GOSUB Show_Start

  ' chaser underline

  FOR idx = 1 TO 5
    FOR idx2 = 2 TO 0
      SEROUT Sout, Baud, ["!SS8W", idx2, _Uline]
      PAUSE 100
      SEROUT Sout, Baud, ["!SS8W", idx2, 0]
    NEXT
    PAUSE 200
  NEXT
```

```
' reset, prep for counter and animation display

SEROUT Sout, Baud, ["!SS8R", $FF]             ' clear buffers
PAUSE 1
SEROUT Sout, Baud, ["!SS8C", 0, 0, 0]         ' disable all columns

' display counter and animated "bugs"

idx2 = 0
FOR cntr = 1 TO 100
  FOR idx = 0 TO 2
    SEROUT Sout, Baud, ["!SS8W", idx, cntr DIG idx]
  NEXT
  SEROUT Sout, Baud, ["!SS8W", 7, 1 << idx2]  ' animated bug
  SEROUT Sout, Baud, ["!SS8W", 6, 1 << (idx2 + 3 // 6)]
  idx2 = idx2 + 1 // 6
  LOOKDOWN cntr, <[10, 100, 1000], last       ' get last digit column
  LOOKUP last, [$FE, $FC, $F8], cMode         ' get column mode
  LOOKUP last, [$C1, $C3, $C7], cEnable       ' get enable bits
  SEROUT Sout, Baud, ["!SS8C", 7, cMode, cEnable]
  PAUSE 100
NEXT
SEROUT Sout, Baud, ["!SS8C", 7, $F8, $07]
PAUSE 500

GOSUB Show_End
PAUSE 1000

' shift display left and right

FOR idx = 1 TO 5
  SEROUT Sout, Baud, ["!SS8<", 1]
  PAUSE 100
NEXT
FOR idx = 1 TO 5
  SEROUT Sout, Baud, ["!SS8>", 1]
  PAUSE 100
NEXT

' flash display using blanking bit

FOR idx = 1 TO 5
  SEROUT Sout, Baud, ["!SS8X", DispOff]
  PAUSE 250
  SEROUT Sout, Baud, ["!SS8X", DispOn]
  PAUSE 250
NEXT

GOTO Main
END
```

```
' -----[ Subroutines ]------------------------------------------------------

' Write "StArt" on LED display

Show_Start:
  SEROUT Sout, Baud, ["!SS8X", DispOff]
  SEROUT Sout, Baud, ["!SS8C", 7, $FF, $FF]
  SEROUT Sout, Baud, ["!SS8B", 8, 0, 0, 0, _t, _r, _A, _t, _S]
  SEROUT Sout, Baud, ["!SS8X", DispOn]
  RETURN


' Write "End" on LED display

Show_End:
  SEROUT Sout, Baud, ["!SS8X", DispOff]
  SEROUT Sout, Baud, ["!SS8C", 7, $FF, $FF]
  SEROUT Sout, Baud, ["!SS8B", 8, _d, _n, _E, 0, 0, 0, 0, 0]
  SEROUT Sout, Baud, ["!SS8X", DispOn]
  RETURN
```

```
' =============================================================================
'
'   File..... SS8_TERM.SXB
'   Purpose... Eight character, 7-Segment Serial Display
'   Author.... Jon Williams
'              (c) Parallax, Inc. -- All Rights Reserved
'   E-mail.... jwilliams@parallax.com
'   Started... 11 NOV 2004
'   Updated... 15 NOV 2004
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Serial "terminal" that controls up to eight, 7-segment LED displays,
' 64 discrete LEDs, or a mixture of both.
'
' Commands to the SS8 are electrically and syntactically compatible with
' the Parallax AppMod protocol, albeit at 9600 baud (open, true).
'
' Valid serial commands from host:
'
' "!SS8R" -- reset SS8, set column mode
' "!SS8C" -- configure last column, column modes, column enable
' "!SS8X" -- display blanking control
' "!SS8W" -- write to specific digit
' "!SS8B" -- write block block to display
' "!SS8<" -- shift display left n places
' "!SS8>" -- shift display right n places
'
' The SS8 does not send data back to the host.


' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE          SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ            4_000_000


' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

Sin             VAR     RA.0                    ' serial in
Cathodes        VAR     RB                      ' LED cathodes
TRIS_Cath       VAR     TRIS_B
```

```
Anodes          VAR     RC                      ' LED anodes
TRIS_Ano        VAR     TRIS_C


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

B2400           CON     16                      ' 2400 Baud
B9600           CON     4                       ' 9600 Baud
BitTm           CON     B9600                   ' samples per bit
BitTm15         CON     3*BitTm/2               ' 1.5 bits (SASM constant)

DigTm           CON     77                      ' 77 x 26 us = 2 ms

Blank           CON     %00000000               ' all segments off
NoDig           CON     %11111111               ' all digits off


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

rxCount         VAR     Byte                    ' bits to receive
rxTimer         VAR     Byte                    ' bit timer for ISR
rxByte          VAR     Byte                    ' serial byte
rxHead          VAR     Byte                    ' available slot
rxTail          VAR     Byte                    ' next byte to read
rxBuf           VAR     Byte(16)                ' circular buffer

flags           VAR     Byte

cmd             VAR     Byte                    ' command byte from host
limit           VAR     Byte                    ' last dig displayed
colMode         VAR     Byte                    ' column mode (0 = decoded)
colEnable       VAR     Byte                    ' column enable (1 = enabled)
idx             VAR     Byte                    ' loop counter

digTimer        VAR     Byte                    ' digit timer for ISR
digPntr         VAR     Byte                    ' digit pointer
digBlank        VAR     flags.0                 ' blank if bit0 = 1
digBuf          VAR     Byte(8)                 ' digit buffer (raw)
anoBuf          VAR     Byte(8)                 ' anodes buffer

regAddr         VAR     Byte                    ' register address
temp1           VAR     Byte                    ' parameter(s)
temp2           VAR     Byte
temp3           VAR     Byte
temp4           VAR     Byte
```

```
' -----------------------------------------------------------------------------
  INTERRUPT
' -----------------------------------------------------------------------------

' ISR is setup to receive N81, true mode.

ISR_Start:
  ASM
    BANK $00
    MOVB C, Sin                             ' sample serial input
    TEST rxCount                            ' receiving now?
    JNZ RX_Bit                              ' yes if rxCount > 0
    MOV W, #9                               ' start + 8 bits
    SC                                      ' skip if no start bit
    MOV rxCount, W                          ' got start, load bit count
    MOV rxTimer, #BitTm15                   ' delay 1.5 bits

RX_Bit:
    DJNZ rxTimer, Multiplex                 ' update bit timer
    MOV rxTimer, #BitTm                     ' reload bit timer
    DEC rxCount                             ' mark bit done
    SZ                                      ' if last bit, we're done
    RR rxByte                               ' move bit into rxByte
    SZ                                      ' if not 0, get more bits
    JMP Multiplex

RX_Buffer:
    MOV FSR, #rxBuf                         ' get buffer address
    ADD FSR, rxHead                         ' point to head
    MOV IND, rxByte                         ' move rxByte to head
    BANK $00
    INC rxHead                              ' update head
    CLRB rxHead.4                           ' keep 0 - 15
  ENDASM

Multiplex:
  INC digPntr                               ' point to next digit
  IF digPntr <= 7 THEN Next_Digit           ' still in  column range?
  digPntr = 0                               ' no, reset to first column

Next_Digit:
  Cathodes = NoDig                          ' deactive all
  IF digBlank = 1 THEN ISR_Exit             ' terminate if blanking on
  IF digPntr > limit THEN ISR_Exit          ' no more active digits
  Anodes = anoBuf(digPntr)                  ' update anodes
  READ DigCtrl + digPntr, Cathodes          ' enable current digit

ISR_Exit:
  RETURNINT 104                             ' 26 uS @ 4 MHz
```

```
' ============================================================================
  PROGRAM Start
' ============================================================================

SegMaps:                                        ' segments maps
'        .gfedcba
  DATA  %00111111                               ' 0
  DATA  %00000110                               ' 1
  DATA  %01011011                               ' 2
  DATA  %01001111                               ' 3
  DATA  %01100110                               ' 4
  DATA  %01101101                               ' 5
  DATA  %01111101                               ' 6
  DATA  %00000111                               ' 7
  DATA  %01111111                               ' 8
  DATA  %01100111                               ' 9
  DATA  %01110111                               ' A
  DATA  %01111100                               ' b
  DATA  %00111001                               ' C
  DATA  %01011110                               ' d
  DATA  %01111001                               ' E
  DATA  %01110001                               ' F

DigCtrl:
  DATA  %11111110                               ' column 0 on
  DATA  %11111101
  DATA  %11111011
  DATA  %11110111
  DATA  %11101111
  DATA  %11011111
  DATA  %10111111
  DATA  %01111111                               ' column 7 on


' ----------------------------------------------------------------------------
' Subroutines Jump Table
' ----------------------------------------------------------------------------

' This routine transfers the input buffer (digBuf) to the anodes
' buffer (anoBuf), checking the status of the cfg bit to determine
' whether the anoBuf gets raw data (cfg bit = 1) or dec/hex decoded
' (cfg bit = 0) 7-segment pattern.

Update_Anodes:
  GOTO @_Update_Anodes


' Use: GOSUB Get_Byte, @aVar
' -- if data is in buffer, the next byte is moved to 'aVar'
' -- will wait for byte to arrive if buffer is empty
```

```
Get_Byte:
  GOTO @_Get_Byte


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  PLP_A = %0001                               ' pull-up unused pins
  Anodes = Blank                              ' clear display
  TRIS_Ano = %00000000                        ' make outputs
  Cathodes = %11111111                        ' all off to start
  TRIS_Cath = %00000000                       ' make outputs
  GOSUB Update_Anodes                         ' prep display buffer
  OPTION = %10001000                          ' interrupt, no prescaler


Main:                                         ' wait for header ("!SS8")
  GOSUB Get_Byte, @cmd
  IF cmd <> "!" THEN Main
  GOSUB Get_Byte, @cmd
  IF cmd <> "S" THEN Main
  GOSUB Get_Byte, @cmd
  IF cmd <> "S" THEN Main
  GOSUB Get_Byte, @cmd
  IF cmd <> "8" THEN Main


Get_Cmd:
  GOSUB Get_Byte, @cmd                        '  get command
  IF cmd = "R" THEN Do_Reset
  IF cmd = "C" THEN Do_Config
  IF cmd = "X" THEN Do_Blanking
  IF cmd = "W" THEN Do_Write
  IF cmd = "B" THEN Do_Block
  IF cmd = "<" THEN Do_ShiftL
  IF cmd = ">" THEN Do_ShiftR
  GOTO Main                                   ' command byte was invalid


Do_Reset:                                     ' reset display
  GOSUB Get_Byte, @colMode                    ' get column mode bits
  rxHead = 0                                  ' clear serial buffer
  rxTail = 0
  limit = 0                                   ' view col 0 only
  colEnable = %11111111                       ' enable all columns
  FOR idx = 0 TO 7                            ' reset input buffer
    digBuf(idx) = 0
  NEXT
  GOSUB Update_Anodes                         ' update display
```

```
  GOTO Main


Do_Config:
  GOSUB Get_Byte, @limit                    ' get limit, 0 - 7
  GOSUB Get_Byte, @colMode                  ' get column mode bits
  GOSUB Get_Byte, @colEnable                ' get column enable bits
  limit = limit MAX 7                       ' keep limit legal
  GOSUB Update_Anodes                       ' update display
  GOTO Main


Do_Blanking:
  GOSUB Get_Byte, @cmd
  digBlank = cmd.0                          ' save blanking bit
  GOTO Main


Do_Write:                                   ' write to a register
  GOSUB Get_Byte, @idx                      ' get register index
  GOSUB Get_Byte, @cmd                      ' get value for register
  IF idx > 7 THEN Main                      ' abort if out of range
  digBuf(idx) = cmd                         ' otherwise, update
  GOSUB Update_Anodes                       ' update display
  GOTO Main


Do_Block:                                   ' block write all regs
  GOSUB Get_Byte, @temp4                    ' get block size
  IF temp4 = 0 THEN Main                     ' prevent illegal block
  IF temp4 > 8 THEN Main
  DEC temp4                                 ' adjust for loop
  FOR idx = 0 TO temp4                      ' loop through block
    GOSUB Get_Byte, @cmd                    ' get new value
    digBuf(idx) = cmd                       ' move to register
  NEXT
  GOSUB Update_Anodes                       ' update display
  GOTO Main


Do_ShiftL:                                  ' shift digit buffer left
  GOSUB Get_Byte, @temp4                    ' get block size
  IF temp4 = 0 THEN Main                     ' no shift
  IF temp4 > 8 THEN Main                    ' too many bytes
  FOR idx = 1 TO temp4                      ' shift buffer
    FOR temp1 = 7 TO 1 STEP - 1
      temp2 = temp1 - 1
      digBuf(temp1) = digBuf(temp2)         ' digBuf(n) = digBuf(n-1)
    NEXT
    digBuf(0) = 0                           ' clear end of buffer
  NEXT
```

```
  GOSUB Update_Anodes                          ' update display
  GOTO Main


Do_ShiftR:                                     ' shift digit buffer right
  GOSUB Get_Byte, @temp4                       ' get block size
  IF temp4 = 0 THEN Main                       ' no shift
  IF temp4 > 8 THEN Main                       ' too many bytes
  FOR idx = 1 TO temp4                         ' shift buffer
    FOR temp1 = 0 TO 6
      temp2 = temp1 + 1
      digBuf(temp1) = digBuf(temp2)            ' digBuf(n) = digBuf(n+1)
    NEXT
    digBuf(7) = 0                              ' clear end of buffer
  NEXT
  GOSUB Update_Anodes                          ' update display
  GOTO Main


' -----------------------------------------------------------------------------
' Page 1 Code
' -----------------------------------------------------------------------------

Page_1:
  ADDRESS $200


' This routine transfers the input buffer (digBuf) to the anodes
' buffer (anoBuf), checking the status of the cfg bit to determine
' whether the anoBuf gets raw data (cfg bit = 1) or dec/hex decoded
' (cfg bit = 0) 7-segment pattern.

_Update_Anodes:
  FOR temp1 = 0 TO 7                           ' update all digits
    temp2 = 0                                  ' start with blank
    temp3 = colEnable >> temp1                 ' align enable for test
    IF temp3.0 = 0 THEN _Put_Dig               ' test enable bit
    temp2 = digBuf(temp1)                      ' get raw value
    temp3 = colMode >> temp1                   ' align mode for test
    IF temp3.0 = 1 THEN _Put_Dig               ' test col mode bit

_Decode_Dig:
    temp2 = temp2 & $0F                        ' mask out high nib
    READ SegMaps + temp2, temp2                ' get dec/hex map

_Put_Dig:
    anoBuf(temp1) = temp2                      ' anode --> buffer
  NEXT
  RETURN
```

```
' Use: GOSUB Get_Byte, @aVar
' -- if data is in buffer, the next byte is moved to 'aVar'
' -- will wait for byte to arrive if buffer is empty

_Get_Byte:
  regAddr = __PARAM1                          ' save return address
  IF rxTail = rxHead THEN _Get_Byte           ' wait for byte
  temp1 = rxBuf(rxTail)                       ' get first available
  INC rxTail                                  ' point to next
  rxTail = rxTail & $0F                       ' keep 0 - 15
  __RAM(regAddr) = temp1                      ' move to target address
  RETURN
```